

EBOOK

TECNOLOGIA WEBSERVICES E RESTFUL



Sumário

Apresentação	5
UNIDADE 1 - PROGRAMAÇÃO NA WEB E SEUS COMPONENTES	6
Rápido crescimento da Internet	6
Internet nossa de cada dia	7
Requisição e Resposta	7
HTTP: Protocolo baseado em documentos	8
HTTP Request	9
Métodos do HTTP	9
Caminho	10
Cabeçalhos	10
Corpo da Requisição	11
HTTP Response	11
Código da resposta	12
Cabeçalho	12
Corpo da Resposta	13
Arquitetura Cliente-Servidor	14
Termos que você deve saber	15
HTTP	15
URI e URL	15
XML	16
JSON	16
SOAP	17
WSDL	18
WADL	18
Considerações Finais	20

UNIDADE 2 - REST COM JAVA, Recursos e comunicação sem estado	21
Conceitos REST	21
Cliente-Servidor	21
Stateless	22
Cache	22
Uniform Interfaces	22
URI	22
Serviço REST na prática	23
Postman	23
Consumindo um serviço REST muito simples	23
Implementando um serviço REST	24
REST com Java	24
Criando o projeto com Maven	25
Recursos	27
Tipos de dados	28
XML	28
XML Namespaces	30
Mapeamento de XML com JAXB	30
Definindo os modelos Livro e Autor	31
Considerações Finais	35
UNIDADE 3 - REST COM JAVA, Media Type e Uniform Interface	37
Tipo de dados: JSON	37
JSON no Java	38
Media Types	38
URIS mais descritivas	40
Endereçabilidade	40

O @Path do JAX-RS	41
Métodos HTTP e a Interface Uniforme	42
Códigos de Status	43
Série 2xx	43
Série 3xx	43
Série 4xx	44
Série 5xx	45
Lista completa de códigos	45
404 Resource Not Found	45
Statelessness	47
Implementando os demais métodos http no nosso serviço	47
Salvar um recurso com método POST	47
Atualizando um recurso com PUT	52
Excluindo um recurso com DELETE	54
Considerações Finais	55
UNIDADE 4 - REST COM JAVA, HATEOAS e JAX-RS Client	57
Segurança e Idempotência	57
HATEOAS	58
Implementado HATEOAS com JAX-RS	60
Cliente REST com JAX-RS	63
Implementação de client REST com JAX-RS	64
Interface fluente de JAX-RS Client API	67
Invocando um método POST	68
WADL	70
Considerações Finais	72
UNIDADE 5 - SEGURANÇA EM SERVIÇO REST	73

Dois Tipos de ataques	73
Como proteger-se destes ataques	74
Certificado auto assinado	75
Implantando SSL no servidor	77
Autenticação e Autorização	79
HTTP Basic	80
Consumindo serviço com autenticação	85
Consumindo serviço com SSL	86
Considerações Finais	87
Referências:	89

Apresentação

Olá! Sou Daniel Petrico e serei seu guia na disciplina de Tecnologia WebServices e RESTful com Java. Antes de começarmos esta jornada, gostaria de me apresentar.

Comecei minha carreira na área de tecnologia em 2008 como programador Java, antes mesmo de concluir minha graduação de Processamento de Dados. Desde então passei por algumas poucas empresas. Comecei em uma fábrica de software, onde desenvolvia e realizava manutenção em sistemas Java Web, usava tecnologias com Java Server Faces e Hibernate. Depois passei por uma empresa no ramo de telecomunicação fazendo integração de sistemas, aqui desenvolvi muitos Web Services com protocolo SOAP. Posteriormente fui trabalhar em uma multinacional do ramo de seguros, onde também desenvolvi Web Services com SOAP e muito RESTful.

A partir de 2018 dirigi meus estudos para área de Ciência de Dados e Big Data. Atualmente trabalho como Engenheiro de Dados em um projeto inovador e muito desafiador no ramo de agropecuária.

Tenho como valores principais a responsabilidade, comprometimento, ética e respeito.

Caso queira me acompanhar, me adicione nas redes sociais:

- Twitter: @danielpetrice
- LinkedIn: [linkedin.com/in/danielpetrice](https://www.linkedin.com/in/danielpetrice)
- Site: danielpetrice.com

UNIDADE 1 - PROGRAMAÇÃO NA WEB E SEUS COMPONENTES

Neste capítulo, você será apresentado aos conceitos básicos da Web. Mostrarei um pouco da concepção da Internet e como ela é usada para execução de sites e sistemas. Também falarei sobre os componentes que estão envolvidos neste cenário e alguns termos comuns que estão relacionados à programação Web.

Rápido crescimento da Internet

A Internet tornou-se um importante meio por onde as pessoas puderam trocar informações úteis de maneira eficiente. Com ela, foram quebradas barreiras significantes como a distância, já que um texto publicado nela pode ser lido por qualquer pessoa (com acesso à Internet) em qualquer lugar do mundo. Este movimento começou no início da década de 90, com adoção maior por parte de universidades e grupos de pesquisa, que a usavam para divulgar e compartilhar informações de suas descobertas com outros pesquisadores. Depois começaram a surgir algumas páginas pessoais, páginas institucionais de departamentos da universidade. Todas divulgando informações relevantes. O número de sites como estes foram crescendo até começar a chamar atenção de empresas que viram ali um potencial para divulgação de seus produtos e serviços. Depois disso foi inevitável o crescimento exponencial de sites, e conseqüentemente dados divulgados até os dias atuais.

Contudo, este rápido crescimento despertou a preocupação por parte dos desenvolvedores que estavam envolvidos na concepção dos padrões da Internet. Logo perceberam que isto poderia afetar de alguma maneira a capacidade da infraestrutura da rede. Vale lembrar que a Internet foi originada a partir da ARPANET, rede criada pelo departamento de defesa dos Estados Unidos durante a década de 60 para troca de documentos. O surgimento de sites e sistema surgiram muito depois, fazendo-se uso praticamente da mesma arquitetura.

Desta preocupação surgiram padrões e regras que visavam a manutenibilidade e escalabilidade de forma sustentável da rede. Durante este curso passaremos por alguns destes padrões, mas antes vamos ver como é o funcionamento básico da Internet, quais são seus componentes principais e como estão relacionados entre si.

Internet nossa de cada dia

Você já parou para pensar como a Internet impacta nossas vidas diariamente? Tenho certeza que ela facilita muito algumas tarefas cotidianas. Vamos a um exemplo. Imagine que você queira comprar um livro, mas pense que ainda não exista a Internet para ajudar. O que fazer? Uma opção seria pegar a famosa lista telefônica e sair ligando para as livrarias da cidade para verificar se alguma delas possui o livro desejado. Se tiver sorte, encontraria o livro em alguma delas, depois era só deslocar-se até uma delas e efetuar a compra. Nada mal.

Mas agora imagine realizando a mesma tarefa com a Internet, como ficaria? Você acha que conseguiria encontrar seu livro mais facilmente? Com certeza sim.

Este é um exemplo interessante que podemos explorar para entender como funciona os sistemas que rodam na Internet. O que está por trás desta simples tarefa cotidiana, de procurar e comprar um livro em alguma livraria virtual?

Para responder a esta pergunta, vamos examinar o caso de nosso usuário imaginário Tobias. Coincidentemente ele está querendo comprar um novo livro para sua coleção pessoal e decidiu usar a Internet para isso.

Em busca de seu livro, o primeiro passo realizado por Tobias é acessar o site da Amazon. Para isso ele abre seu navegador, escreve na barra de endereço “amazon.com” e pressiona a tecla “Enter”. O site se abre. Dentro do site, Tobias encontra o campo de pesquisa, digita o nome do livro que procura, “Harry Potter e a pedra secreta” e pressiona a tecla “Enter” novamente. Uma página mostrando todos os detalhes do livro é apresentada a ele. Agora está todo feliz da vida pois poderá comprar seu tão desejado livro. Parabéns Tobias!

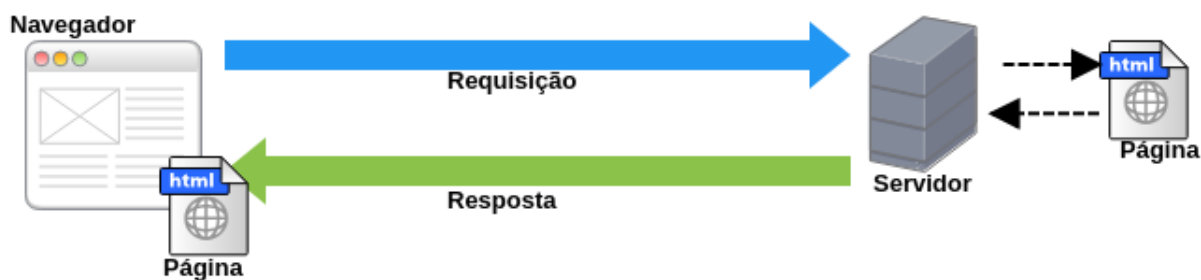
Mas espera aí, vamos analisar o que aconteceu no detalhe.

Requisição e Resposta

O que acontece quando nosso amigo tenta acessar o site, é que o navegador pega o endereço digitado por ele, e faz uma requisição para um servidor que está em algum lugar na Internet. Este servidor envia como resposta um arquivo escrito em HTML, que é uma linguagem que o navegador consegue interpretar e renderizar elementos gráficos que formam o layout do site.

Este processo de solicitar alguma coisa a um servidor é conhecido como Requisição e Resposta. Sempre neste cenário estão envolvidos basicamente dois atores: um cliente que faz a requisição, solicitando alguma coisa, e um servidor que irá fornecer o que foi solicitado, como mostra a figura 1.1 abaixo.

Figura 1.1 - Exemplo de Requisição e Resposta



Fonte: O autor

Mas como isso está acontecendo? Como a requisição chega até o servidor e como a resposta chega de volta ao cliente?

HTTP: Protocolo baseado em documentos

Para entender o que está acontecendo por detrás dos panos, vamos falar do HTTP. HTTP vem de “*HyperText Transfer Protocol*”, ou protocolo de transferência de hipertexto. Pode se dizer que ele é a base para o funcionamento de serviços que rodam pela Internet. Trata-se de um protocolo de comunicação baseado em documentos, já que por ele é possível enviar e receber dados estruturados de uma maneira entendível pelas partes envolvidas, normalmente um cliente e servidor.

Seu funcionamento pode ser descrito da seguinte maneira: um cliente coloca um documento dentro de um envelope e o envia para o servidor. O servidor retorna como resposta um documento, também dentro de um envelope, de volta para o cliente. Simples não?

O HTTP é o protocolo que define como deve ser a estrutura do envelope trocado entre as partes. Isto é fundamental para que haja entendimento da comunicação entre o cliente e o servidor.

Agora vamos voltar ao nosso exemplo de acesso à página da Amazon realizada pelo Tobias. Passaremos pelos detalhes da requisição realizada pelo cliente, ou o navegador utilizado pelo nosso amigo, até a resposta retornada pelo servidor.

HTTP Request

Para mostrar a comunicação realizada entre cliente e servidor, vamos fazer uso da ferramenta de desenvolvedor que é disponibilizada pela maioria dos navegadores. O acesso a esta ferramenta difere de navegador para navegador, você pode ver como acessá-la na documentação do navegador que estiver usando. Aqui vou utilizar o Google Chrome, acessando a ferramenta do desenvolvedor pressionando a tecla “F12” e clicando na aba “Network”.

Figura 1.2 - Request ao site amazon.com visto da ferramenta de desenvolvedor do Chrome



Fonte: O autor

A figura 1.2 mostra o resultado de uma requisição realizada para acessar o site amazon.com. Agora vamos analisar o que é cada parte apresentada por esta ferramenta, primeiramente passando pelos elementos da requisição e posteriormente analisando os da resposta.

Métodos do HTTP

Nesta figura podemos ver que na seção “General” a propriedade “Request Method” contém o valor “GET”. Esta propriedade indica qual o método que está sendo executado nesta operação. Ele serve para indicar como o cliente, no caso o navegador, está esperando que o servidor atenda à sua requisição.

No caso do método GET, é o método que o protocolo HTTP determina que um envelope seja enviado quando o cliente deseja obter alguma coisa do servidor.

Ao todo, o protocolo HTTP define 9 métodos, sendo o GET, POST, PUT, DELETE, OPTIONS, HEAD, TRACE, CONNECT e o PATCH. Cada um destes métodos possui aplicações bem definidas. Mais para frente vamos ver com mais detalhes os métodos mais utilizados em Web Services RESTful.

Caminho

A propriedade *“Request URL”* ainda na seção *“General”*, mostra o endereço que foi digitado na barra de endereços do navegador. Nela está definida o local que a requisição será enviada. Já *“Remote Address”* indica o local IP de onde o servidor está hospedado.

Além destas propriedades, vemos mais abaixo na seção *“Request Headers”*, a propriedade *“Path”*. Ela indica para o servidor o recurso que o cliente está buscando. Neste caso o valor da propriedade é *“/”*, indica que o cliente deseja buscar o que está na raiz publicada pelo servidor, ou normalmente a página inicial do site. Quando o cliente deseja buscar alguma outra coisa que não seja a página inicial do site, como uma página específica ou uma imagem, o conteúdo da propriedade *“Path”* conterá o local onde o servidor deverá buscar este recurso.

Estas propriedades funcionam da mesma maneira que um endereço colocado em um envelope que é enviado para alguém.

Cabeçalhos

A seção *“Request Headers”* contém o cabeçalho da requisição. Esta é dedicada para os metadados da mensagem enviada, ou seja, os dados referente aos dados que estão dentro do envelope. São representados por um conjunto de pares Chave-Valor, onde a chave é o nome de uma propriedade e o valor é o conteúdo desta propriedade. É como variáveis em uma linguagem de programação.

Podem haver vários pares nesta seção, alguns sendo definidos pelo próprio protocolo HTTP, veremos alguns dos principais mais a frente. As aplicações web podem definir seus próprios pares de Chave-Valor, formando um conjunto de propriedades entendidas somente por aquela aplicação e seus clientes.

No nosso exemplo, a requisição gerou 9 dados no cabeçalho. A tabela 1.1 mostra os atributos mais relevantes contidos no cabeçalho da requisição do nosso exemplo.

Tabela 1.1 - Headers da requisição

Propriedade	O que significa
method	Método do protocolo HTTP.
path	Caminho do recurso que sendo solicitado para o servidor.
accept	Sinaliza para o servidor qual o tipo de dados que o cliente está esperando. Neste caso o navegador deseja receber um arquivo text/html.
user-agent	Informações a respeito do cliente que está fazendo a requisição. Note que o cliente é o navegador

Fonte: O autor

Corpo da Requisição

Nesta seção vai o documento que é enviado dentro do envelope. Neste exemplo nenhum documento foi enviado, por isso esta seção foi omitida na visualização. Normalmente isso acontece em requisições do tipo GET, como a que estamos fazendo. Mas não se preocupe, mais a frente vamos ver casos onde dados são enviados ao servidor, por exemplo, em formatos XML ou JSON.

O importante por hora é ficar sabendo que uma requisição é dividida basicamente em método, cabeçalho e corpo.

HTTP Response

Trata-se basicamente do documento dentro do envelope que é retornado pelo servidor. Utilizando a mesma visualização da ferramenta do desenvolvedor do Google Chrome, podemos ver os detalhes desta resposta, conforme mostra a figura 1.3 abaixo.

Figura 1.3 - Response do site amazon.com visto da ferramenta de desenvolvedor do Chrome

▼ General

Request URL: https://www.amazon.com/
Request Method: GET
Status Code: 🟢 200
Remote Address: 72.246.131.124:443
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers

cache-control: no-cache
content-encoding: gzip
content-language: en-US
content-type: text/html; charset=UTF-8
date: Sat, 27 Jul 2019 16:53:46 GMT
expires: -1
pragma: no-cache
server: Server
set-cookie: session-id-time=20827872011; Domain=.amazon.com; Expires=Tue, 01-Jan-2036 08:00:01 GMT; Path=/
set-cookie: sp-cdn="L5Z9:BR"; Version=1; Domain=.amazon.com; Max-Age=518540775; Expires=Tue, 01-Jan-2036 08:00:01 GMT; Path=/
set-cookie: session-id=146-3615712-6505348; Domain=.amazon.com; Expires=Tue, 01-Jan-2036 08:00:01 GMT; Path=/
set-cookie: skin=noskin; path=/; domain=.amazon.com
set-cookie: i18n-prefs=USD; Domain=.amazon.com; Expires=Tue, 01-Jan-2036 08:00:01 GMT; Path=/
status: 200
strict-transport-security: max-age=47474747; includeSubDomains; preload
vary: Accept-Encoding,User-Agent,X-Amzn-CDN-Cache,X-Amzn-AX-Treatment
x-amz-rid: Y55GE6CZXKwMDMB9RNYS
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-ua-compatible: IE=edge
x-xss-protection: 1;

► Request Headers (9)

Fonte: O autor

A resposta é dividida basicamente em código da resposta, cabeçalho e corpo.

Código da resposta

Na figura 1.3 podemos ver a propriedade “*Status Code*”. Aqui vai ser retornado um código numérico que indica ao cliente o resultado do processamento realizado pelo servidor. No nosso exemplo, o valor retornado foi o 200, indicando que houve sucesso no processamento realizado pelo servidor.

O protocolo HTTP possui uma série de códigos específicos para cada situação, como erros, redirecionamento, entre outros. Não se preocupe, pois veremos mais sobre estes código mais a frente.

Cabeçalho

Da mesma maneira que o cliente envia no cabeçalho metadados para o servidor, o servidor também envia para o cliente. As informações contidas no cabeçalho de

resposta informam ao cliente a respeito do conteúdo retornado. Neste exemplo, ajudam o navegador mostrando como o navegador deve renderizar o conteúdo para o usuário.

Tabela 1.2 - Headers da resposta

Propriedade	O que significa
content-type	O tipo do conteúdo retornado. Neste caso o text/html ajuda o navegador a renderizar a página.
date	Data do processamento.
status	Código numérico indicando o status do processamento.
expires	Data indicando quando o conteúdo expira. Quando contém um valor inválido, como -1, indica que o conteúdo já está expirado.

Fonte: O autor

A tabela 1.2 acima mostra alguns dos valores mais relevantes contidos no cabeçalho de resposta.

Corpo da Resposta

Para visualizar o conteúdo da resposta, vamos selecionar a aba “Response” da ferramenta do desenvolvedor, conforme mostra a figura 1.4.

Nesta seção está contido o documento requisitado pelo cliente dentro do envelope. Neste exemplo, o servidor nos retorna uma página escrita em html, por isso podemos ver o código fonte da página. Aqui também poderia ter sido retornado um outro tipo de documento, como uma imagem, um documento em PDF, ou texto, ou planilha, vídeos, etc. Tudo poderia ser visto da mesma maneira, utilizando a ferramenta de desenvolvedor do navegador.

Figura 1.4 - Trecho do conteúdo retornado pelo servidor

```

: Headers Preview Response Cookies Timing
1 <!doctype html><html lang="en-us" class="a-no-js" data-19ax5a9jf="dingo"><!-- sp:feature:head-start -->
2 <head><script>var aPageStart = (new Date()).getTime();</script><meta charset="utf-8">
3 <script type="text/javascript">var ue_t0=ue_t0||new Date();</script><!-- sp:feature:cs-optimization -->
4 <meta http-equiv="x-dns-prefetch-control" content="on"><link rel="dns-prefetch" href="//images-na.ssl-images-amazon.com"><link rel="dns-prefe
5 window.ue_ihb = (window.ue_ihb || window.ueinit || 0) + 1;
6 if (window.ue_ihb === 1) {
7
8 var ue_csm = window,
9 ue_hob = +new Date();
10 (function(d){var e=d.ue=d.ue||{};f=Date.now||function(){return+new Date};e.d=function(b){return f()-(b?0:d.ue_t0)};e.stub=function(b,a){if(!b
11
12
13 var ue_err_chan = 'jserr-rw';
14 (function(d,e){function h(f,b){if(!(a.ec>a.mxe)&&f){a.ter.push(f);b=b||{};var c=f.logLevel||b.logLevel;c&&c!=='k&&c!=='m&&c!=='n&&c!=='p||a.ec++;(
15 pec:0,ts:0,erl:[],ter:[],mxe:50,startTimer:function(){a.ts++;setInterval(function(){d.ue&&a.pec<a.ec&&d.uex("at");a.pec=a.ec,1E4)});l.skipTr
16
17
18 var ue_id = 'Y55GE6CZXKWMDB9RNYS',
19 ue_url = '/gp/uedata',
20 ue_navtiming = 1,
21 ue_mid = 'ATVPDKIKX0DER',
22 ue_sid = '146-3615712-6505348',
23 ue_sn = 'www.amazon.com',
24 ue_furl = 'fls-na.amazon.com',
25 ue_surl = 'https://unagi-na.amazon.com/1/events/com.amazon.csm.nexusclient.prod',
26 ue_int = 0,
27 ue_fcsn = 1,
28 ue_urt = 3,
29 ue_rpl_ns = 'cel-rpl',
30 ue_ddq = 1,
31 ue_fpf = '//fls-na.amazon.com/1/batch/1/OP/ATVPDKIKX0DER:146-3615712-6505348:Y55GE6CZXKWMDB9RNYS$uedata=s:',
32 ue_rsc = 0,
33

```

Fonte: O autor

Podemos ver que o conteúdo é um texto em formato HTML, que é uma linguagem muito parecida com o XML, que pode ser interpretado pelo navegador. Agora, como o navegador sabe que ele precisa renderizar este conteúdo em HTML? Ele faz isso graças à informação “Content-type” contida no cabeçalho da resposta.

Arquitetura Cliente-Servidor

Até aqui foi mostrado um exemplo de como um site ou sistema funciona na Web. É importante que você tenha claro em sua mente este mecanismo. Você precisa saber que a arquitetura envolvida nestes tipos de sistema é o que comumente chamamos de Cliente-Servidor. A figura 1.5 exemplifica este conceito, onde você tem um cliente que requisita algo e um servidor que responde a esta requisição.

Figura 1.5 - Representação arquitetura Cliente-Servidor



Fonte: O autor

Este conhecimento irá te ajudar muito na hora de desenvolver suas soluções, Web Services em Rest, já que estes serviços são projetados desta maneira, obedecendo aos padrões da Internet. Neste modelo, o processamento pesado fica a cargo do servidor.

Termos que você deve saber

Antes de seguirmos para os assuntos relacionados especificamente para os serviços RESTful, ainda existem alguns termos que permeiam o contexto de programação para WEB que você deve saber. Então, para não se confundir nesta sopa de letrinhas, esta parte deste capítulo introdutório é dedicada para esclarecer alguns dos termos mais comumente utilizados.

HTTP

Este é o protocolo que todos sistemas da Web têm em comum. Por isso ele está aqui novamente, para que você tenha esta percepção de importância deste protocolo. Você verá que aplicações desenvolvidas utilizando RESTful aproveita bastante as definições trazidas por este protocolo. Por isso é fundamental que você tenha um sólido conhecimento sobre ele.

URI e URL

Frequentemente você se deparará com estes dois termos. Vamos ver o significado de cada um deles.

URI vem de “*Uniform Resource Identifier*”, ou Identificador Uniforme de Recurso em português. Este é o mecanismo utilizado para dar um nome único às coisas na Internet. Por exemplo, <https://www.amazon.com/index.html> é o nome único da página inicial do site da Amazon. Importante notar que ele identifica este recurso globalmente.

Já URL, vem de “*Uniform Resource Locator*”, ou Localizador Uniforme de Recurso. Ele é utilizado para indicar a localização de coisas na Internet. Por exemplo, <https://www.amazon.com> é um endereço onde está localizado o site da Amazon.

Note que, toda URL pode ser considerado um URI, já que ele identifica um recurso, ou um local onde esteja o recurso. Mas nem toda URI pode ser considerada uma URL. Isso porque podem ser utilizadas para definir “*namespaces*” em arquivos XML por exemplo, como demonstra a figura 1.6.

Figura 1.6 - Exemplo de Namespace


```

<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="https://www.w3schools.com/furniture">

  <h:table>
    <h:tr>
      <h:td>Apples</h:td>
      <h:td>Bananas</h:td>
    </h:tr>
  </h:table>

  <f:table>
    <f:name>African Coffee Table</f:name>
    <f:width>80</f:width>
    <f:length>120</f:length>
  </f:table>

</root>

```

Fonte: https://www.w3schools.com/xml/xml_namespaces.asp

Apesar de ser parecida com uma URL, a URI “https://www.w3schools.com/furniture” não representa um local onde se possa obter algum recurso.

Daqui para frente vamos utilizar apenas o termo URI para facilitar o entendimento.

XML

XML vem de “*eXtensible Markup Language*”, ou Linguagem de Marcação extensível em português. É um formato de arquivo estruturado de certa forma que dá a capacidade de armazenar e transportar dados. Ele foi projetado para ser facilmente entendível por humanos e máquinas. Em tempo, a figura 1.6 mostra um exemplo de arquivo XML.

JSON

JSON vem de “*JavaScript Object Notation*”, ou algo como Notação para objetos JavaScript em português. Também é um formato de arquivo estruturado que serve para armazenar e transportar dados. Seu formato é textual e é entendível tanto por humanos quanto por máquinas.

Figura 1.7 - Exemplo de JSON

```
{"employees":[  
  { "firstName":"John", "lastName":"Doe" },  
  { "firstName":"Anna", "lastName":"Smith" },  
  { "firstName":"Peter", "lastName":"Jones" }  
]}
```

Fonte: https://www.w3schools.com/js/js_json_xml.asp

A figura 1.7 mostra um exemplo de JSON. Nota-se que seu formato é mais limpo que o XML, por isso seu tamanho é menor que arquivos XML. Por causa disso, seu uso é preferido, principalmente em situações de troca de dados entre dispositivos móveis, onde há restrições com relação a largura de banda de redes móveis.

SOAP

Sigla para “*Simple Object Access Protocol*”, ou Protocolo Simples de Acesso a Objetos. Protocolo utilizado para desenvolvimento de Web Services. Também conhecido como envelope SOAP. O desenvolvimento de Web Services utilizando este protocolo segue uma estrutura mais rígida baseado em XML e envelope.

O modelo de desenvolvimento em SOAP é muito diferente do encontrado em REST. Por vezes tentam comparar Web Services desenvolvidos em REST com os desenvolvidos com SOAP, mas esta comparação não faz sentido, uma vez que SOAP é um protocolo enquanto REST é um estilo de arquitetura, ou seja, coisas completamente diferente entre si.

Tome cuidado para não cair em armadilhas como essas, a de comparar uma coisa com outra para verificar qual é a melhor! Tanto um quanto o outro possuem vantagens e desvantagens.

Web Services desenvolvidos com SOAP são mais encontrados em empresas corporativas, onde há uma necessidade maior de se ter uma burocracia e contratos mais bem definidos entre clientes e serviços. Nestes casos, SOAP atende bem a estas características.

A título de curiosidade, a secretaria da fazenda brasileira utiliza serviços SOAP como solução para a Nota Fiscal Eletrônica (NFE) do país.

WSDL

“*Web Service Description Language*”, ou Linguagem de Descrição de Serviços Web, é um arquivo baseado em XML onde se faz toda definição de serviços SOAP. Nele são definidas coisas como os tipos de dados que serão trocados entre cliente e serviço, quais operações serão disponibilizadas, dentre outras coisas. A figura 1.8 mostra a estrutura principal de um WSDL.

Figura 1.8 - Estrutura de um WSDL

```
<definitions>

<types>
  data type definitions.....
</types>

<message>
  definition of the data being communicated....
</message>

<portType>
  set of operations.....
</portType>

<binding>
  protocol and data format specification....
</binding>

</definitions>
```

Fonte: https://www.w3schools.com/xml/xml_wsdl.asp

Com o WSDL o desenvolvedor pode criar um cliente utilizando qualquer linguagem de programação para que seu sistema seja capaz de se comunicar com o Web Service.

WADL

“*Web Application Description Language*”, ou Linguagem de Descrição de Aplicativos Web. Arquivo XML que descreve um Web Service RESTful. A figura 1.9 traz um exemplo de WADL.

Figura 1.9 - Exemplo de WADL

```

<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
  xmlns:tns="urn:yahoo:yn"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:yn="urn:yahoo:yn"
  xmlns:ya="urn:yahoo:api"
  xmlns="http://wadl.dev.java.net/2009/02">
  <grammars>
    <include
      href="NewsSearchResponse.xsd"/>
    <include
      href="Error.xsd"/>
  </grammars>

  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
      <method name="GET" id="search">
        <request>
          <param name="appid" type="xsd:string"
            style="query" required="true"/>
          <param name="query" type="xsd:string"
            style="query" required="true"/>
          <param name="type" style="query" default="all">
            <option value="all"/>
            <option value="any"/>
            <option value="phrase"/>
          </param>
          <param name="results" style="query" type="xsd:int" default="10"/>
          <param name="start" style="query" type="xsd:int" default="1"/>
          <param name="sort" style="query" default="rank">
            <option value="rank"/>
            <option value="date"/>
          </param>
          <param name="language" style="query" type="xsd:string"/>
        </request>
        <response status="200">
          <representation mediaType="application/xml"
            element="yn:ResultSet"/>
        </response>
        <response status="400">
          <representation mediaType="application/xml"
            element="ya:Error"/>
        </response>
      </method>
    </resource>
  </resources>
</application>

```

Fonte: <https://www.w3.org/Submission/wadl/>

Este arquivo tem o mesmo propósito de um WSDL, onde o desenvolvedor pode gerar um cliente para se comunicar com o serviço utilizando qualquer linguagem de programação. Contudo sua utilização não é obrigatória em serviços RESTful, dada sua natureza mais livre e menos burocrática.

Considerações Finais

Neste capítulo tivemos a oportunidade de ver, de modo geral, como é o funcionamento de sistemas que rodam na Internet. É importante que você tenha em mente estes conceitos, pois são base para o estudo de Web Services RESTful.

Vimos que tais sistemas funcionam em arquitetura do tipo Cliente-Servidor, onde um cliente, como um navegador Web, realiza uma requisição a um servidor, que então responde de volta com o conteúdo solicitado.

Este modelo de requisição e resposta segue um padrão que é definido pelo protocolo HTTP. Talvez este seja o elemento mais importante de sistemas voltados para Internet, pois se trata do protocolo que dita as regras de como a comunicação deve ser realizada entre cliente e servidor.

Por fim você foi apresentado a alguns termos que permeiam o contexto de programação Web, como XML, JSON, SOAP, WSDL e WADL. Alguns destes termos não farão parte dos nossos estudos, mas é importante que você possa os distinguir para evitar confusões.

UNIDADE 2 - REST COM JAVA, Recursos e comunicação sem estado

Neste capítulo, veremos dois princípios do estilo de arquitetura REST: o primeiro é que todas as coisas devem ter um identificador único e estes são chamados de recursos, o segundo é a comunicação sem estado, ou stateless.

Em REST, busca-se tirar o maior proveito de toda infraestrutura já consolidada da Web, fazendo bom uso do protocolo HTTP e seus métodos. Veremos como obter isso na prática, começando a desenvolver nosso próprio Web Service REST utilizando o JAX-RS, que determina como deve ser desenvolvido este tipo de programas em Java.

Conceitos REST

REST é acrônimo para “*REpresentational State Transfer*”, ou Transferência de Estado Representativo em português. É um estilo de desenvolvimento de softwares para WEB. O termo foi cunhado por Roy Fielding em sua tese de doutorado, onde ele afirma que REST “ênfatiza a escalabilidade das interações do componente, a generalidade das interfaces, a implementação independente de componentes, com componentes intermediários para diminuir a latência, garantindo segurança e encapsulando sistemas legados” (FIELDING, Roy - 2000).

Este estilo lista uma série de princípios que determinam uma maneira estruturada e sustentável para desenvolvimento de softwares para a chamada Web moderna. Utilizando REST, você tira proveito de toda infraestrutura já bem definida da Internet, fazendo melhor uso das especificações do protocolo HTTP, URI e métodos condizentes com cada operação que se deseja realizar.

Cliente-Servidor

A primeira coisa que você deve saber é que REST utiliza a arquitetura Cliente-Servidor demonstrada no capítulo anterior. Esta arquitetura provê uma boa separação de responsabilidade, deixando questões como processamento pesado e armazenamento de dados para ser realizado por um servidor, enquanto o cliente se preocupa com questões como exibir dados em uma interface gráfica. Como estamos falando em Web Services, normalmente clientes são navegadores para páginas Web ou então aplicativos de Smartphone. E esta é uma característica importante desta arquitetura, pois proporciona uma grande escalabilidade.

Stateless

O próximo princípio trazido por REST é o de não armazenar estado das operações no lado servidor, o chamado “*Stateless*”. Desta maneira, cada requisição feita pelo cliente deve conter toda informação necessária para que o servidor possa processá-la sem precisar consultar dados adicionais.

Isso traz a estes serviços melhorias em quesitos como confiabilidade e escalabilidade. Confiabilidade pois fica fácil de se recuperar de falhas no serviço e escalabilidade pois o servidor não precisa se preocupar em montar toda uma estrutura para ficar armazenando dados de diferentes sessões em memória ou banco de dados. Uma desvantagem aqui é que cada requisição precisa carregar uma quantidade de dados maior.

Cache

A estrutura da web, com as definições do protocolo HTTP, providenciam um mecanismo nativo de cacheamento para estes serviços. O cache é uma funcionalidade onde há reaproveitamento de dados já consultados anteriormente pelo cliente.

Desta maneira, um dado que seja marcado como sendo de cache só é enviado para o cliente se ele estiver desatualizado. Isto diminui o número de interações entre cliente e servidor melhorando performance e eficiência.

Uniform Interfaces

Este princípio determina que as interações entre cliente e serviço devem ser realizadas através do uso adequado dos métodos do HTTP.

URI

Em REST tudo deve conter uma identificação, um ID. Como vimos no capítulo anterior, o mecanismo de identificação utilizado em sistemas Web é o URI. Portanto todas as coisas, ou recursos que serão disponibilizados, deverão ser alcançados através de uma URI.

Serviço REST na prática

Para melhorar o entendimento destes conceitos e irmos introduzindo a outros essenciais trazidos por este estilo de arquitetura, iremos analisar um Web Service REST muito simples, que irá nos dar a oportunidade de ver na prática o que foi descrito até aqui.

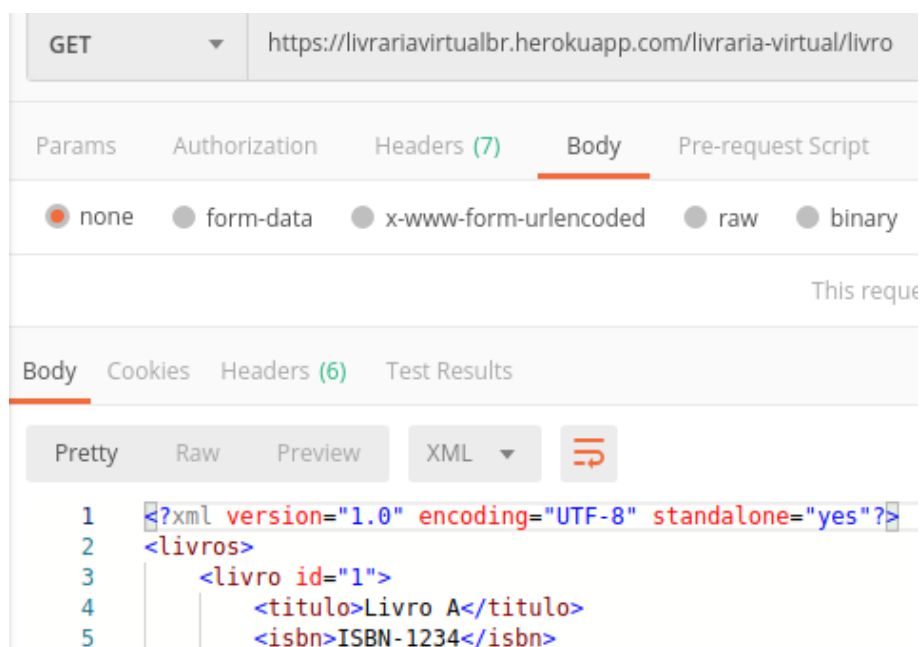
Postman

Para seguir com o exemplo iremos utilizar a ferramenta para interação com Web Services chamada Postman. Você poderá obtê-la em seu site oficial <https://getpostman.com>. Baixe e instale este programa em seu computador, vamos utilizar esta ferramenta daqui para frente pois ela facilita a visualização dos elementos da requisição e respostas dos serviços.

Consumindo um serviço REST muito simples

Através da ferramenta Postman, digite <https://livrariavirtualbr.herokuapp.com/livraria-virtual/livro> na barra de endereços, escolha o método GET e pressione Enter.

Figura 2.1 - Serviço no Postman



Fonte: O autor

Na figura 2.1 acima podemos ver os conceitos apresentados até aqui. O Postman é o cliente do serviço enquanto o host Heroku é o servidor, logo temos uma arquitetura Cliente-Servidor. O serviço é Stateless, pois enviamos as informações necessárias

para execução correta do serviço. O endereço digitado é a URI que identifica o recurso buscado. Além disso utilizamos o método GET do protocolo HTTP.

Implementando um serviço REST

A partir de agora podemos começar a implementar nossos próprios serviços REST, para ir aprimorando aos poucos nossos conhecimentos. Como objeto de estudos, vamos desenvolver um Web Service para uma livraria virtual fictícia. Ao final, teremos todos serviços necessários para:

- Listar livros disponíveis;
- Obter um livro específico;
- Cadastrar e atualizar livros;
- Deletar um livro específico;

Se você já está acostumado com desenvolvimento de softwares em geral, já deve ter percebido que a listagem acima refere-se ao mesmo que fazer um CRUD, ou Criação, Recuperação, Atualização e Delete de registros em banco dados.

REST com Java

A implementação de Web Services REST com Java se dá seguindo a especificação JSR 311 (<https://jcp.org/en/jsr/detail?id=311>), também conhecida como JAX-RS (*Java API for RESTful Web Services*). Para seguir no desenvolvimento de nossa aplicação para livraria virtual, vamos utilizar o Jersey, que é a implementação de referência do JAX-RS. O Jersey pode ser baixado em <https://jersey.java.net/download.html> e sua documentação está disponível em <https://jersey.github.io>.

Além do Jersey, vamos utilizar como IDE o Eclipse e framework Maven para gerenciar as dependências de bibliotecas. Baixe e instale a última versão do Eclipse em seu computador através do site <https://www.eclipse.org/>, ela já vem com o Maven pronto para usar.

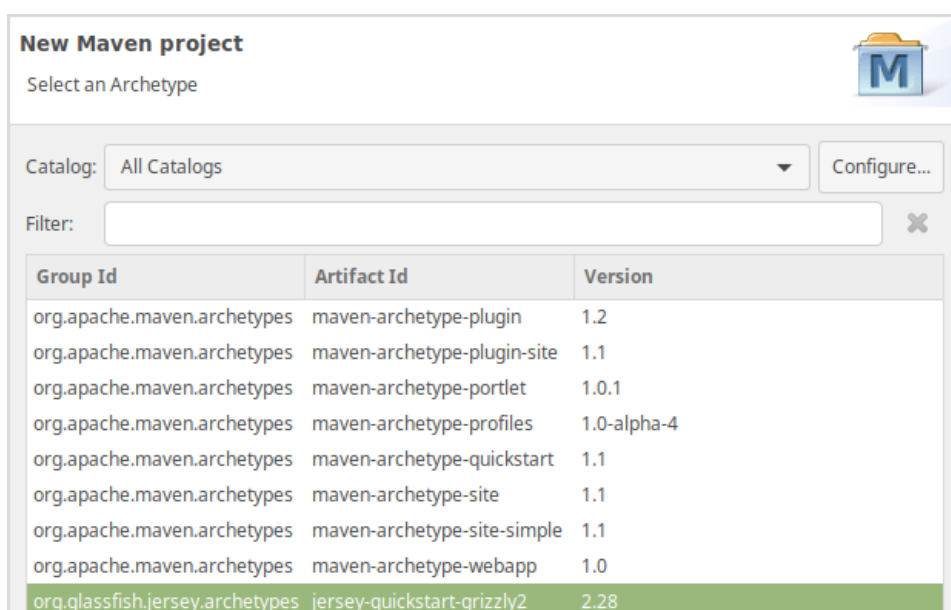
Aqui vou mostrar o passo a passo de como você pode criar um Web Service REST do zero. Se preferir, poderá baixar o código-fonte e importar no seu workspace do Eclipse no repositório do github <https://github.com/danielpetrico/livraria-virtual>.

Criando o projeto com Maven

Vamos criar o projeto no Eclipse utilizando o Maven, seguindo a recomendação do Jersey e utilizar um archetype que já traz as dependências necessárias (<https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/getting-started.html#new-from-archetype>).

No Eclipse, clique em *File, New e Maven Project*. Na tela que se abrir desmarque a caixa *Use default Workspace location* e insira no campo *Location* o local onde você deseja que seu projeto seja salvo. Clique em *Next*. A tela da figura 2.2 será exibida.

Figura 2.2 - Novo projeto Maven no Eclipse



Fonte: O autor

Clique em *Add Archetype* e preencha os campos com os valores da tabela 2.1:

Tabela 2.1 - Dados do archetype do Jersey

Group ID	org.glassfish.jersey.archetypes
Artifact ID	jersey-heroku-webapp
Version	2.28
Repository URL	Deixe em branco

Fonte: O autor

Clique em *OK*. Você será redirecionado à tela anterior. Certifique-se de que o Archetype que você acabou de inserir esteja selecionado e clique em *Next*.

Na próxima tela preencha os dados referentes a seu projeto. A tabela 2.2 mostra um exemplo de como pode ser preenchidos.

Tabela 2.2 - Dados do seu projeto

Group ID	br.com.uniciv.rest
Artifact ID	livraria-virtual
Version	0.0.1-SNAPSHOT
Package	br.com.uniciv.livraria

Fonte: O autor

Clique em *Finish* e aguarde que o projeto seja inserido no Workspace do Eclipse e suas dependências sejam baixadas.

O projeto recém-criado obedece a estrutura de pastas de um típico projeto Maven. Contém um arquivo pom.xml para gerenciar as dependências e também foram criadas 2 classes, Main.java e MyResource.java. A primeira classe inicializa a aplicação web no servlet contêiner Jetty, que vem embarcado neste archetype. Esta classe também configura a aplicação JAX-RS no contêiner. Já a segunda classe é um exemplo básico de um recurso REST.

Abra a classe Main.java e altere o texto contido no parâmetro do método `root.setContextPath("/")` para `root.setContextPath("/livraria-virtual")`. Este método determina a URI base da nossa aplicação. A figura 2.3 mostra como fica a classe depois da alteração.

Agora execute esta classe, clicando com o botão direito dentro da classe, depois em *Run as e Java Application*.

No navegador acesse <http://localhost:8080/livraria-virtual/myresource>.

Uma página com o texto “*Hello, Heroku!*” deverá ser exibida e indica que as configurações realizadas funcionaram. Caso não tenha aparecido este texto, revise os passos executados.

Mas de onde está vindo este texto?

Figura 2.3 - Classe Main.java

```
public static void main(String[] args) throws Exception{
    // The port that we should run on can be set into an environment variable
    // Look for that variable and default to 8080 if it isn't there.
    String webPort = System.getenv("PORT");
    if (webPort == null || webPort.isEmpty()) {
        webPort = "8080";
    }

    final Server server = new Server(Integer.valueOf(webPort));
    final WebApplicationContext root = new WebApplicationContext();

    root.setContextPath("/livraria-virtual");
    // Parent loader priority is a class loader setting that Jetty accepts.
    // By default Jetty will behave like most web containers in that it will
    // allow your application to replace non-server libraries that are part of the
    // container. Setting parent loader priority to true changes this behavior.
    // Read more here: http://wiki.eclipse.org/Jetty/Reference/Jetty\_Classloading
    root.setParentLoaderPriority(true);

    final String webappDirLocation = "src/main/webapp/";
    root.setDescriptor(webappDirLocation + "/WEB-INF/web.xml");
    root.setResourceBase(webappDirLocation);

    server.setHandler(root);

    server.start();
    server.join();
}
```

Fonte: O autor

Volte ao projeto no Eclipse e abra a classe MyResource.java. Veja que o método `getIt()` é o responsável por retornar este texto.

Agora vamos modificar este serviço e ir entendendo mais sobre os princípios REST.

Recursos

Segundo o próprio Roy Fielding escreve, “A abstração chave de uma informação no REST é um recurso. Qualquer informação que possa ser nomeada pode ser um recurso: um documento ou imagem, etc” (FIELDING, Roy, 2000). Desta maneira, sempre que formos modelar nossos serviços iremos pensar em recursos.

Por exemplo, a classe MyResource.java disponibiliza um recurso através da URI `http://localhost:8080/livraria-virtual/myresource`. A representação do recurso contida ali é realizada através de um texto, o “*Hello, Heroku!*”.

Pode se dizer, que recursos sejam consideradas entidades do sistema. No caso de nossa aplicação Livraria Virtual, um livro pode ser um recurso. Perceba que logo mais

vamos criar serviços para recuperar uma lista de livros, inserir, atualizar e deletar livros. Ou seja, livro é uma entidade e portanto um recurso para o REST.

Sendo assim, vamos criar nosso primeiro recurso utilizando a URI `http://localhost:8080/livraria-virtual/livro`.

No eclipse, crie a classe `LivroResource.java` conforme a figura 2.4.

Figura 2.4 - Classe que mapeia o recurso livro

```
1 package br.com.uniciv.livraria;
2
3 import javax.ws.rs.Path;
4
5 @Path("livro")
6 public class LivroResource {
7
8 }
```

Fonte: O autor

Note que acima da definição da classe colocamos a anotação `@Path`, que indica o caminho do nosso recurso. Ela irá formar a URI, por onde os clientes poderão interagir com este recurso.

Agora precisamos resolver uma questão importante: como será a representação do nosso recurso livro? Em outras palavras, como os dados dos livros serão disponibilizados para os clientes do nosso serviço?

Tipos de dados

Quando falamos em Web Services, estamos falando de sistemas que interagem entre si de maneira agnóstica, onde a linguagem de programação utilizada tanto por parte do cliente quanto por parte do servidor, não tenha interferência. Para isso, dois formatos de arquivos são amplamente utilizados por sua capacidade de conseguir representar e transportar dados de maneira eficiente, o XML e o JSON.

XML

XML ou *eXtensible Markup Language*, Linguagem de Marcação Extensível em português, é uma linguagem de marcação muito parecida com o HTML. Ela foi projetada para armazenar e transportar dados. Por ser auto-descritivo, tem a capacidade de ser entendido tanto por máquinas quanto por pessoas.

O XML não é uma linguagem de programação, você não conseguirá fazer nada com ela, apenas descrever algo. A figura 2.5 apresenta um exemplo de XML.

Figura 2.5 - Exemplo de XML

```
<livro id="0001">
  <titulo>Harry Potter</titulo>
  <autor>J K. Rowling</autor>
  <ano>2005</ano>
  <preco>59.99</preco>
</livro>
```

Fonte: O autor

Note que os dados estão estruturados de maneira que se pode compreender sua informação facilmente. Não há necessidade de se utilizar tags pré-definidas, as tags do exemplo acima foram inventadas.

Ele é considerado extensível, pois mesmo se adicionar ou remover dados da sua estrutura as aplicações que fazem uso dos dados continuarão funcionando normalmente.

A estrutura do XML é muito simples, contendo apenas um elemento raiz e filhos aninhados:

```
<raiz>
  <filho>
    <filho>.....</filho>
  </filho>
</raiz>
```

Além disso, um XML pode conter partes que servem para informar como o dado contido nele deve ser processado. No começo do arquivo pode conter a linha como abaixo:

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

Esta linha é chamada de prólogo e não faz parte do dado que está sendo transportado, mas traz informações como a versão do xml que está sendo usado e a codificação de caracteres que deve ser utilizado no momento de ler os dados.

O XML é formado por elementos. Um elemento compreende uma tag completa, tag inicial, o conteúdo contido dentro da tag e a tag final. Um elemento pode conter textos, atributos e outros elementos.

Um elemento pode conter atributos específicos para ele. No exemplo abaixo, “id” é um atributo do elemento livro:

```
<livro id="00001">
```

XML Namespaces

Da mesma maneira como em Java as classes são separadas por pacotes para evitar conflito de nomes, no XML é possível utilizar os Namespaces para evitar conflito de nomes entre elementos. Um elemento pode ser livremente definido pelo desenvolvedor. Em um caso onde uma aplicação necessite fazer a junção de dois XML's diferentes, um conflito de nomes pode ocorrer caso haja elementos com mesmo nome em cada um dos arquivos. A figura 2.6 mostra como foi feita a separação entre dois elementos com o nome *table*, onde um table significa tabela do arquivo HTML enquanto outro table significa mesa.

Figura 2.6 - Exemplo de Namespace

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

Fonte: https://www.w3schools.com/xml/xml_namespaces.asp

Mapeamento de XML com JAXB

O Java provê uma API que automatiza o mapeamento entre documentos XML em objetos Java. O *Java Architecture for XML Binding*, ou JAXB tem métodos para converter arquivos XML para Java e vice-versa. Seu site oficial,

<https://github.com/javaee/jaxb-v2>, possui uma ótima documentação e deve ser consultada sempre que estiver em dúvida sobre sua funcionalidade.

Definindo os modelos Livro e Autor

Agora que já sabemos como é a estrutura do XML e qual biblioteca a ser utilizada no Java para manipular documentos deste tipo, podemos seguir com nosso exemplo e fazer a nossa primeira operação, a de retornar uma lista de livros.

Para representar os livros, podemos pensar em um modelo onde teremos duas classes, Livro e Autor. Em orientação à objetos, temos esta composição, onde um Livro pode ser escrito por um ou mais autores. Por isso vamos criar as duas classes e relacioná-las conforme mostra as figuras 2.7 e 2.8.

Figura 2.7 - Classe Livro

```
public class Livro {  
  
    private Long id;  
    private String titulo;  
    private String isbn;  
    private String genero;  
    private Double preco;  
    private List<Autor> autor;  
  
    // Getters and Setters omitidos  
}
```

Fonte: O autor

Figura 2.8 - Classe Autor

```
public class Autor {  
  
    private Long id;  
    private String nome;  
  
    // Getters and Setters omitidos  
}
```

Fonte: O autor

Com as duas classes criadas, vamos adicionar as anotações do JAXB para que estas classes possam ser convertidas em XML em nosso serviço.

Figura 2.9 - Classe Livro com anotações JAXB


```

@XmlRootElement(name = "livro")
@XmlAccessorType(XmlAccessType.FIELD)
public class Livro {

    @XmlAttribute
    private Long id;
    @XmlElement
    private String titulo;
    @XmlElement
    private String isbn;
    @XmlElement
    private String genero;
    @XmlElement
    private Double preco;
    @XmlElement
    private List<Autor> autor = new ArrayList<Autor>();

    // Getters and Setters omitidos
}

```

Fonte: O autor

Vamos analisar o que significa cada uma destas anotações.

@XmlElement, essa anotação indica que a classe será o elemento raiz do XML que será gerado a partir dela. A propriedade *name* é usada para determinar o nome da tag do elemento.

@XmlElement, define como os campos serão serializados pelo JAXB. Os valores para esta anotação podem ser:

PUBLIC_MEMBER: Serializa atributos públicos.

PROPERTY: Serializa atributos que tem os métodos getters anotados.

FIELD: Serializa os atributos anotados.

NONE: Serializa atributos e métodos getters anotados.

@XmlAttribute, indica que o atributo da classe será transformado em atributo do elemento “livro” do xml.

@XmlElement, indica que o atributo da classe será transformado em elemento no XML.

Repita o procedimento anotando a classe Autor conforme a figura 2.10.

Figura 2.10 - Classe Autor com anotações JAXB

```

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Autor {

    @XmlAttribute
    private Long id;
    @XmlElement
    private String nome;

    // Getters and Setters omitidos

}

```

Fonte: O autor

Agora precisamos fazer uma alteração na nossa classe Livro Resource, criando um método para responder a uma requisição do tipo HTTP GET. Este método irá retornar uma lista de livros. Para nos ajudar, vamos criar também uma lista que servirá como repositório de livros. Ela contém uma lista fixa de livro. Em um sistema real, esta classe iria buscar os livros em um banco de dados. Os códigos podem ser vistos nas figuras 2.11 e 2.12.

Figura 2.11 - Classe LivroRepositorio que retorna uma lista de livros

```

public class LivroRepositorio {

    private Map<Long, Livro> livros = new HashMap<>();

    public LivroRepositorio() {
        Livro livro1 = new Livro(
            1L, "Livro A", "ISBN-1234", "Genero A", 23.99, "Autor 1");
        Livro livro2 = new Livro(
            2L, "Livro B", "ISBN-4321", "Genero B", 19.99, "Autor 2");

        livros.put(livro1.getId(), livro1);
        livros.put(livro2.getId(), livro2);
    }

    public List<Livro> getLivros() {
        return new ArrayList<>(livros.values());
    }

}

```

Fonte: O autor

Figura 2.12 - Classe LivroResource com método GET

```

@Path("/livro")
public class LivroResource {

    private LivroRepositorio livroRepo = new LivroRepositorio();

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public List<Livro> getLivros() {
        return livroRepo.getLivros();
    }
}

```

Fonte: O autor

Agora basta parar a aplicação e executá-la novamente pelo método main. Depois acesse o browser e acesse <http://localhost:8080/livraria-virtual/livro>. Você deve obter um resultado como o mostrado na figura 2.13.

Figura 2.13 - Resultado do GET livros

```

<livroes>
  <livro id="1">
    <titulo>Livro A</titulo>
    <isbn>ISBN-1234</isbn>
    <genero>Genero A</genero>
    <preco>23.99</preco>
    <autor id="5">
      <nome>Autor 1</nome>
    </autor>
  </livro>
  <livro id="7">
    <titulo>Livro B</titulo>
    <isbn>ISBN-4321</isbn>
    <genero>Genero B</genero>
    <preco>19.99</preco>
    <autor id="8">
      <nome>Autor 2</nome>
    </autor>
  </livro>
</livroes>

```

Fonte: O autor

Perceba que a lista de livros que o método `getLivros()` retornou foi transformada em uma estrutura de XML. É possível ver que há dois elementos livros aninhados em um elemento raiz chamado “livroes”. Este nome meio estranho é resultado do sufixo “es” adicionado ao final do nome do elemento que forma a lista. Isto é um comportamento padrão quando se está trabalhando com listas.

Para alterar o nome do elemento raiz do nosso XML para apresentar a tag livros ao invés de “livroes”, iremos criar mais uma classe que servirá como elemento raiz, e dentro desta classe vamos colocar uma lista de livros. Além disso vamos alterar o método `getLivros` da classe `LivroResource` para retornar a nova classe criada. Desta forma, quando o objeto Java for transformado em XML, irá seguir a estrutura de objetos definida por nós, com nomes mais representativos.

Crie uma classe Livros.java como mostra a figura 2.14

Figura 2.14 - Classe Wrapper para Livros

```
@XmlElement(name="livros")
@XmlAccessorType(XmlAccessType.FIELD)
public class Livros {

    @XmlElement(name="livro")
    private List<Livro> livros = new ArrayList<Livro>();

    public List<Livro> getLivros() {
        return livros;
    }

    public void setLivros(List<Livro> livros) {
        this.livros = livros;
    }

}
```

Fonte: O autor

Agora altere a classe LivroResource.java para retornar uma instância da classe Livros.java no método getLivros().

Figura 2.15 - Classe LivroResource

```
private LivroRepositorio livroRepo = new LivroRepositorio();

@GET
@Produces(MediaType.APPLICATION_XML)
public Livros getLivros() {
    Livros livros = new Livros();
    livros.setLivros(livroRepo.getLivros());
    return livros;
}
```

Fonte: O autor

Agora é só reiniciar a aplicação e testar novamente, agora o xml retornado possui como elemento raiz a tag livros.

Considerações Finais

Este capítulo mostrou alguns princípios do estilo de arquitetura de sistemas para Web, que mostram como é vantajoso desenvolver Web Services tomando o maior proveito de toda infraestrutura já bem consolidados da Internet, como seus protocolo HTTP e a URI que são componentes principais do REST.

Vimos que para desenvolver Web Services REST com Java precisamos seguir a especificação JSR 311, mais conhecida como JAX-RS. Já a implementação de referência desta especificação é o Jersey, que provê uma biblioteca completa para criar aplicações REST com Java.

Tendo o conhecimento dos fundamentos de REST e de como deve ser implementado em Java, iniciamos o desenvolvimento de nosso próprio Web Service, utilizando como exemplo uma livraria-virtual fictícia.

Através deste exemplo podemos ver na prática como é o funcionamento deste estilo de programação. Com ele tivemos a oportunidade de ver como disponibilizar dados em estrutura de XML para que um cliente possa utilizá-lo. A partir de agora vamos aprofundar ainda mais nestes exemplos.

UNIDADE 3 - REST COM JAVA, Medita Type e Uniform Interface

Neste capítulo iremos abordar mais alguns princípios de REST. Veremos como um recurso pode ter mais de uma representação (XML, JSON) utilizando *Media Types*. Utilizar URI's descritivas para identificar recursos específicos. Salvar recurso sem manter estado no servidor. Utilizar códigos de respostas corretamente. Por fim vamos implementar a atualização e exclusão de recursos para completar o CRUD.

Tipo de dados: JSON

“*JavaScript Object Notation*”, ou simplesmente JSON, é um sintaxe textual baseado em JavaScript para armazenar e transportar dados, assim como o XML.

Por ser textual, tem a capacidade de ser compreendido tanto por humanos quanto por máquinas. É agnóstico à linguagem de programação, podendo ser convertido para objetos e vice-versa através de bibliotecas específicas.

Os dados no JSON são organizados através de pares chave/valor, sendo chave o nome de atributo e valor o conteúdo. Cada atributo é separado por vírgulas. Chaves delimitam quando começa e termina um objeto e colchetes é utilizado para arrays.

Um exemplo muito básico de JSON é: { "nome":"João da Silva" }

Note que tanto a chave quanto o valor são envoltos por aspas.

O JSON tem algumas vantagens em relação ao XML, como o fato de ter uma sintaxe mais simples e com menos boilerplate. Os exemplos abaixo mostram o mesmo dado sendo representado por XML e JSON.

Tabela 3.1 - Comparação XML e JSON

<pre>{ "pessoas": [{ "nome": "João", "idade": "18" }, { "nome": "Maria", "idade": "25" }, { "nome": "Pedro", "idade": "31" }]}</pre>	<pre><pessoas> < Pessoa> < nome>João</ nome> < idade>18</ idade> </ Pessoa> < Pessoa> < nome>Maria</ nome> < idade>25</ idade> </ Pessoa> < Pessoa> < nome>Pedro</ nome> < idade>31</ idade> </ Pessoa> </ pessoas></pre>
--	---

Fonte: O autor

Por ser mais simples, o tamanho (em bytes) do JSON é menor que o do XML, o que favorece aplicações que tenham restrição de largura de banda de Internet, como APP's para Smartphones.

JSON no Java

A implementação do nosso Web Service utilizando o Jersey já nos dá suporte para utilizarmos tipos de dados JSON. Para isso vamos utilizar a biblioteca “jersey-media-json-binding”, que vem comentada no arquivo pom.xml.

A primeira coisa que vamos fazer é descomentar essas linhas, conforme figura 3.1.

Figura 3.1 - Dependência para JSON no pom.xml

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-binding</artifactId>
</dependency>
```

Fonte: O autor

Agora basta alterar o método `getLivros()` da classe `LivroResource` adicionando mais uma opção na anotação `@Produces`. Esta anotação recebe um array que diz quais tipos de dados aquele método pode retornar. Como queremos que ele passe a retornar JSON e XML, vamos adicionar `MediaType.APPLICATION_JSON`.

Figura 3.2 - Método retornando XML e JSON

```
@GET
@Produces({MediaType.APPLICATION_XML,
          MediaType.APPLICATION_JSON})
public Livros getLivros() {
    Livros livros = new Livros();
    livros.setLivros(livroRepo.getLivros());
    return livros;
}
```

Fonte: O autor

Agora este método está pronto para retornar XML e JSON. Mas como o serviço sabe qual tipo de dado deverá retornar em cada momento? Para entender isso vamos ver o mecanismo de `MediaType` do protocolo HTTP.

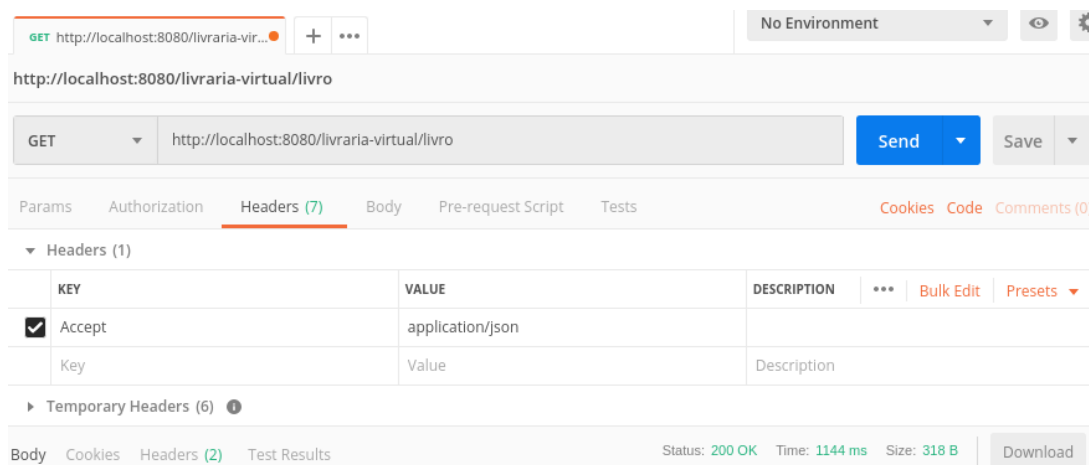
Media Types

Através do mecanismo de Media Types, um cliente pode negociar com o servidor qual tipo de dado ele prefere receber como resposta. Os tipos mais comuns quando se trata de serviços REST são o XML e JSON, mas o protocolo HTTP permite a utilização

de tipos texto, imagem, vídeos, entre outros. Tudo isso para representar melhor o recurso.

A maneira como um cliente negocia o tipo de dados com o servidor é através de informações passadas no cabeçalho da requisição através da propriedade *Accept*. Para testar isso vamos voltar a utilizar o *Postman*. Como ele podemos mudar o cabeçalho conforme necessitamos.

Figura 3.3 - Header Accept no Postman



Fonte: O autor

Note que adicionamos a *key Accept* com o valor *application/json* na seção *Headers* do Postman. Aqui informamos que queremos receber como resposta um JSON.

Figura 3.4 - Resultado JSON retornado

```
{
  "livros": [
    {
      "autor": [
        {
          "id": 8,
          "nome": "Autor 1"
        }
      ],
      "genero": "Genero A",
      "id": 0,
      "isbn": "ISBN-1234",
      "preco": 23.99,
      "titulo": "Livro A"
    },
    {
      "autor": [
        {
          "id": 5,
          "nome": "Autor 2"
        }
      ],
      "genero": "Genero B",
      "id": 8,
      "isbn": "ISBN-4321",
      "preco": 19.99,
      "titulo": "Livro B"
    }
  ]
}
```

Fonte: O autor

Altere o valor desta propriedade para *application/xml* e o resultado será um XML.

O protocolo HTTP permite que vários tipos possam ser negociados entre cliente e servidor. Os tipos sempre contém este formato de *tipo/subtipo*, e cada tipo tem seu próprio objetivo:

- **Application:** Dados para aplicações.
- **Audio:** Formatos de áudio.
- **Imagem:** Formatos de imagem.
- **Video:** Formatos de vídeo.
- **Text:** Textos que podem ser entendidos por humanos.
- **VND:** Para tipos específicos de alguns programas.

Ainda é permitido que se passe mais de um tipo para o *Header Accept*, separando-os por vírgula, como por exemplo, *“application/json,application/xml”*. Neste caso é informado que o cliente prefere receber JSON, mas se o serviço não consegue retornar este tipo, pode-se aceitar XML.

URIS mais descritivas

O que faz um recurso ser um recurso é sua URI, uma vez que ela informa qual o nome do recurso e seu endereço. A URI é considerada o identificador de um recurso, logo podemos ter algo como:

- <http://localhost:8080/livraria-virtual/livro/1234>
- <http://localhost:8080/livraria-virtual/livro/9785467865475>
- <http://localhost:8080/livraria-virtual/livro/2019/01>

A primeira identifica um livro por um código específico, talvez o id no sistema, a segunda pode ser uma identificação dado seu ISBN e a terceira identifica os livros que foram lançados no mês de Janeiro de 2019.

Endereçabilidade

Um recurso precisa ser endereçável para que possa ser encontrado. Assim, quando colocarmos alguma das URI's apresentadas no item anterior no nosso navegador, iremos obter o mesmo recurso, onde quer que estejamos.

Esta é uma capacidade interessante não só para REST mas para o HTTP, pois com isso é possível realizar atividades com a de compartilhar recursos com outras pessoas simplesmente enviando a URI por e-mail, ou em post, livros,... também é possível marcá-lo como favorito e visitar mais tarde.

O @Path do JAX-RS

A anotação `@Path` do JAX-RS é responsável por tratar a URI da maneira que discutimos aqui. Com ela poderemos montar URI's mais descritivas através de utilização de variáveis.

Agora vamos implementar estes conceitos no nosso serviço de livreria. Na classe `LivroResource` adicione mais um método como o da figura 3.5.

Figura 3.5 - Método `getLivroPorIsbn` na classe `LivroResource.java`

```
@GET
@Path("/{isbn}")
public Livro getLivroPorIsbn(@PathParam("isbn") String id) {
    return livroRepo.getLivroPorIsbn(id);
}
```

Fonte: O autor

Além deste, vamos adicionar o método `getLivroPorIsbn` na classe `LivroRepositorio`. Ainda nesta classe, altere o modificador do atributo `livros` para `static`, figura 3.6.

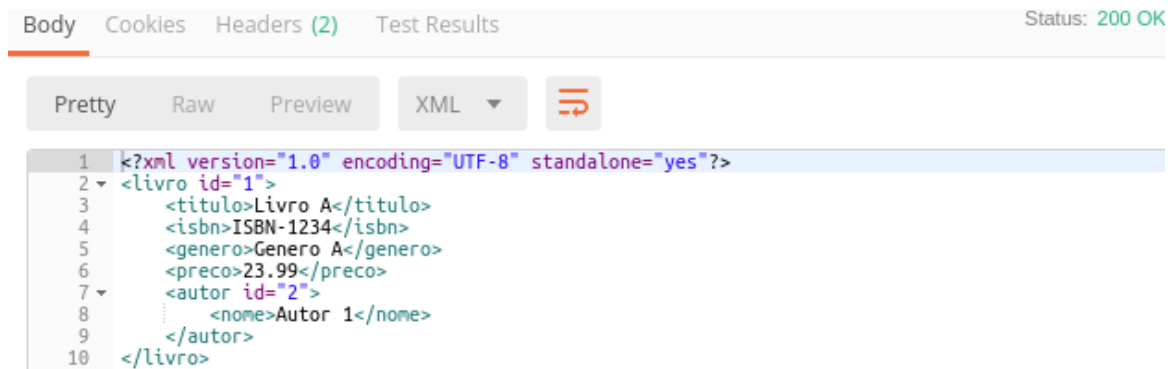
Figura 3.6 - Método `getLivroPorIsbn` na classe `LivroRepositorio.java`

```
public Livro getLivroPorIsbn(String isbn) {
    for (Livro livro : livros.values()) {
        if (isbn.equals(livro.getIsbn())) {
            return livro;
        }
    }
    return null;
}
```

Fonte: O autor

Para testar se está tudo funcionando, digite a URI no Postman `http://localhost:8080/livraria-virtual/livro/ISBN-1234`.

Figura 3.7 - Status 200 (OK) com apenas um livro



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <livro id="1">
3   <titulo>Livro A</titulo>
4   <isbn>ISBN-1234</isbn>
5   <genero>Genero A</genero>
6   <preco>23.99</preco>
7   <autor id="2">
8     <nome>Autor 1</nome>
9   </autor>
10 </livro>
```

Fonte: O autor

Métodos HTTP e a Interface Uniforme

Este princípio do REST diz para utilizar a interface de métodos já disponibilizada pelo protocolo HTTP para fazer interações com seus serviços. Os quatro métodos mais utilizados são:

1. **GET**: para recuperar recurso
2. **POST**: para criar um novo recurso
3. **PUT**: para atualizar um recurso existente
4. **DELETE**: para excluir um recurso

Esses são os mais utilizados pois mapeiam muito bem as 4 operações básicas do banco de dados, ou CRUD (Create, Retrieve, Update e Delete).

Se formos comparar com Orientação a objetos, teríamos estes métodos representados pela seguinte classe:

```
class Resource {
    Response get();
    Response post(Request r);
    Response put(Request r);
    Response delete();
}
```

Os métodos POST e PUT carregam em seu corpo de requisição os dados que serão inseridos ou atualizados no servidor. Os tipos de dados mais utilizados para este fim são o XML e o JSON. Todos os métodos retornam um response.

Além destes ainda existem os métodos HEAD e OPTIONS que são utilitários. HEAD recupera metadados sobre um recurso e OPTIONS permite descobrir o que é permitido fazer com um recurso.

Já o conceito de Interface Uniforme se dá por que você faz todas estas operações utilizando a mesma URI. Por exemplo, solicitar uma lista de livros através do path /livro, ou buscar por um livro específico através de /livro/{id}, inserir um livro através de /livro, atualizar um livro através de /livro, ou até mesmo excluir um livro através de /livro/{id}.

Códigos de Status

Outro princípio de REST diz respeito sobre utilizar corretamente os códigos de resposta do HTTP. Quando uma requisição é feita para um serviço de maneira exitosa, um código 200 (“OK”) será retornado, ou qualquer outro da série 2xx. Mas se algo der errado, um código de erro das séries 3xx, 4xx ou 5xx será retornado.

O protocolo HTTP possui estas séries distintas de código para que o cliente saiba como lidar com a resposta do serviço. Cada série possui seu significado.

Série 2xx

Códigos desta série indicam sucesso na requisição realizada. Os mais comuns são:

200 - OK

Operação realizada com sucesso.

201 - Created

Retornado após um POST, indica que o recurso foi criado corretamente. Retorna-se um cabeçalho *Location* juntamente com a resposta com a URI do recurso criado.

202 - Accepted

Utilizado para processamento assíncrono. Também retorna um cabeçalho *Location*.

204 - No content

Utilizado em POST e PUT quando o servidor não retorna dado algum.

Série 3xx

Esta série indica que deve se redirecionar a outro local para obter o recurso.

301 - Moved Permanently

Indica que o recurso procurado foi movido para outra URI. É retornado o cabeçalho *Location* indicando o local atualizado.

303 - See other

Indica que um processo assíncrono foi ou está sendo realizado e o recurso deve ser obtido em outro local. É retornado o cabeçalho *Location* indicando o local atualizado.

304 - Not modified

Utilizado em requisições GET, quando um recurso é cacheado. O serviço não retorna dados pois estes não sofreram modificações, indicando para o cliente utilizar o recurso pesquisado anteriormente.

307 - Temporary Redirect

Parecido com o 301, mas indica que o recurso foi movido temporariamente.

Série 4xx

Códigos desta série indicam que o cliente enviou dados errados na requisição.

400 - Bad Request

Código genérico indicando que ocorreu erro no processamento devido a qualquer erro nos dados enviados do cliente para o serviço.

401 - Unauthorized

Quando um cliente tenta acessar um recurso sem os dados de autenticação ou com autenticação inválida.

403 - Forbidden

Quando um cliente tenta acessar um recurso sem ter a permissão necessária.

404 - Not Found

Retornado quando o recurso solicitado não existe.

405 - Method Not Allowed

Quando o método utilizado na requisição não é suportado pelo serviço. Inclui o cabeçalho *Allow* indicando quais métodos podem ser utilizados.

409 - Conflict

Retornado em métodos POST, quando se tenta criar um recurso mas este recurso já existe. Retorna o cabeçalho *Location* indicando o local do recurso.

415 - Unsupported Media Type

Quando o cliente solicita ao serviço um tipo de dado que não é suportado.

Série 5xx

Códigos deste tipo referem-se a erros ocorridos por causa de problemas no servidor.

500 - Internal Server Error

Resposta genérica indicando que houve qualquer tipo de erro.

503 - Service Unavailable

Indica que o servidor está funcionando mas o serviço específico não está respondendo corretamente.

Lista completa de códigos

A lista completa de código pode ser consultados em:
<https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

404 Resource Not Found

E se tentarmos recuperar um recurso que não existe, o que deveria acontecer? Vimos que o protocolo HTTP possui o código de Status 404, que deve ser utilizado quando se tenta acessar um recurso que não existe mais. Por isso vamos fazer uma alteração nestes métodos para retornar este código ao cliente.

O método `getLivroPorIsbn` da classe `LivroRepositorio` irá lançar uma exceção caso não encontre o livro requerido.

Figura 3.8 - Lança exceção caso não encontre o livro

```

public Livro getLivroPorIsbn(String isbn) {
    for (Livro livro : livros.values()) {
        if (isbn.equals(livro.getIsbn())) {
            return livro;
        }
    }
    throw new LivroNaoEncontradoException();
}

```

Fonte: O autor

Criamos uma classe de exceção, a LivroNaoEncontradoException:

Figura 3.9 - Classe LivroNaoEncontradoException

```

public class LivroNaoEncontradoException
    extends RuntimeException {
}

```

Fonte: O autor

Agora, na classe LivroResource, fazemos o tratamento para retornar o código 404 ao cliente caso a exceção seja lançada.

Figura 3.10 - Retorna 404 caso não encontrar um livro

```

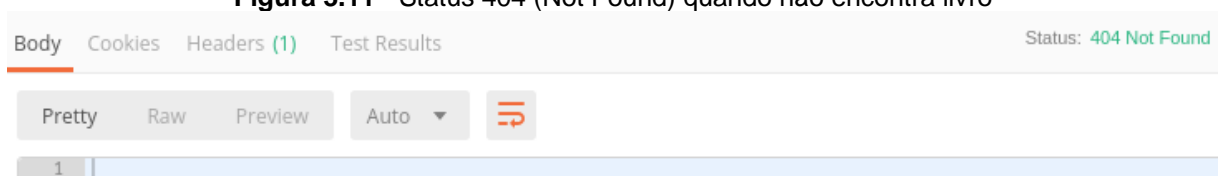
@GET
@Path("/{isbn}")
public Livro getLivroPorIsbn(@PathParam("isbn") String id) {
    try {
        return livroRepo.getLivroPorIsbn(id);
    } catch (LivroNaoEncontradoException e) {
        throw new WebApplicationException(Status.NOT_FOUND);
    }
}

```

Fonte: O autor

Tentando buscar no Postman <http://localhost:8080/livraria-virtual/livro/INEXISTENTE>:

Figura 3.11 - Status 404 (Not Found) quando não encontra livro



Fonte: O autor

Statelessness

Toda requisição HTTP é feita de maneira isolada. Isso quer dizer que quando um cliente faz uma requisição, nela são enviados todos os dados necessários para que o servidor possa realizar seu trabalho.

Um servidor não deveria manter o estado entre as requisições. Cabe ao cliente que está invocando o serviço, ou até mesmo o próprio recurso, a tarefa de manter o estado durante as transações entre cliente-servidor.

O benefício desta abordagem é proporcionar maior escalabilidade. Aumentar a quantidade de máquinas servidoras para processarem mais requisições de mais clientes seria muito difícil se cada máquina tivesse que manter o estado de seus clientes. Além disso, se uma dessas máquinas precisa ser desligadas para manutenção, todos clientes com sessões abertas ali perderiam todas informações.

Implementando os demais métodos http no nosso serviço

Agora com mais estes conhecimentos, vamos evoluir nosso serviço adicionando os demais métodos para formarmos as quatro operações básicas CRUD. Até agora já fizemos a implementação do método GET, onde é possível recuperar uma lista de livros ou um específico através de seu código ISBN.

Salvar um recurso com método POST

Agora vamos implementar o método POST para salvar um livro no nosso repositório. Este método irá adicionar um livro à lista contida na classe LivroRepositorio.java, na resposta irá retornar o código de status 201 (*Created*) e o cabeçalho *Location* contendo a URI do novo recurso criado.

A primeira coisa que vamos fazer é implementar o método adicionaLivro na classe LivroRepositorio para guardar o livro.

Figura 3.12 - Método que adiciona livro na lista de livros

```
public void adicionaLivro(Livro livro) {  
    livros.put(livro.getId(), livro);  
}
```

Fonte: O autor

Agora, na classe `LivroResource`, implemente o método `criaLivro` que atenderá à requisição feita com o método `POST`. Por isso vamos anotar este método com `@Post` do JAX-RS:

Figura 3.13 - Método `criaLivro` na classe `LivroResource.java`

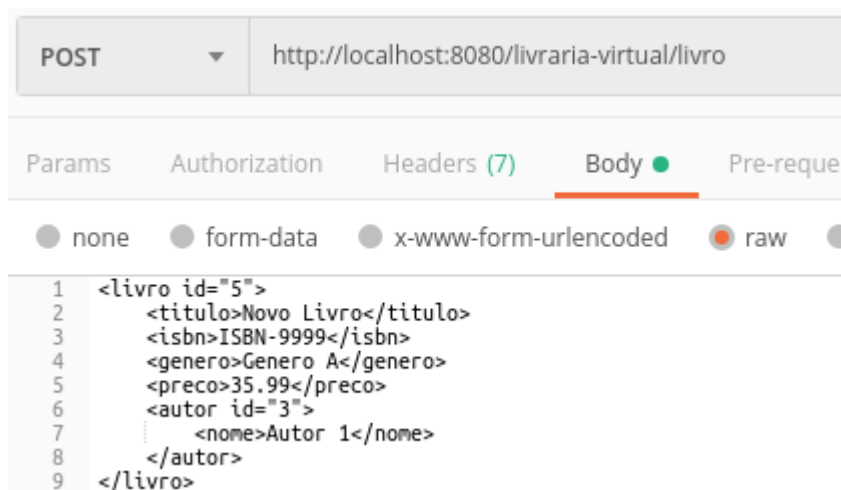
```
@POST
@Consumes(MediaType.APPLICATION_XML)
public Livro criaLivro(Livro livro) {
    livroRepo.adicionaLivro(livro);
    return livro;
}
```

Fonte: O autor

Veja que também anotamos este método com `@Consumes`. Esta anotação especifica qual o tipo de dado que o recurso espera receber do cliente. Desta forma o Jersey consegue fazer o parse do conteúdo XML que está vindo no corpo da requisição, convertendo para o tipo de objeto que está sendo esperado com parâmetro do método.

Vamos testar utilizando o Postman. Perceba que a URI a ser utilizada é a mesma que utilizamos para obter uma lista de livros do serviço. Apenas iremos alterar o método `http` para `POST`. Além disso, no corpo da requisição, montaremos um XML que representará os dados do livro.

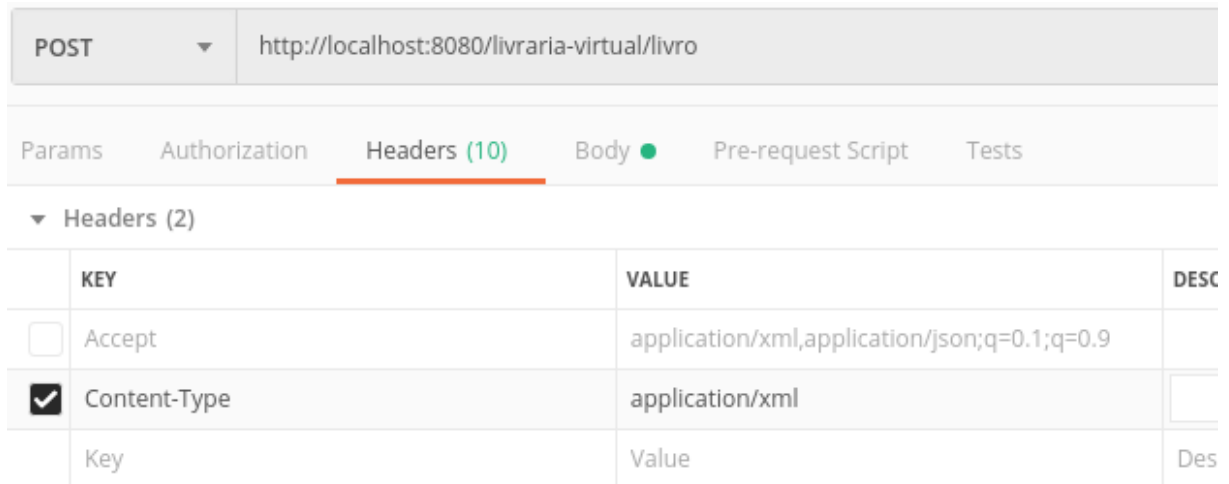
Figura 3.14 - Exemplo de XML a ser enviado pelo Postman



Fonte: O autor

Para escrever o XML acima, clique na aba `Body` e verifique se a opção `Raw` está selecionada. Depois clique na aba `Headers`, para adicionar o cabeçalho `content-type`, indicando que o conteúdo que está sendo enviado é um `application/xml`.

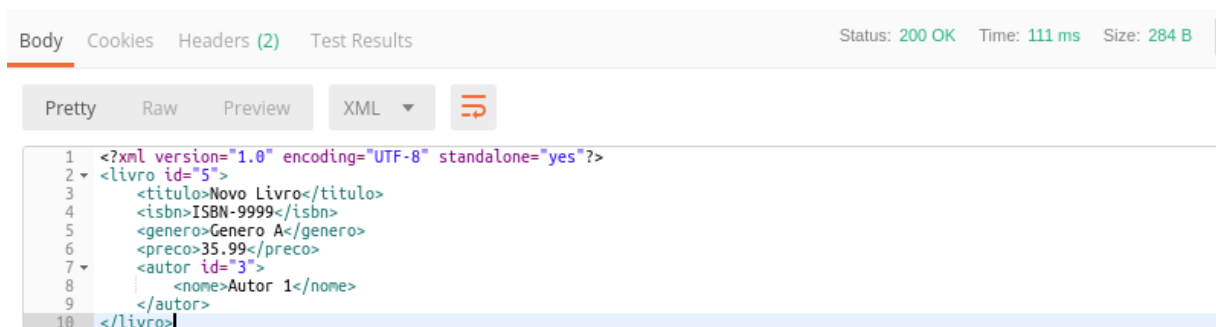
Figura 3.15 - Header Content-Type no método POST



Fonte: O autor

Clique em *Send*, o resultado abaixo será exibido.

Figura 3.16 - Retorno do método POST com código 200



Fonte: O autor

Tudo parece ter funcionado corretamente, exceto pelo fato de o código de retorno ter sido 200 e não 201. Vamos fazer esta alteração na no método `criaLivro`.

Figura 3.17 - Método `criaLivro` retornando código 201

```
@POST
@Consumes(MediaType.APPLICATION_XML)
public Response criaLivro(Livro livro) {
    livroRepo.adicionaLivro(livro);

    java.net.URI uriLocation = UriBuilder
        .fromPath("livro/{isbn}")
        .build(livro.getIsbn());

    return Response.created(uriLocation).entity(livro).build();
}
```

Fonte: O autor

A primeira coisa a se fazer é criar uma variável do tipo `java.net.URI` montando a URI que irá buscar o livro recém formado. Esta URI irá direcionar o usuário para o método `getLivroPorIsbn` criado anteriormente.

Depois disso, alteramos o tipo de retorno do método de Livro para Response. Para montar o retorno, foi utilizada a classe utilitária *Response*, invocando o método *created*, que é o responsável em retornar o código 201 como *Status Code*, encadeado do método *entity*, que coloca o livro recém criado no corpo da resposta, e por fim invocamos o método *build* para construir o objeto. Pronto, o resultado pode ser conferido no Postman.

Figura 3.18 - Status Code 201 e cabeçalho Location



Fonte: O autor

E se tentarmos passar um JSON representando um livro a este método? Receberemos como resposta um erro com o código 415 (*Unsupported Media Type*). Isso acontece por que nosso recurso só aceita XML. Vamos alterar para que passe a aceitar JSON adicionando à anotação *@Consumes* a constante de JSON.

Figura 3.19 - Método criaLivro aceita XML e JSON

```
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response criaLivro(Livro livro) {
    livroRepo.adicionaLivro(livro);

    java.net.URI uriLocation = UriBuilder
        .fromPath("livro/{isbn}")
        .build(livro.getIsbn());

    return Response.created(uriLocation).entity(livro).build();
}
```

Fonte: O autor

Agora sim podemos passar um JSON para o método que irá funcionar corretamente.

Figura 3.20 Retorno do recurso com JSON



```
1 {
2   "autor": [
3     {
4       "id": 7,
5       "nome": "Autor 1"
6     }
7   ],
8   "genero": "Genero A",
9   "id": 6,
10  "isbn": "ISBN-1010",
11  "preco": 18.99,
12  "titulo": "Livro JSON"
13 }
```

Fonte: O autor

O método POST não é seguro nem idempotente, logo, se tentarmos criar outro livro com o mesmo ID, deveríamos ter um erro indicando que já existe um recurso igual. Por isso vamos fazer uma alteração no método `adicionaLivro` da classe `LivroRepositorio`, lançando uma exceção se a lista já contiver um livro com mesmo ID.

Figura 3.21 - método `adicionaLivro` lança exceção se lista já contém livro

```
public void adicionaLivro(Livro livro) {
    if (livros.containsKey(livro.getId())) {
        throw new LivroExistenteException();
    }
    livros.put(livro.getId(), livro);
}
```

Fonte: O autor

Além disso criamos a classe `LivroExistenteException` que herda os comportamentos de `RuntimeException`.

Figura 3.22 - `LivroExistenteException`

```
public class LivroExistenteException
    extends RuntimeException {
}
```

Fonte: O autor

Agora precisamos alterar a classe `LivroResource` para capturar esta exceção e retornar um código de status que indique ao cliente que o recurso que ele está tentando criar já existe. O código indicado para este caso é o 409 *Conflict*.

Figura 3.23 - retornando o código 409 através de `WebApplicationException`

```

@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response criaLivro(Livro livro) {
    try {
        livroRepo.adicionaLivro(livro);
    } catch (LivroExistenteException e) {
        throw new WebApplicationException(Status.CONFLICT);
    }

    java.net.URI uriLocation = UriBuilder
        .fromPath("livro/{isbn}")
        .build(livro.getIsbn());

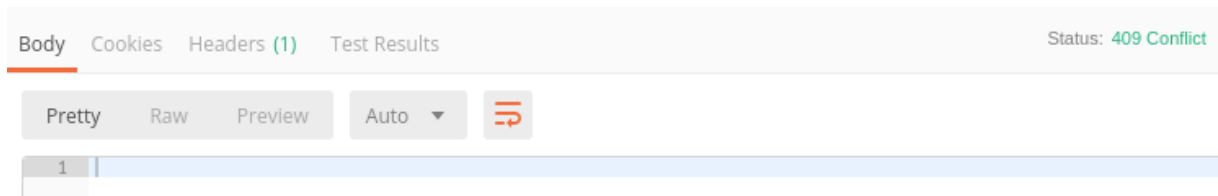
    return Response.created(uriLocation).entity(livro).build();
}

```

Fonte: O autor

Agora vamos testar tentando inserir o mesmo livro 2 vezes.

Figura 3.24 - Status Code 409 ao tentar salvar livro 2 vezes



Fonte: O autor

Para finalizar o CRUD, faltam apenas a atualização e remoção de recurso.

Atualizando um recurso com PUT

Utilizando os princípios vistos até agora, criaremos o método para atualizar o recurso na classe LivroRepositorio. Este método irá atualizar os valores dos demais atributos conforme os novos valores passados através do corpo da requisição feita pelo método HTTP PUT.

Figura 3.25 - Método atualizaLivro na classe LivroRepositorio

```

public void atualizaLivro(Livro livro) {
    livros.put(livro.getId(), livro);
}

```

Fonte: O autor

Também transforme esta classe em um singleton, para que a atualização de recursos tenha efeito:

Figura 3.26 - Singleton LivroRepositorio

```

private static LivroRepositorio repo;

public static LivroRepositorio getInstance() {
    if (repo == null)
        repo = new LivroRepositorio();

    return repo;
}

```

Fonte: O autor

Arrume a instânciação da classe LivroRepositorio em LivroResource:

Figura 3.27 - Instanciando LivroRepositorio

```

private LivroRepositorio livroRepo = LivroRepositorio.getInstance();

```

Fonte: O autor

E o método na classe LivroResource é implementado para responder à requisições feitas com o método PUT:

Figura 3.28 - Método que responde ao método HTTP PUT

```

@PUT
@Path("/{isbn}")
@Consumes({MediaType.APPLICATION_XML,
    MediaType.APPLICATION_JSON})
public Response atualizaLivro(@PathParam("isbn") String isbn,
    Livro livro) {

    try {
        Livro livroDoEstoque = livroRepo.getLivroPorIsbn(isbn);

        livroDoEstoque.setAutor(livro.getAutor());
        livroDoEstoque.setGenero(livro.getGenero());
        livroDoEstoque.setPreco(livro.getPreco());
        livroDoEstoque.setTitulo(livro.getTitulo());

        livroRepo.atualizaLivro(livroDoEstoque);
    } catch (LivroNaoEcontradoException e) {
        throw new WebApplicationException(Status.NOT_FOUND);
    }

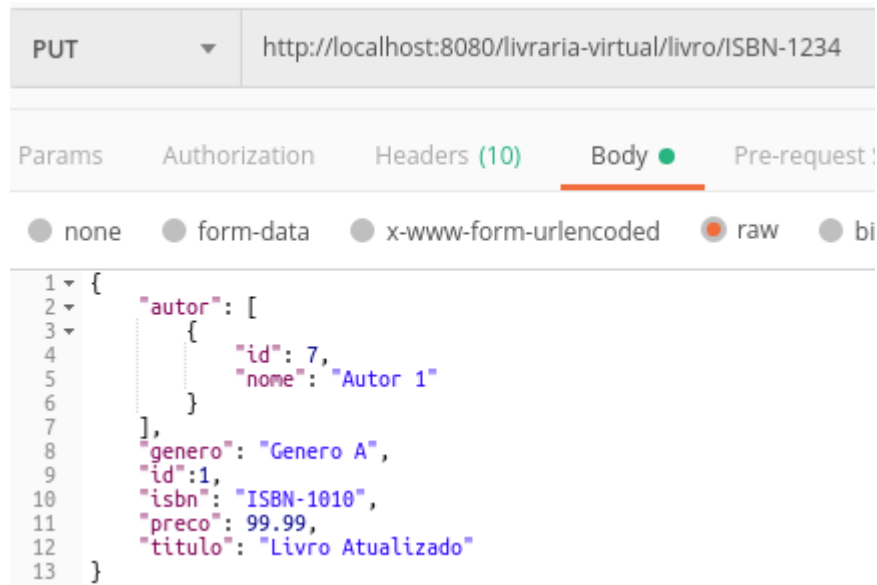
    return Response.ok().entity(livro).build();
}

```

Fonte: O autor

Perceba que retornamos um Response.ok(), para retornar o código 200 ao cliente.

Figura 3.29 - Requisição com HTTP PUT



Fonte: O autor

Excluindo um recurso com DELETE

Iremos utilizar os mesmos princípios do método anterior para implementar a exclusão de um recurso. A diferença é que o método DELETE não contém corpo de requisição, por isso vamos passar o ID do livro como parâmetro, como fizemos no método de busca por livro específico GET.

Primeiro vamos implementar a exclusão na classe LivroRepositorio.

Figura 3.30 - Método remove livro da classe LivroRepositorio

```

public void removeLivro(Long id) {
    if (livros.containsKey(id)) {
        livros.remove(id);
    } else {
        throw new LivroNaoEcontradoException();
    }
}

```

Fonte: O autor

Depois implementamos o método que responde à requisições feitas pelo método HTTP DELETE:

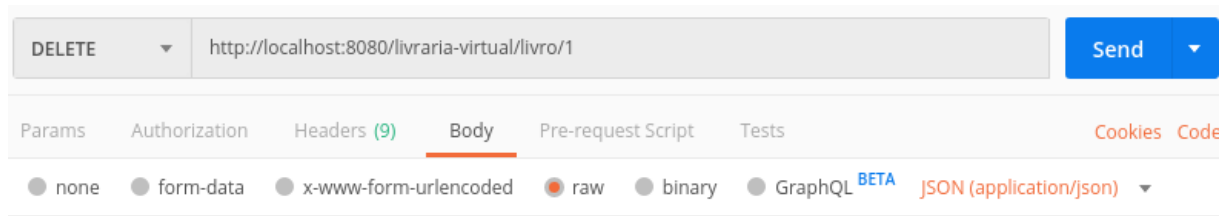
Figura 3.31 - Método remove livro da classe LivroResource

```
@DELETE
@Path("/{id}")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public void removeLivro(@PathParam("id") Long id) {
    try {
        livroRepo.removeLivro(id);
    } catch (LivroNaoEncontradoException e) {
        throw new WebApplicationException(Status.NOT_FOUND);
    }
}
```

Fonte: O autor

Para testar no POSTMAN utilize uma URI como a seguir e escolha o método DELETE.

Figura 3.32 - Invocando DELETE no Postman



Fonte: O autor

O *Status Code* neste caso é um 204 *No Content*.

Considerações Finais

Este capítulo mostrou alguns princípios importantes quanto ao estilo de arquitetura REST com Java.

O primeiro princípio visto foi que os recursos podem ter múltiplas representações, como XML e JSON. Fizemos isso quando alteramos o método (GET) de listagem de livros para que retornasse os dois tipos de dados dependendo do valor contido no cabeçalho Accept.

O princípio de utilizar URI's mais descritivas foi visto quando implementamos o método GET para recuperar um recurso específico. Vimos que todo recurso passa a ter um identificador único com este princípio.

Depois começamos a ver a importância do princípio da *Uniform Interface*, implementando métodos específicos para cada ação que o cliente quer realizar. Neste capítulo implementamos o método POST para salvar um recurso.

O princípio de *Statelessness* diz que o servidor não deve manter o estado das requisições, por isso o método POST foi implementado de tal maneira que o corpo da requisição carregasse todos dados necessários para seu processamento.

Além destes princípios vimos que a comunicação entre consumidor e servidor fica muito mais eficiente quando se faz o uso correto de *Status Code* nas requisições.

UNIDADE 4 - REST COM JAVA, HATEOAS e JAX-RS Client

Neste capítulo vamos ver a importante relação no quesito segurança e idempotência dos métodos HTTP utilizados para o desenvolvimento do CRUD de livros.

Também iremos estudar o último princípio de REST, vendo como as interações podem ser ditadas pelo servidor ao cliente utilizando *Hypermedia*, o que é chamado de HATEOAS.

Além disso, vamos utilizar uma API do JAX-RS para implementar um cliente capaz de consumir um serviço REST.

Por fim vamos entender o que é e como usar o WADL para descobrir quais métodos e tipos de recursos um serviço desenvolvido por terceiros disponibiliza.

Segurança e Idempotência

Um método é considerado seguro se, quando invocado, não causar alterações no recurso. Como o método GET, onde uma requisição feita para se obter um recurso.

Já os métodos POST, PUT e DELETE não são considerados seguros, pois estes métodos são designados para requisições onde haja alteração de dados.

Basicamente, a segurança aqui diz respeito sobre o quão confiável é fazer uma requisição considerando o método utilizado. Por exemplo, você poderá fazer uma requisição GET sempre que precisar sem medo de estar alterando ou apagando informações. Já os outros métodos o risco de alterar dados existe.

Idempotência vem do conceito da matemática que diz que sempre que uma operação for feita repetidas vezes e retornar sempre o mesmo resultado, esta operação é dita idempotente. Por exemplo, qualquer multiplicação por 0 é sempre zero, não importa quantas vezes você multiplicar.

Da mesma maneira ocorre com um recurso. É considerado idempotente, se ao fazer uma requisição, seu resultado é o mesmo ao se fazer uma série de requisições. Ou seja, a cada requisição realizada, o estado deixado do recurso é exatamente o mesmo da primeira requisição.

Os métodos PUT e DELETE são considerados idempotentes. PUT é utilizado para atualizar um recurso. Se você atualizar um recurso várias vezes, o recurso estará, ao final das requisições, sempre com o mesmo estado. O mesmo ocorre com o DELETE,

ao se apagar um recurso ele deixa de existir, se fizer uma nova requisição DELETE, o recurso ainda terá deixado de existir. Já o método POST não é considerado idempotente, pois a cada requisição um novo recurso é criado.

	Idempotente	Seguro
GET	x	x
POST		
PUT	x	
DELETE	x	

Tabela 3.1 - Métodos conforme segurança e idempotência

HATEOAS

Hypermedia as the Engine of Application State, ou HATEOAS. Em tradução livre seria algo como Hypermedia como o Motor de Estado da Aplicação. Não se preocupe pois, apesar do nome, este princípio não é complexo de compreender.

O primeiro ponto que temos que ter claro em mente é o que é *Hypermedia*? Quem já está mais acostumado com desenvolvimento Web, ou que tenha mais vivência com a Internet, sabe que as páginas dos sites possuem textos e muitos outros links de recursos que são carregados automaticamente pelo navegador. Por exemplo, arquivos com scripts JavaScript, folhas de estilos CSS, imagens, sons, etc. Além é claro dos links para as outras páginas que as pessoas podem acessar.

Figura 4.1 - Links em uma página html

```
<html>
<head>
  <link rel="icon" href="/img/favicon.ico"/>
  <link rel="stylesheet" href="/css/estilo.css" type="text/css"/>
</head>

<body>
  ...
</body>
</html>
```

Fonte: O autor

A figura 4.1 mostra um trecho de página HTML que mostra alguns destes links. Estes links ditam para o navegador como esta página HTML deve ser renderizada, informando onde ele deve buscar os recursos necessários, como o ícone da página e a folha de estilo.

Em outras palavras, podemos ter que *Hypermedia* faz referência a qualquer conteúdo que contenha links para outros tipos de recursos, independente de seu formato, seja texto, imagem, etc.

Roy Fielding, o pai do REST, enfatizou este comportamento trazendo o princípio do HATEOAS a este estilo de arquitetura de software. Com ele, você usa os links nas respostas de seus serviços, que ditam para os clientes quais os próximos passos disponíveis para navegarem.

Por exemplo, no nosso exemplo de livraria virtual, imagine que o cliente faça uma busca por um determinado livro, a resposta do serviço irá trazer além dos dados do livro um link para a próxima ação disponível, que pode ser ou uma URI para que o usuário possa fazer a compra do livro, ou uma URI para que o usuário seja avisado quando o livro estiver disponível caso o exemplar já tenha sido esgotado.

Perceba que este princípio traz muita dinâmica para a comunicação cliente servidor. Isso por que o cliente não precisa saber de antemão quais passos ele pode seguir, o serviço irá informar.

Além disso, traz um desacoplamento entre cliente-servidor. Imagine neste cenário que o link que direcionará o usuário para realizar a compra do livro seja tratado por um sistema e que o tratamento de avisar quando o livro chegar seja feito por outro sistema. Com HATEOAS esta transação fica transparente para o cliente.

Figura 4.2 - Exemplo de link para compra de livro

```
<itemBusca id="1234">
  <livro id="1">
    <titulo>Livro A</titulo>
    <isbn>ISBN-1234</isbn>
    <preco>23.99</preco>
  </livro>
  <link rel="compra" href="/compra/1234"/>
</itemBusca>
```

Fonte: O autor

A figura 4.2 mostra como seria um exemplo de retorno em XML contendo um link para a URI de compra do livro.

Note que a tag link do exemplo possui dois atributos, *rel* e *href*. O atributo *rel* é um texto auxiliar para o cliente, enquanto *href* é a URI do recurso.

Além destes existem os atributos *title*, que traz uma descrição entendível para humanos, *method*, indica quais métodos podem ser utilizados para invocar a URI e *type* que indica os media types suportados. Estes 2 últimos são poucos utilizados.

Implementado HATEOAS com JAX-RS

O Jersey, oferece uma maneira de criarmos Hyperlinks por meio da classe utilitária `Link`, que facilita o trabalho de construir URI's programaticamente.

Para demonstrar o uso do HATEOAS no nosso serviço usando JAX-RS, iremos fazer uma alteração na classe `LivroResource.java`, mais especificamente no método `getLivroPorIsbn()`, fazendo com que retorne um objeto `ItemBusca` que encapsula o objeto livro e um objeto link, que indicará qual a próxima ação que o cliente poderá tomar por meio de uma URI.

No nosso exemplo, vamos retornar uma URI para que o usuário possa adicionar o item buscado no carrinho de compras.

Figura 4.3 Classe `ItemBusca`

```
@XmlRootElement(name = "itemBusca")
@XmlAccessorType(XmlAccessType.FIELD)
public class ItemBusca {

    @XmlElement
    private Livro livro;

    @XmlElement
    private List<Link> links = new ArrayList<Link>();

    public void addLink(Link link) {
        links.add(link);
    }

    // Getters and Setters omitidos
}
```

Fonte: O autor

Primeiro criamos uma classe Chamada `ItemBusca` como a acima.

Atenção, a classe `Link` é do pacote `javax.ws.rs.core.Link`, que representa o Web Link na especificação JAX-RS. Lembre-se de adicionar os métodos *Getters and Setters* dos atributos desta classe. Além disso, adicionamos um método `addLink` para facilitar a inclusão de links à lista.

Agora, vamos alterar o método `getLivroPorIsbn` da classe `LivroResource.java` para que fique como abaixo.

Figura 4.4 - Método `getLivroPorIsbn` retornando Hyperlinks

```
@GET
@Path("/{isbn}")
public ItemBusca getLivroPorIsbn(@PathParam("isbn") String id) {
    try {
        Livro livro = livroRepo.getLivroPorIsbn(id);

        ItemBusca item = new ItemBusca();
        item.setLivro(livro);

        Link link = Link
            .fromUri("/carrinho/" + livro.getId())
            .rel("carrinho")
            .type("POST").build();

        item.addLink(link);

        return item;
    } catch (LivroNaoEncontradoException e) {
        throw new WebApplicationException(Status.NOT_FOUND);
    }
}
```

Fonte: O autor

Muitas alterações foram realizadas aqui. Primeiro o retorno do método foi alterado para o tipo `ItemBusca` criado anteriormente. Dentro do método, o objeto de retorno foi montado contendo o livro e um link. O livro foi obtido a partir do repositório, como já estava fazendo. O link, foi construído a partir da classe `Link`, que possui uma interface fluente e auxilia na construção dos atributos do link vistos na seção anterior. O método `fromUri`, indica qual será a URI do link, o método `rel` é o texto auxiliar, o `type` indica o tipo de ação permitida para esta URI, neste caso um POST, e por fim o método `build` é invocado para concluir a construção do objeto `Link`.

Neste momento, se você testar o serviço utilizando o Postman, verá apenas uma tag “link” vazia. Isto acontece por que o JAX-RS não sabe como fazer a serialização deste objeto para a mensagem que está sendo retornada para o cliente. Para termos isso, precisamos informar a ele que use um Adaptador que será responsável em fazer o *marshall* e *unmarshall* deste atributo.

Voltando na classe `ItemBusca`, adicione a anotação `@XmlJavaTypeAdapter` em cima do atributo `links`.

Figura 4.5 -XmlJavaTypeAdapter

```
@XmlRootElement(name = "itemBusca")
@XmlAccessorType(XmlAccessType.FIELD)
public class ItemBusca {

    @XmlElement
    private Livro livro;

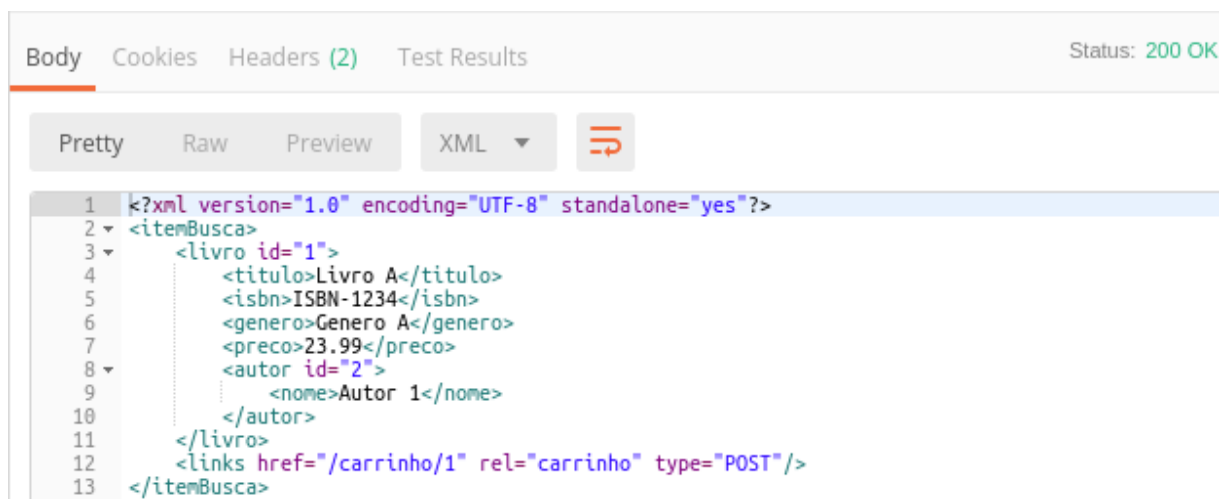
    @XmlElement
    @XmlJavaTypeAdapter(Link.JaxbAdapter.class)
    private List<Link> links = new ArrayList<Link>();
```

Fonte: O autor

Aqui estamos informando ao Jersey que utilize o adaptador do JAXB para serializar o atributo links.

Agora sim, testando no Postman obtemos o resultado esperado.

Figura 4.6 Retorno no Postman com Hyperlink



Fonte: O autor

Neste exemplo, a URI “/carrinho/1” contida no link informa ao cliente que ao fazer um POST ele estará adicionando o livro no carrinho de compras.

Da mesma maneira, uma lógica poderia ser feita no serviço para que, caso o livro estivesse em falta, retornasse o link do tipo “Avise-me quando este produto estiver disponível”, ou então “Adicione na lista de desejos”.

É desta forma que REST indica a mudança de estados da aplicação, dependendo da situação retorna um ou outro link.

Cliente REST com JAX-RS

Vimos como desenvolver um serviço utilizando estilo de arquitetura REST, agora, iremos implementar um cliente que seja capaz de consumir um serviço desenvolvido com esta arquitetura. Esta é uma habilidade muito importante, pois em sua carreira como desenvolvedor de software, não será raro atividades que demandem consumir serviços REST desenvolvidos por terceiros.

O JAX-RS provê uma API para desenvolvimento de clientes REST. Chamada de JAX-RS Client API, é uma interface fluente que tem por objetivo facilitar a comunicação com serviços REST expostos via protocolo HTTP.

Esta API pode ser utilizada para consumir qualquer serviço exposto em HTTP independentemente da linguagem de programação em que este serviço tenha sido escrito.

Como vimos anteriormente, o princípio de *Uniform Interface*, é muito importante pois possibilita que qualquer cliente, seja um navegador ou um cliente desenvolvido em qualquer linguagem de programação, possa utilizar a mesma interface para comunicar com qualquer serviço. Isto traz algumas vantagens:

- Simplicidade: arquitetura fácil de entender e manter.
- Baixo acoplamento: favorece a evolução de clientes e serviços.

Para que um cliente REST funcione, o serviço exposto (a ser consumido) deve:

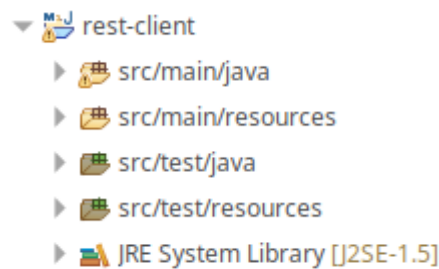
- Ter recursos identificados por URI.
- Possibilitar interação feita sobre protocolo HTTP via Request e Response utilizando os métodos HTTP padrão (GET, POST, PUT, DELETE, etc).
- Retornar representações identificáveis via media types.
- Fornecer links dentro da resposta para outros recursos (HATEOAS).

Voltando à API fornecida por JAX-RS, ela é fácil de usar e possibilita reutilização em várias partes do sistema, pois faz uso de anotações Java. Faz uso de classes utilitárias como UriBuilder que facilitam a criação de URIs, utilizando o padrão de interface fluente para construir as diversas requisições para o serviço.

Implementação de client REST com JAX-RS

Aqui iremos implementar um cliente que será capaz de consumir o serviço que desenvolvemos anteriormente. Primeiro, vamos criar um novo projeto Java separado do serviço que vínhamos desenvolvendo, para deixar claro que estamos em contextos totalmente separados.

Figura 4.7 - estrutura básica do projeto cliente



Fonte: O autor

Depois de criado o projeto, vamos adicionar as dependências necessárias:

```
<dependencies>
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>${jersey.version}</version>
  </dependency>

  <dependency>
    <groupId>org.glassfish.jersey.inject</groupId>
    <artifactId>jersey-hk2</artifactId>
    <version>${jersey.version}</version>
  </dependency>

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

  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-grizzly2-http</artifactId>
    <version>${jersey.version}</version>
  </dependency>
</dependencies>

<properties>
  <jersey.version>2.28</jersey.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Depois vamos criar uma classe que conterá o método *main*, esta será a porta de entrada do nosso cliente, onde ele será executado. Dentro deste método iremos construir um objeto *Client*, utilizando a classe útil *ClientBuilder*:

Figura 4.8 - objeto Client

```
public static void main(String[] args) {  
    Client client = ClientBuilder.newClient();  
}
```

Fonte: O autor

Agora podemos detalhar as informações necessárias para fazer requisições para o serviço. Depois iremos invocar o serviço e obteremos como resposta um objeto Java, no nosso caso um objeto do tipo Livro, já preenchido.

A partir do objeto *Client* instanciado, vamos criar um objeto do tipo *WebTarget*.

Figura 4.9 WebTarget

```
public static void main(String[] args) {  
    Client client = ClientBuilder.newClient();  
  
    WebTarget targetBase = client.target(  
        "http://localhost:8080/livraria-virtual/");  
}
```

Fonte: O autor

Perceba que criamos o *WebTarget* a partir do objeto client. O método target possui uma sobrecarga que permite a criação por vários modos, seja passando uma String contendo a URI ou utilizando diretamente um objeto URI. Aqui utilizamos uma String para facilitar. Além disso este target foi nomeado de targetBase e a URI aponta para a raiz, ou o contexto da aplicação livraria virtual. Esta URI não possui nenhum recurso que podemos invocar, mas ela servirá como base para criar os outros targets que apontarão para os recursos. Isso é interessante pois evita a repetição desta URI em várias partes do código.

Agora a partir de targetBase, vamos criar outro target apontando para a URI “livro”. Esta forma em que a API é disponibilizada é muito interessante, pois podemos criar um target como base e disponibilizá-la para todo o resto da aplicação. Assim, a cada novo recurso que se queira consumir, basta criar um target a partir do target base.

Figura 4.10 - WebTarget para “livro”

```
public static void main(String[] args) {  
    Client client = ClientBuilder.newClient();  
  
    WebTarget targetBase = client.target(  
        "http://localhost:8080/livraria-virtual/");  
    WebTarget targetLivro = targetBase.path("livro");  
}
```

Fonte: O autor

Com este target poderemos fazer o request para o serviço, então o próximo passo é invocar o método request do objeto targetLivro. Este método espera receber como parâmetro qual Media Type iremos solicitar ao serviço. Lembre-se que nosso serviço está preparado para fornecer tanto representações em XML ou JSON, desta forma poderemos solicitar qualquer um destes tipos:

Figura 4.11 - Informando para o request o media type XML

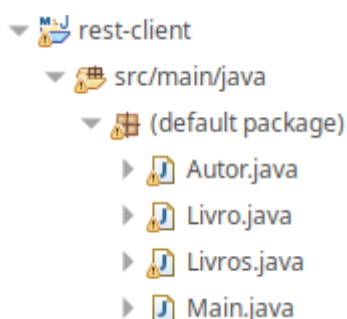
```
public static void main(String[] args) {  
    Client client = ClientBuilder.newClient();  
  
    WebTarget targetBase = client.target(  
        "http://localhost:8080/livraria-virtual/");  
    WebTarget targetLivro = targetBase.path("livro");  
  
    Builder request = targetLivro.request(MediaType.APPLICATION_XML);  
}
```

Fonte: O autor

Note que o request retorna um objeto do tipo *Builder*. Este objeto serve para utilizar a interface fluente, logo iremos fazer uma modificação neste código onde isso fará sentido.

A partir deste ponto já podemos utilizar qualquer um dos métodos HTTP disponíveis, get, post, etc. Mas antes precisamos fazer algo importante: já que iremos invocar o método GET, e este retorna uma lista de Livros, temos que ter dentro do projeto cliente classes que representem o livro tal qual é fornecido pelo serviço.

Figura 4.12 - Classes do projeto cliente



Fonte: O autor

Para isso, para ficar mais fácil, vamos copiar as classes Livro, Autor e Livros do projeto anterior para dentro do projeto cliente como mostra a figura 4.12.

Com isso podemos invocar o método *get* para obter uma lista de Livros:

Figura 4.13 - Invocando o método get

```
public static void main(String[] args) {
    Client client = ClientBuilder.newClient();

    WebTarget targetBase = client.target(
        "http://localhost:8080/livraria-virtual/");
    WebTarget targetLivro = targetBase.path("livro");

    Builder request = targetLivro.request(MediaType.APPLICATION_XML);
    Livros livros = request.get(Livros.class);
}
```

Fonte: O autor

Aqui invocamos o método get do objeto request. Este método espera como parâmetro que informemos o tipo de objeto que será retornado pelo serviço. desta maneira o JAX-RS consegue transformar o XML do serviço em classes Java do nosso cliente.

Para testar, suba o serviço livraria-virtual e execute este método main. Para visualizar o retorno pode colocar um laço for na lista de livros e imprimir o título de cada livro:

Figura 4.14 - Iterando sobre os livros retornados pelo Serviço REST

```
for (Livro livro : livros.getLivros()) {
    System.out.println(livro.getTitulo());
}
```

Fonte: O autor

Interface fluente de JAX-RS Client API

Eu disse na seção anterior que esta API fornece uma interface fluente, então vamos ver na prática o benefício disso.

Figura 4.15 - cliente com interface fluente

```
public static void main(String[] args) {

    Livros livros = ClientBuilder.newClient()
        .target("http://localhost:8080/livraria-virtual/")
        .path("livro")
        .request(MediaType.APPLICATION_XML)
        .get(Livros.class);

    for (Livro livro : livros.getLivros())
        System.out.println(livro.getTitulo());
}
```

Fonte: O autor

Este código faz exatamente a mesma coisa que fizemos, mas de uma forma direta, fluente. Assim fica muito fácil de criar um cliente direto que consome um serviço REST.

Invocando um método POST

Outra maneira que podemos interagir com nosso serviço REST é através do método HTTP Post para fazer a criação de um novo Livro.

Podemos fazer isso de modo similar como fizemos ao invocar o método GET. A diferença aqui é que o POST espera que um objeto seja enviado ao serviço, no caso, um objeto do tipo livro que será salvo. Logo, a primeira coisa que precisamos fazer é criar um novo objeto Livro.

Figura 4.16 - Instância de novo Livro

```
Livro livro = new Livro(
    3L, "Livro C", "ISBN-9999", "Gênero A", 19.99, "Autor 1");
```

Fonte: O autor

Depois disso, vamos montar a requisição para o método POST.

Figura 4.17 - Requisição ao método POST

```
Livro livroResposta = ClientBuilder.newClient()
    .target("http://localhost:8080/livraria-virtual/")
    .path("livro")
    .request()
    .post(Entity.entity(livro, MediaType.APPLICATION_XML),
        Livro.class);
```

```
System.out.println(livroResposta.getTitulo());
```

Fonte: O autor

Vamos analisar o que está acontecendo aqui. Primeiro utilizamos a classe `ClientBuilder` para criar o client, apontando para a URI do recurso livro, como fizemos anteriormente. Já o método `request` ficou um pouco diferente, nele não passamos o parâmetro `MediaType`. Por fim invocamos o método `post`, que irá fazer a requisição ao serviço. Este método esperar receber por parâmetro um objeto do tipo *Entity*, que é a representação do objeto que vamos enviar ao serviço. Aqui informamos a ele que a entidade irá carregar o objeto livro em forma de XML, o indicativo disto é o parâmetro `MediaType.APPLICATION_XML`. Informamos também ao método `post` um segundo parâmetro com o valor `Livro.class`. Este informa que estamos esperando receber como resposta um objeto do tipo Livro.

Há uma outra maneira mais indicada para fazer esta mesma requisição. Lembre-se que estamos invocando um método POST, que não é seguro nem idempotente. Logo, podem ocorrer erros que deveremos tratar no lado do cliente. Por exemplo, caso um livro seja criado com sucesso, o serviço irá retornar o *Status Code 201 - Created*, juntamente com o *header Location*, contendo uma URI que leva até o recurso recém criado. Fizemos este teste no capítulo anterior. Neste caso, caso nosso cliente necessite mostrar dados deste novo recurso, primeiro vamos precisar verificar o status retornado e depois pegar a URI Location. Igualmente, caso algum erro aconteça, o código será diferente de 201, e então poderemos fazer o tratamento mais adequado para o caso.

Por isso, vamos alterar este código para ficar como mostrado na figura 4.18:

Figura 4.18 - Requisição ao método POST mais indicada

```
Response response = ClientBuilder.newClient()  
    .target("http://localhost:8080/livraria-virtual/")  
    .path("livro")  
    .request()  
    .post(Entity.xml(livro));  
  
System.out.println(response.getStatus());  
System.out.println(response.getLocation());
```

Fonte: O autor

Note que agora estamos obtendo da chamada ao método POST um objeto do tipo *Response*. Este objeto contém métodos que auxiliam na obtenção do *Status Code* e outras informações retornadas no *header*.

Veja que a diferença aqui é que não mais informamos ao método *post* o tipo de objeto que esperamos receber. Este por sua vez retorna um objeto do tipo *Response*. Também alteramos a maneira de construir o *Entity*, como queremos enviar um objeto em tipo XML, podemos utilizar diretamente o método *xml*, que encapsula dentro dele a criação do objeto com o Media Type igual a XML.

Com a implementação assim, podemos fazer uma verificação mais fina no retorno do serviço e adicionar algumas validações. Por exemplo, caso o código retornado seja 201, monte uma requisição para o método GET contendo a URI retornada pelo método

`response.getLocation()`). Caso seja um código da série 4XX, prepare uma mensagem informando que algum dados não foi enviado corretamente.

Figura 4.19 - Exemplo de como tratar o Response

```
if (response.getStatus() == Status.CREATED.getStatusCode()) {
    ItemBusca item = ClientBuilder.newClient()
        .target(response.getLocation())
        .request(MediaType.APPLICATION_XML)
        .get(ItemBusca.class);

    System.out.println(item.getLivro().getTitulo());

} else if (response.getStatus() == Status.BAD_REQUEST.getStatusCode()) {
    System.out.println("Verifique se todos dados estão corretos");
}
```

Fonte: O autor

Acima é um exemplo de como poderia ficar o tratamento da resposta do serviço. Poderia ser adicionado o tratamento específico para outros códigos de status, como verificar se é necessário fazer redirecionamento (códigos 3xx) para outra URI, ou se há algum erro do servidor (códigos 5XX) e fazer tentativas mais tarde, etc.

WADL

Web Application Description Language, WADL, é uma documentação que descreve o Web Service que está publicado. Ele é muito útil para que você saiba o que recursos, URIs, métodos, etc estão sendo expostos.

No tópico anterior nós criamos um cliente para o serviço que havíamos desenvolvido anteriormente. Mas imagine que você tivesse que desenvolver um client para um Web Service desenvolvido por outra pessoa. Como você saberia que URI está exposta? Que recursos são retornados?

É para isso que serve o WADL. Ele é como se fosse o WSDL de serviços desenvolvidos com o protocolo SOAP. O WADL é baseado em XML e é publicado juntamente com o serviço em um endereço específico.

O Jersey já publica automaticamente um WADL juntamente com o serviço exposto, e pode ser acessado em `http://localhost:8080/livraria-virtual/application.wadl`. Vamos entender o que quer dizer cada parte deste documento.

Figura 4.20 - Tag doc do WADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.28 2019-01-25 15:18:13"/>
  <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="This is simplified WADL with user and core virtual/application.wadl?detail=true"/>
  <resources>
    <resource path="/livraria-virtual/">
      <resource path="/application.wadl" method="GET"/>
    </resource>
  </resources>
</application>
```


Fonte: O autor

A primeira tag que devemos olhar no WADL é a “doc”. Ela é opcional e tem a função de prover uma documentação inicial para o desenvolvedor. O Jersey gera uma tag doc contendo a versão em que o serviço foi desenvolvido.

Figura 4.21 - tag grammars do WADL

```
<grammars>
  <include href="application.wadl/xsd0.xsd">
    <doc title="Generated" xml:lang="en"/>
  </include>
</grammars>
```

Fonte: O autor

Nesta seção estão descritos os XSDs que são usados pelo serviço. Você poderá acessá-los em <http://localhost:8080/livraria-virtual/application.wadl/xsd0.xsd>. Arquivos XSDs são responsáveis em descrever a estrutura de arquivos XMLs. Ele conterá todos os elementos que formam os recursos disponibilizados pelo serviço. Este arquivo é extremamente útil, pois é possível gerar as classes Java que representam os recursos, e que serão recebidos ou enviados pelos serviço.

Figura 4.22 - tag resources do WADL

```
<resources base="http://localhost:8080/livraria-virtual/">
  <resource path="livro">
```

Fonte: O autor

Esta seção descreve os resources disponíveis no serviço. Perceba que o atributo *base* contém a URI que usamos como base para todas URIS do nosso cliente. Já a tag resource aninhada contém o path do recurso.

Figura 4.23 - tag method do WADL

```
<method id="criaLivro" name="POST">
  <request>
    <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02" xmlns=""
      element="livro" mediaType="application/xml"/>
    <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02" xmlns=""
      element="livro" mediaType="application/json"/>
  </request>
  <response>
    <representation mediaType="application/xml"/>
    <representation mediaType="application/json"/>
  </response>
</method>
```

Fonte: O autor

Depois o arquivo contém uma seção com várias tags method. Estas descrevem cada método exposto. O atributo id contém o nome do método Java. O atributo name contém o nome do método HTTP, neste exemplo é o POST. Aninhada a esta tag vem a tag request, que descreve o tipo de dado que deve ser enviado ao serviço e a tag

response, que informa o mediaType que será retornado. A tag response também pode trazer uma descrição do objeto a ser retornado, é o caso de métodos GET.

E é basicamente isso. Uma documentação muito simples e que auxilia muito o desenvolvedor que irá criar um cliente para o serviço.

Considerações Finais

Neste capítulo vimos os conceitos de segurança e idempotência de cada método HTTP. Depois vimos o último princípio do estilo de arquitetura que torna um serviço RESTful, o HATEOAS, que provê um motor de estado para a aplicação. Desenvolvemos um cliente utilizando a API do JAX-RS que é capaz de se comunicar com serviços REST expostos, independente da linguagem em que foram escritos. Por fim, vimos que serviços RESTs pode fornecer um arquivo WADL que descreve os recursos disponibilizados por ele.

UNIDADE 5 - SEGURANÇA EM SERVIÇO REST

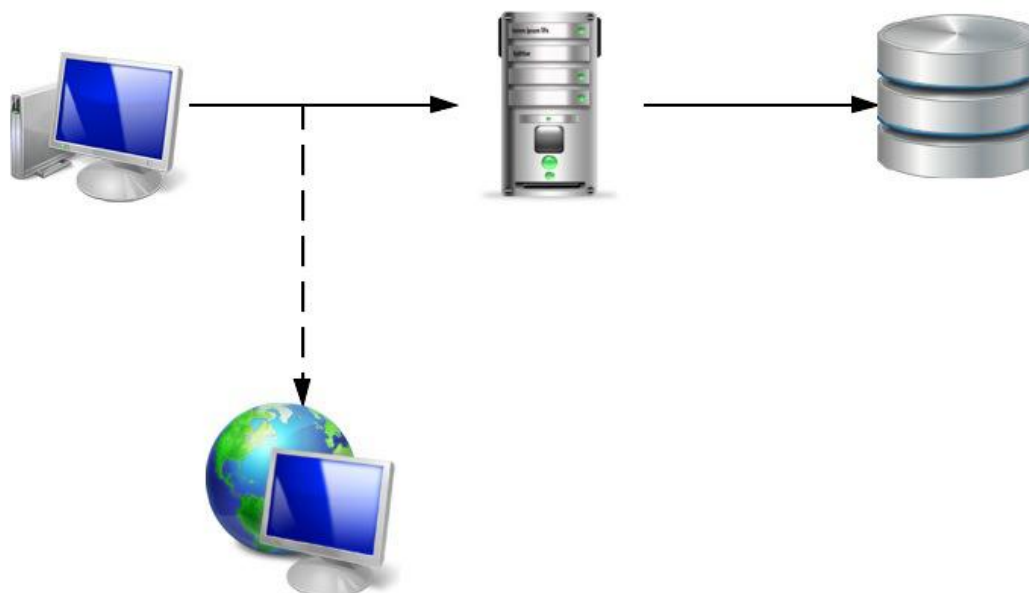
A área de Segurança é muito ampla, existindo vários aspectos que devem ser abordados. Aqui neste capítulo vamos focar em duas áreas de segurança: ataque para roubo de informações e controlar quem pode e o que pode acessar no nosso serviço. Quanto à ataques vamos abordar dois tipos principais que consistem em interceptação de requisição e podem ser evitados com a criptografia de dados. Já quanto ao controle de acesso vamos abordar os temas de autenticação e autorização.

Dois Tipos de ataques

Existem vários tipos e técnicas para se atacar um sistema online para roubo de informações. Vamos ver aqui dois tipos que resumem o conceito das técnicas utilizados pelos atacantes.

A primeira maneira de ataque é conhecida como *Eavesdropping*. Esta técnica consiste em um atacante ficar interceptando e ouvindo as requisições a um serviço.

Figura 5.1 - Esquema do Eavesdropping



Fonte: SAUDATE, 2017

O objetivo desta interceptação é muitas vezes para se ter conhecimento do conteúdo que está sendo trafegado na mensagem. Por exemplo, pode se saber valores que estão sendo enviados em uma transação bancária, ou até mesmo para roubo de informações de concorrentes, visto que hoje em dia se dá muito valor à dados que

possam trazer vantagens competitivas aos negócios. Também pode ter o objetivo de repetir a requisição contendo os mesmos dados, o que é conhecido como *replay attack*.

Outro tipo de ataque conhecido é o *man-in-the-middle*. Neste o atacante também intercepta a requisição de serviços, mas a diferença é que o objetivo é alterar o conteúdo da mensagem e reenviar para o servidor como se tivesse sido enviada pelo cliente verdadeiro. No exemplo de transações bancárias, o atacante poderia mudar o número da conta onde o dinheiro deveria ser depositado.

Agora, por que este tipo de ataque é possível e por que você deve se preocupar?

Primeiro estes ataques podem ocorrer por causa da arquitetura em que a Internet é montada. Existem muitos pontos de intersecção entre o cliente que está realizando uma requisição e o servidor que está expondo um recurso. Componentes de infraestrutura como *Firewalls*, *proxies*, *switches*, *etc*, são equipamentos sob responsabilidade de empresas cujo não temos domínio algum. Apesar de toda preocupação com segurança que elas possam ter, ainda assim não estão livres de serem vítimas e permitirem que estes tipos de ataques aconteçam. Uma pessoa mal intencionada pode invadir estes locais, alterar as configurações destes equipamentos e roubar informações valiosas.

Para você ter uma ideia do que estou falando utilize o comando *tracert* do Windows ou o *traceroute* do Linux para rastrear o caminho percorrido por uma requisição até chegar ao destino de um site, como os servidores do site google.com.

No Windows, abra o prompt de comando e digite:

```
tracert google.com
```

No Linux, abra o terminal e digite:

```
traceroute google.com
```

Cada ip que é listado ali refere-se a algum equipamento que está encaminhando sua requisição até chegar ao destino final.

Como proteger-se destes ataques

A proteção para estes ataques pode ser feita utilizando criptografia.

Como serviços RESTs são feitos sob o protocolo HTTP, o mecanismo que provê criptografia é o HTTPS, de *Hypertext Transfer Protocol over Secure Sockets Layer*, que aplica uma camada de segurança sobre o protocolo HTTP. Esta camada criptografa as requisições utilizando certificados digitais que auferem a autenticidade dos servidores onde os serviços estão expostos.

O uso de certificado informa ao cliente da autenticidade do servidor que está provendo aquele recurso, e também protege o conteúdo que está sendo trafegado com criptografia.

Este certificado pode ser emitido por uma entidade chamada de *Certification Authority* - CA, que é reconhecida por vários clientes que confiam nela. Por isso estas entidades tem o poder de certificar se uma empresa é ou não quem ela diz que é. Em suma é basicamente isso que o certificado autentica, já que qualquer um poderia criar um certificado se passando por outra empresa. Certificados emitidos por estas entidades são mais confiáveis.

É possível também gerar seus próprios certificados digitais, contudo, como estes não foram emitidos por uma entidade de confiança, eles não são considerados seguros pelos navegadores. Mas isso não quer dizer que o conteúdo da requisição não esteja protegido. Ainda assim é possível criptografar as requisições com certificados deste tipo. Os certificados gerados desta maneira recebem o nome de autoassinados.

A criptografia fornecida pelos certificados é baseada no mecanismo de chave pública e privada. A chave pública criptografa os dados, enquanto a privada descriptografa.

O Java vem com o JKS (*Java Key Store*), que é um mecanismo de armazenamento de certificados. Também é possível gerar estes certificados com a ferramenta keytool que vem junto com a JDK.

Certificado auto assinado

Para proteger as requisições ao nosso serviço com criptografia, vamos gerar um certificado auto assinado com o Keytool e configurar o servidor para utilizá-lo.

O Keytool é um utilitário de linha de comando que gerencia certificados que vem junto com a JDK. Com ele você pode administrar pares de chaves pública/privada e gerar seus próprios certificados autoassinados.

Na linha de comando do seu sistema operacional, navegue até o diretório *bin* de onde está instalado o JDK e digite o comando abaixo:

```
keytool -genkey -keyalg RSA -alias minha_chave -keystore server.keystore -storepass livraria -validity 360 -keysize 2048
```

O que cada parte deste comando significa?

-genkey: comando para gerar chaves.

-keyalg RSA: tipo de algoritmo utilizado na chave.

-alias minha_chave: alias da entrada no keystore que contém as chaves geradas.

-keystore server.keystore: o nome do banco de chaves que está sendo criado.

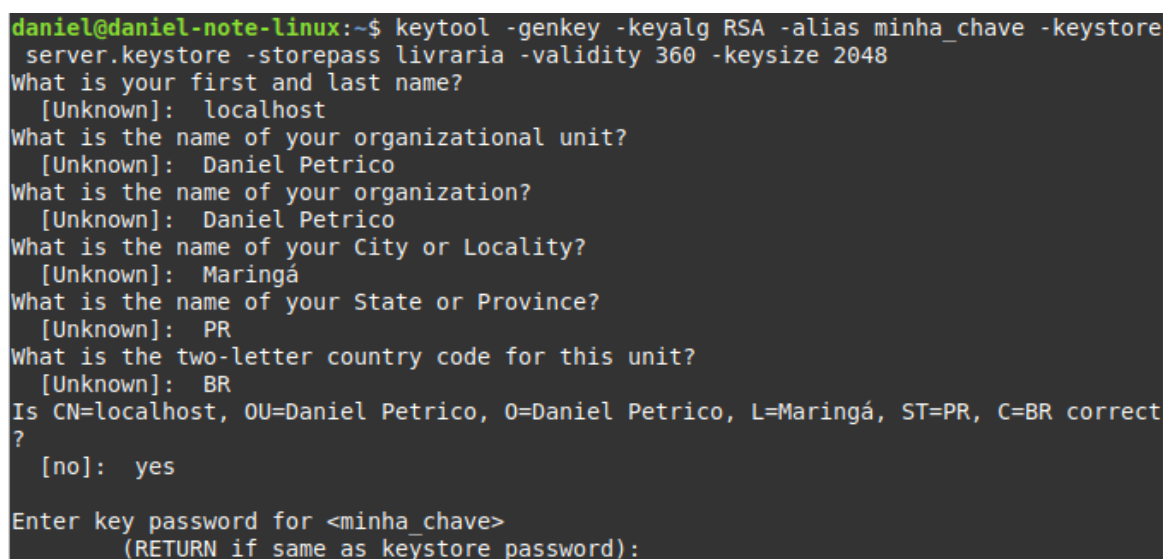
-storepass: senha do keystore.

-validity: validade do certificado em dias.

-keysize: tamanho da chave, neste caso 2048 bits.

Será apresentada uma série de perguntas referentes à organização:

Figura 5.2 - Resultado da geração do certificado com o keytool



```
daniel@daniel-note-linux:~$ keytool -genkey -keyalg RSA -alias minha_chave -keystore
server.keystore -storepass livraria -validity 360 -keysize 2048
What is your first and last name?
[Unknown]: localhost
What is the name of your organizational unit?
[Unknown]: Daniel Petrico
What is the name of your organization?
[Unknown]: Daniel Petrico
What is the name of your City or Locality?
[Unknown]: Maringá
What is the name of your State or Province?
[Unknown]: PR
What is the two-letter country code for this unit?
[Unknown]: BR
Is CN=localhost, OU=Daniel Petrico, O=Daniel Petrico, L=Maringá, ST=PR, C=BR correct?
[no]: yes
Enter key password for <minha_chave>
(RETURN if same as keystore password):
```

Fonte: O autor

A primeira pergunta refere-se ao domínio do site, no nosso exemplo vamos utilizar localhost. O arquivo server.keystore será gerado no diretório onde o comando foi executado. Para visualizar a entrada gerada no keystore digite o comando:

```
keytool -list -v -keystore server.keystore
```

Será solicitada a senha gerada no comando anterior. O resultado será algo como:

Figura 5.3 - Visualizando a entrada no do nosso certificado no keystore

```

daniel@daniel-note-linux:~$ keytool -list -v -keystore server.keystore
Enter keystore password:
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: minha_chave
Creation date: Aug 24, 2019
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Daniel Petrico, O=Daniel Petrico, L=Maringá, ST=PR, C=BR
Issuer: CN=localhost, OU=Daniel Petrico, O=Daniel Petrico, L=Maringá, ST=PR, C=BR
Serial number: 1b8a70d1
Valid from: Sat Aug 24 15:56:37 BRT 2019 until: Tue Aug 18 15:56:37 BRT 2020
Certificate fingerprints:
    MD5:  F8:D0:FA:30:7F:45:7C:AF:D3:2F:7E:B9:FC:51:7A:8F
    SHA1:  F1:56:1D:2C:8C:78:B2:4E:D3:4F:76:6F:A3:2B:6C:8C:EF:C2:4E:1D
    SHA256: 87:74:56:EF:DD:48:4E:39:47:5C:F8:D7:A8:62:3F:09:BC:A7:21:23:0C:7B:92:38:2C:AB:23:52:CD:87:EB:B0
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 56 E1 D0 EB CE C6 86 00  D2 86 07 32 8F 14 01 96  V.....2....
0010: 17 DE 2B FE                ..+.

```

Fonte: O autor

Implantando SSL no servidor

É preciso configurar o servidor que está rodando nosso serviço para utilizar SSL. Neste projeto estamos usando o Jetty, um servidor embarcado no maven. Este servidor possui um modo de configuração que possibilita ajustar seu funcionamento através de linhas de código. Uma das configurações possíveis é o uso de múltiplos conectores. Estes conectores ditam como o servidor deve responder às requisições feitas para o serviço. Neste caso, vamos fazer utilizar 2 conectores, um para responder às requisições feitas através de HTTP e outro para as requisições HTTPS.

Para fazer alteração, precisaremos alterar o arquivo pom.xml, adicionando mais uma dependência ao projeto:

```

<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-security</artifactId>
  <version>${jetty.version}</version>
</dependency>

```

Agora copie o arquivo server.keystore (que foi gerado pela ferramenta keytool anteriormente) para a raiz do projeto. Depois disso, vamos alterar o método main da classe Main.java para ficar como mostrado abaixo:

Figura 5.4 - Configurando servidor para usar HTTPS

```

public static void main(String[] args) throws Exception{
    File keystoreFile = new File("server.keystore");

    Server server = new Server();

    HttpConfiguration httpConfig = new HttpConfiguration();
    httpConfig.setSecureScheme("https");
    httpConfig.setSecurePort(8443);

    ServerConnector http = new ServerConnector(server,
        new HttpConnectionFactory(httpConfig));
    http.setPort(8080);

    SslContextFactory sslContextFactory = new SslContextFactory();
    sslContextFactory.setKeyStorePath(keystoreFile.getAbsolutePath());
    sslContextFactory.setKeyStorePassword("livraria");
    sslContextFactory.setKeyManagerPassword("livraria");

    HttpConfiguration httpsConfig = new HttpConfiguration(httpConfig);

    ServerConnector https = new ServerConnector(server,
        new SslConnectionFactory(sslContextFactory,
            HttpVersion.HTTP_1_1.asString()),
        new HttpConnectionFactory(httpsConfig));
    https.setPort(8443);

    server.setConnectors(new ServerConnector[] {http, https});

    final WebApplicationContext root = new WebApplicationContext();
    root.setContextPath("/livraria-virtual");
    root.setParentLoaderPriority(true);

    final String webappDirLocation = "src/main/webapp/";
    root.setDescriptor(webappDirLocation + "/WEB-INF/web.xml");
    root.setResourceBase(webappDirLocation);

    server.setHandler(root);
    server.start();
    server.join();
}

```

Fonte: O autor

Vamos analisar o que alteramos aqui.

Como queremos configurar SSL sobre o HTTP, será necessário informar onde está armazenado o certificado digital que criamos. Por isso criamos um objeto File para o arquivo server.keystore que copiamos a raiz do projeto.

Em seguida iniciamos a configuração do HTTP com *HttpConfiguration*, onde definimos o protocolo “https” de segurança e a porta de comunicação segura igual a 8443, que é a padrão para estes cenários.

Depois definimos as configurações para o protocolo HTTP por onde serão enviadas as requisições não seguras, mantendo a porta 8080 como padrão.

Em seguida criamos o objeto *SslContextFactory* informando os caminho e a senha do arquivo server.keystore.

Depois disso foi preciso criar uma configuração específica para o HTTPS. Aqui criamos o objeto *HttpConfiguration* passando no construtor um objeto criado anteriormente, isso faz com que o novo objeto tenha as mesmas informações do objeto anterior.

Criamos um segundo *Connector* passando a configuração do HTTPS.

Por fim passamos os dois conectores criados para o objeto server. O código restantes já havia sido criado anteriormente e refere-se a aplicação como um todo.

Agora basta acessar o serviço pela URI <https://localhost:8443/livraria-virtual/livro> e verificar pelo navegador que as requisições estão criptografadas, seguras. Não se preocupe se seu navegador mostrar alguma mensagem dizendo que a conexão não é segura, o que ele está tentando te informar é que o certificado desta conexão não foi emitido por uma CA, Autoridade Certificadora. Isso acontece por que nosso certificado é auto assinado.

Autenticação e Autorização

Ainda quando falamos em segurança de qualquer aplicação, dois fatores que temos que ter em mente é a autenticação e autorização.

A autenticação refere-se ao ato de determinar se a pessoa que está tentando utilizar o sistema é realmente quem diz quem ela é. Normalmente esse procedimento é realizado utilizando um nome de usuário e senha. Desta maneira, o sistema ou serviço pode certificar que quem quer utilizar o sistema pode ou não acessar o sistema ou serviço.

Já a autorização determina quais operações uma pessoa pode realizar dentro de sistema depois de já ter se autenticado. Por exemplo, caso ela deseje obter a lista de livros disponíveis, o sistema pesquisa as credenciais desta pessoa para ver se ela possui autorização para acessar determinado recurso.

A autenticação e autorização serve para proteger o sistema de forma que possibilite apenas pessoas autorizadas acessarem as informações contidas nele de forma controlada.

HTTP Basic

Na autenticação *basic*, em cada requisição, é informado o *header* HTTP *Authorization*, que conterá as credenciais da pessoa, nome de usuário e senha, codificado com o algoritmo Base 64 na seguinte estrutura:

Authorization: Basic <credenciais>

Caso deseje obter mais informações sobre o Base64 acesse <https://tools.ietf.org/html/rfc989>.

O fluxo de autenticação neste caso funciona assim: o cliente tenta acessar um recurso sem informar suas credenciais, então o servidor responde o código HTTP 401 - Unauthorized, indicando que o cliente não informou as credenciais necessárias para consumir aquele recurso. O cliente então precisa enviar uma requisição com o *header Authorization*, contendo as credenciais seguindo o formato descrito acima em Base 64. O servidor então verifica as credenciais, se estiverem corretas estará autenticado, ou seja, o usuário e senha fornecidos são conhecidos pelo servidor. O próximo passo executado pelo servidor é verificar se o cliente possui a autorização para consumir aquele recurso. Caso o cliente informe credenciais que não possua autorização para aquele recurso, será retornado um código HTTP 403 - Forbidden.

Para utilizar a autenticação Basic, devemos fazer algumas modificações no projeto. Novamente, as configurações serão feitas no método *main* da classe *Main.java*.

Figura 5.5 - Configurando Jetty para utilizar autenticação básica usuário e senha

```

HashLoginService loginService = new HashLoginService("MyRealm");
loginService.setConfig("myrealm.properties");

server.addBean(loginService);

ConstraintSecurityHandler security = new ConstraintSecurityHandler();
server.setHandler(security);

Constraint constraint = new Constraint();
constraint.setName("auth");
constraint.setAuthenticate(true);
constraint.setRoles(new String[] {"user", "admin"});

ConstraintMapping mapping = new ConstraintMapping();
mapping.setPathSpec("/*");
mapping.setConstraint(constraint);

security.setConstraintMappings(Arrays.asList(mapping));
security.setAuthenticator(new BasicAuthenticator());
security.setLoginService(loginService);

security.setHandler(root);

server.start();
server.join();
}

```

Fonte: O autor

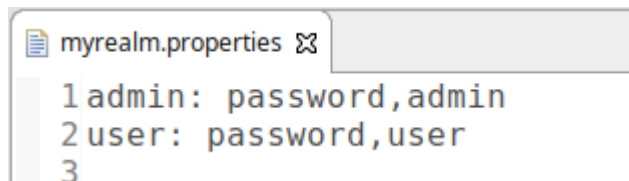
Vamos analisar as alterações que foram feitas.

Primeiro instanciamos um objeto do tipo *HashLoginService*. Esta é uma implementação que armazena dados de usuário, como login, senha e papéis, em memória. Ele serve para propósitos de testes. Os dados obtidos por esta classe estão armazenados em um arquivo de propriedades, o *myrealm.properties*. O formato esperado deste arquivo é:

login: senha [, papéis,...]

Primeiro vem o nome do usuário, seguido da senha dele e seus papéis separados por vírgula. Este arquivo deve ser criado e colocado na raiz do projeto. A figura 5.6 mostra o conteúdo do arquivo. nele pode ser visto que foram criados dois usuários, *admin* e *user*, ambos com a senha *password*, sendo o admin com o papel *admin* e user com o papel *user*.

Figura 5.6 -Conteúdo do arquivo *myrealm.properties*

A screenshot of a text editor window titled 'myrealm.properties'. The editor contains three lines of text: '1 admin: password,admin', '2 user: password,user', and '3'.

```
myrealm.properties
1 admin: password,admin
2 user: password,user
3
```

Fonte: O autor

Além da classe `HashLoginService`, existem outras implementações de `LoginService`, como a `JDBLoginService`. Esta classe obtém os dados de usuários a partir de um banco de dados, sendo ela mais indicada para uso em ambiente de produção.

Seguindo a configuração do Jetty para autenticação básica, criamos uma restrição através do objeto `Constraint`. Esta restrição diz que o usuário deve ser autenticado (`setAuthenticate(true)`) e definimos dois papéis (roles) para os usuários, `user` e `admin`. Lembra do arquivo `myrealm.properties`? Estas roles são as mesmas definidas neste arquivo. Esta restrição será utilizada mais adiante no objeto `ConstraintSecurityHandler`.

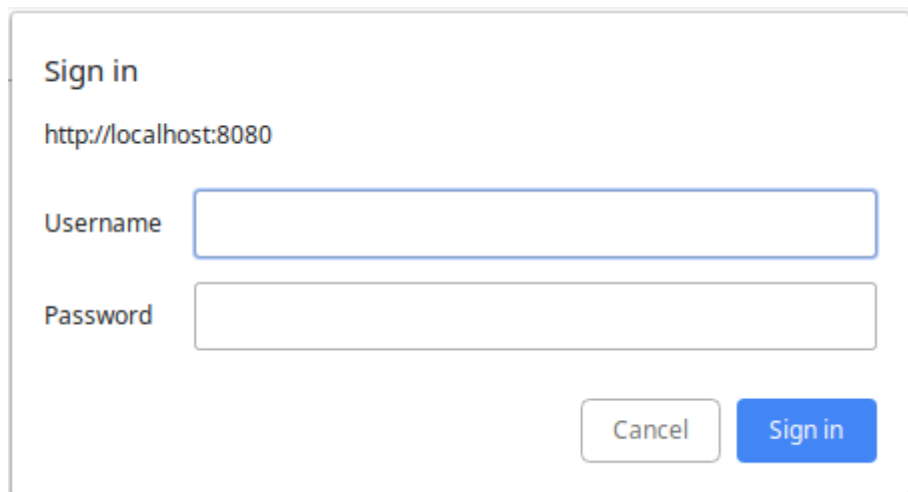
Depois é hora de definir quais URI's serão seguradas, ou seja, quais URI's do nosso serviço necessitarão de autenticação de usuário para que possa ser acessada. Esta configuração é feita através do objeto `ConstraintMapping` e aqui definimos que todas URI's do nosso serviço ("/") serão seguradas.

Os objetos `Constraint` e `ConstraintMapping` trabalham em conjunto, permitindo que seja possível realizar uma configuração mais refinada de segurança, como por exemplo, definindo que certas URI's possam ser acessadas por um role, `admin` por exemplo, e outras URI's por outra role, `user` por exemplo.

Depois instanciamos um objeto `ConstraintSecurityHandler`, que é um manipulador responsável em garantir as restrições de segurança. Neste objeto é atribuído as restrições de usuário e mapeamento de URI criados anteriormente. Também atribuímos o `loginService`, para que ele possa obter os dados do usuário. Desta maneira, o `constraintSecurityHandler` conseguirá determinar que se alguém está tentando acessar uma URI protegida sem antes se autenticar e, neste caso, solicitar que informe as credenciais, usuário e senha. Quando for informado estes dados, este objeto poderá certificar que se trata de um usuário válido ou não, comparando os dados fornecidos com os que estão gravados no arquivo `myrealm.properties`.

Entendido isso, podemos iniciar o servidor novamente e realizar alguns testes. Caso você tente acessar alguma operação do tipo GET utilizando algum navegador, uma tela como a mostrada na figura 5.7 será exibida.

Figura 5.7 - Janela de autenticação exibida pelo navegador Google Chrome

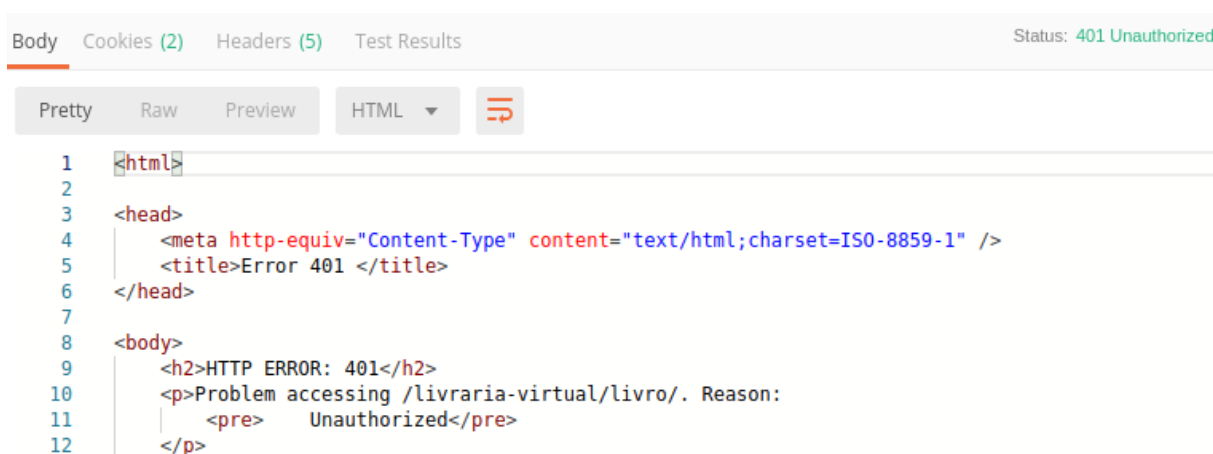


Fonte: O autor

Aqui basta informar as credenciais de qualquer um dos usuários, user ou admin, que o resultado será retornado normalmente.

E como fica para utilizar o serviço a partir do Postman? Caso você tente invocar algum serviço criado anteriormente, obterá como resposta o erro 401:

Figura 5.8 - Erro 401 no Postman



Fonte: O autor

Para poder usar o serviço protegido, é preciso informar no cabeçalho da requisição os dados de usuário e senha. O Postman permite que façamos isso de maneira muito fácil. Clique na aba *Authorization* e no campo de seleção *Type* escolha *Basic Auth*.

Figura 5.9 - Basic Auth no Postman

The screenshot shows the Postman interface with the 'Authorization' tab selected. At the top, the method is 'GET' and the URL is 'http://localhost:8080/livraria-virtual/livro/'. The 'Authorization' tab is active, showing a 'TYPE' dropdown set to 'Basic Auth'. Below this, a message states: 'The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)'. A 'Preview Request' button is visible. On the right, there are input fields for 'Username' (containing 'user') and 'Password' (containing masked characters '*****'). A checkbox labeled 'Show Password' is currently unchecked. A warning message at the top right of the right pane says: 'Heads up! These parameters hold sensitive data. To keep this data secure while environment, we recommend using variables. [Learn more about variables](#)'.

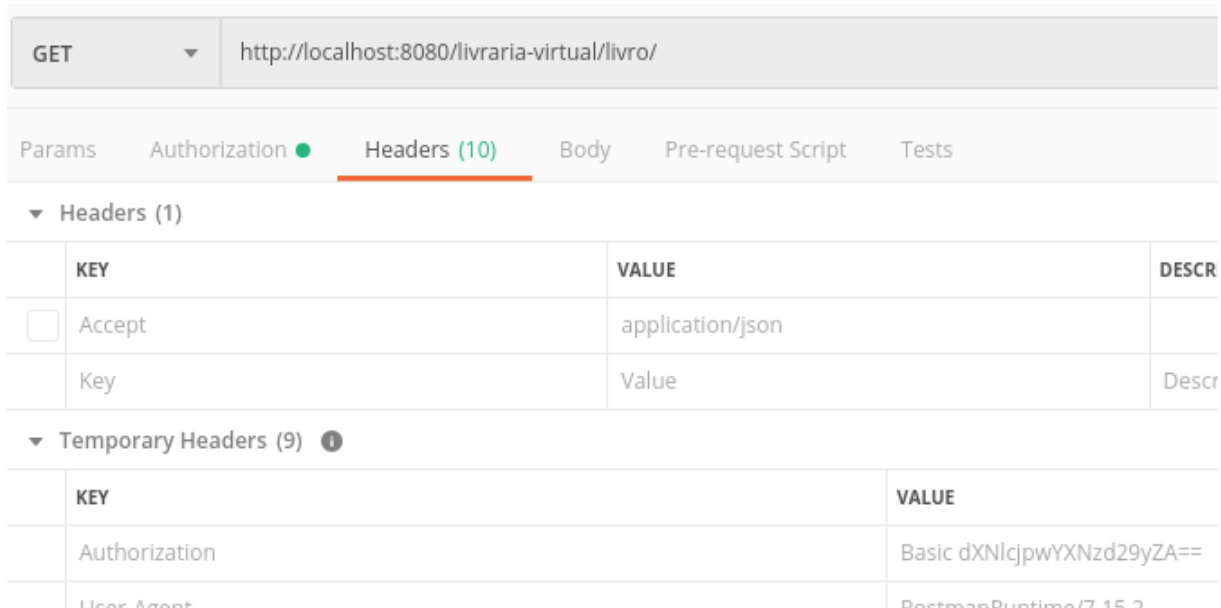
Fonte: O autor

Informe os dados de usuário e senha válidos e faça uma requisição. Os dados deverão ser retornados normalmente.

Agora tente enviar dados de usuários incorretos e veja o que acontece.

O Postman facilita muito o envio dos dados de usuário na requisição pela sua interface gráfica. Mas lembre-se que este dado deveria ser enviado no cabeçalho criptografado em Base64. O que aconteceu então que podemos fazer a requisição sem problemas. Acontece que por trás dos panos o Postman criptografa estes dados em Base64 e adiciona no header automaticamente. Para comprovar, veja os dados contidos na aba *Headers*:

Figura 5.10 - Header Authorization no Postman



Fonte: O autor

Perceba que foi adicionado a chave Authorization no cabeçalho contendo uma String criptografada conforme explicado anteriormente.

Consumindo serviço com autenticação

O que aconteceu com o cliente que desenvolvemos anteriormente que consumia dados deste serviço? Lembre-se que antes não havia a necessidade de autenticação para utilizá-lo. Se você tentar usar o cliente agora receberá o seguinte erro:

Figura 5.11 - Erro 401 no cliente

```
Exception in thread "main" javax.ws.rs.NotAuthorizedException: HTTP 401 Unauthorized
    at org.glassfish.jersey.client.JerseyInvocation.convertToException(JerseyInvocation.java:1056)
    at org.glassfish.jersey.client.JerseyInvocation.translate(JerseyInvocation.java:859)
    at org.glassfish.jersey.client.JerseyInvocation.lambda$invoke$1(JerseyInvocation.java:743)
    at org.glassfish.jersey.internal.Errors.process(Errors.java:292)
    at org.glassfish.jersey.internal.Errors.process(Errors.java:274)
    at org.glassfish.jersey.internal.Errors.process(Errors.java:205)
    at org.glassfish.jersey.process.internal.RequestScope.runInScope(RequestScope.java:390)
    at org.glassfish.jersey.client.JerseyInvocation.invoke(JerseyInvocation.java:741)
    at org.glassfish.jersey.client.JerseyInvocation$Builder.method(JerseyInvocation.java:404)
    at org.glassfish.jersey.client.JerseyInvocation$Builder.get(JerseyInvocation.java:300)
    at Main.main(Main.java:18)
```

Fonte: O autor

HTTP 401 Unauthorized soa familiar não? Isso acontece por que não nosso cliente não está preparado para informar os dados de usuário e senha. Vamos adequar isso adicionando autenticação no cliente fornecida pelo Jersey.

Figura 5.12 - Ajuste no cliente para informar credenciais

```

Client client = ClientBuilder.newClient();

HttpAuthenticationFeature auth =
    HttpAuthenticationFeature.basic("admin", "password");

client.register(auth);

Livros livros = client
    .target("http://localhost:8080/livraria-virtual/")
    .path("livro")
    .request(MediaType.APPLICATION_XML)
    .get(Livros.class);

```

Fonte: O autor

Pronto! O que fizemos aqui foi adicionar um objeto responsável em carregar as credenciais do usuário na requisição. Agora o cliente está funcionando normalmente.

Consumindo serviço com SSL

E agora, se tentarmos consumir o mesmo serviço utilizando a porta 8443, que possui a camada de criptografia SSL?

Figura 5.13 - Consumindo serviço com SSL

```

Livros livros = client
    .target("https://localhost:8443/livraria-virtual/")
    .path("livro")
    .request(MediaType.APPLICATION_XML)
    .get(Livros.class);

```

Fonte: O autor

Primeiro mudamos a url para protocolo HTTPS e porta 8443, e o que obtemos ao executar o cliente é o erro:

```

Caused by: sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
    at
    sun.security.provider.certpath.SunCertPathBuilder.build(SunCertPathBuilder.
    java:141)
    at
    sun.security.provider.certpath.SunCertPathBuilder.engineBuild(SunCertPathBu
    ilder.java:126)
    at java.security.cert.CertPathBuilder.build(CertPathBuilder.java:280)
    at
    sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:392)
    ... 31 more

```

Isso acontece por que não temos no cliente um *trust store* registrado, que é responsável em verificar a autenticidade do servidor. Para resolver este problema, vamos registrar o *trust store* utilizando a classe *SSLConfigurator* do Jersey.

Primeiro copie o arquivo `server.keystore` do projeto servidor para a raiz do projeto cliente. Depois altere o método `main` da classe `Main.java` do cliente:

Figura 5.14 - Configurando SSL no cliente

```
public static void main(String[] args) {
    SslConfigurator config = SslConfigurator
        .newInstance()
        .trustStoreFile("server.keystore")
        .trustStorePassword("livraria");

    SSLContext context = config.createSSLContext();

    Client client = ClientBuilder.newBuilder()
        .sslContext(context).build();
}
```

Fonte: O autor

Pronto! Basta executar a aplicação novamente para ver os livros retornados.

O que foi feito aqui?

A classe `SslConfigurator` serve para configurar os dados da base de certificados digitais. Neste caso estamos obtendo a mesma base gerada anteriormente com o comando `keytool`. Ela contém a chave do certificado auto assinado por nós. Para acessar esta base é preciso a senha, por isso a informamos através do método `trustStorePassword("livraria")`.

Depois disso só precisamos criar um `SSLContext` a partir do objeto `config`, e passar este contexto para o objeto `client`.

Com isso o cliente já pode consumir um serviço que usa segurança SSL e autenticação básica.

Considerações Finais

Neste capítulo abordamos alguns riscos envolvidos ao expor serviços REST na Internet. Um deles é o *Man-in-the-middle*, que consiste em um ataque onde alguém intercepta uma requisição, altera os dados e continua realizando a requisição com os dados alterados como se fosse o cliente original. O outro é o *Eavesdropping*.

Vimos que para se proteger destes tipos de ataque podemos configurar nossos serviços para utilizar uma camada de segurança sobre o protocolo HTTP, que é o Security Service Layer, ou SSL. Fizemos isso utilizando um certificado digital auto assinado, onde já é possível realizar requisições com criptografia.

Outra preocupação que devemos ter é quanto a quem está utilizando nossos serviços. Devemos certificar que quem está fazendo alguma requisição possui autorização para acessar determinado recurso. Pudemos fazer essa proteção utilizando a autenticação básica provida pelo protocolo HTTP.

Referências:

BUCEK, Pavel; GEERTSEN, Santiago Pericas. JAX-RS: Java API for RESTful Web Services.{online}. Disponível na Internet via <https://github.com/jax-rs/spec/blob/master/spec.pdf>. Arquivo capturado em 21 de jul. 2019

FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

RICHARDSON, Leonard; RUBY, Sam. RESTful Web Services. Sebastopol: O'Reilly Media, Inc., 2007 . 440 (Web Services for the Real World).

SAUDATE, Alexandre. REST: Construa API's inteligentes de maneira simples. São Paulo: Casa do Código, 2017 . 332p (Livros para o programador).

SAUDATE, Alexandre. SOA aplicado: integrando com web services e além. São Paulo: Casa do Código, 2013 . 287 (Livros para o programador).