Web Services REST versus SOAP

Aprenda as diferenças e vantagens ao adotar cada uma dessas tecnologias em seu projeto Java.

Fique por dentro

Este artigo abordará as principais tecnologias para criação de web services: REST e SOAP, explicando o conceito por trás de cada uma delas e como aplicá-las em um projeto Java.

O assunto em análise será de bastante interesse para desenvolvedores que não possuem domínio sobre web services e que desejam conhecer estas opções, assim como para desenvolvedores mais experientes, que buscam compreender as diferenças e vantagens de cada uma dessas abordagens e quando adotá-las.

A comunicação entre sistemas e a capacidade de expor serviços através da Internet se tornaram uma necessidade comum para a grande maioria dos sistemas corporativos.

Seja apenas para prover suporte a outros módulos na mesma rede privada ou para permitir o acesso público a um determinado serviço, os web services são uma das tecnologias mais utilizadas tanto no Java como em outras linguagens de grande porte e fazem parte do dia a dia dos desenvolvedores.

Entre as abordagens existentes para a implementação de web services, o protocolo SOAP e o REST são as opções de maior destaque nos dias de hoje, estando presentes em grande parte das discussões relacionadas a arquiteturas orientadas a serviços na web.

Ganhando destaque no mercado do início da década de 2000, o protocolo SOAP teve grande importância em 2003, quando passou a ser uma recomendação da W3C para desenvolvimento de serviços web, sendo o padrão mais implementado na época e deixando um legado de sistemas e integrações que perdura até hoje.

O REST, por sua vez, foi desenvolvido juntamente com o protocolo HTTP 1.1 e, ao contrário do SOAP, que tem como objetivo estabelecer um protocolo para comunicação de objetos e serviços, propôs algumas ideias de como utilizar corretamente os verbos HTTP (GET, POST, PUT, HEAD, OPTIONS e DELETE) para criar serviços que poderiam ser acessados por qualquer tipo de sistema.

Sua concepção, portanto, não era de um protocolo, mas sim de um Design Pattern arquitetural para serviços expostos numa rede, como a internet, através do protocolo HTTP.

Para introduzir essas duas tecnologias, nesse artigo apresentaremos o funcionamento de ambos os métodos, explicando as principais características, vantagens e desvantagens de cada um deles e também demonstraremos como implementá-los em um projeto Java.

Além disso, apresentaremos também as boas práticas de desenvolvimento de cada uma dessas tecnologias e introduziremos situações reais em que essas soluções podem ser aplicadas, através de um exemplo a ser desenvolvido no final do artigo.

Com isso, seremos capazes de diferenciar claramente ambas as abordagens e teremos o conhecimento necessário para entender em quais cenários cada uma delas se aplica, permitindo, assim, que o leitor tenha um arsenal maior para o desenvolvimento e consumo de serviços dos mais variados formatos.

Uma breve introdução sobre web services

Web services são, por definição, serviços expostos em uma rede que permitem a comunicação entre um ou mais dispositivos eletrônicos, sendo capazes de enviar e processar dados de acordo com sua funcionalidade.

Essa comunicação, por sua vez, segue alguns protocolos, principalmente relacionados ao formato da transmissão de dados, o que permite que, uma vez que um sistema implemente essas regras, qualquer outro sistema que siga o mesmo protocolo seja capaz de se comunicar com ele.

Devido a isso, os web services se tornaram extremamente populares, pois acabaram por permitir a comunicação entre plataformas completamente diferentes (Java, C#, C++, Ruby) sem grandes esforços.

Entre essas padronizações estipuladas, as duas de maior destaque são o protocolo SOAP e o modelo de design REST, as quais iremos discutir nos próximos tópicos.

Protocolo SOAP

O protocolo SOAP, abreviação para *Simple Object Access Protocol*, é uma especificação para a troca de informação entre sistemas, ou seja, uma especificação de formato de dados para envio de estruturas de dados entre serviços, com um padrão para permitir a interoperabilidade entre eles.

Seu design parte do princípio da utilização de XMLs para a transferência de objetos entre aplicações, e a utilização, como transporte, do protocolo de rede HTTP.

Os XMLs especificados pelo SOAP seguem um padrão definido dentro do protocolo. Esse padrão serve para que, a partir de um objeto, seja possível serializar o mesmo para XML e, também, deserializá-lo de volta para o formato original.

Além do formato dos objetos, nesse protocolo também são definidos os padrões que os serviços SOAP devem seguir, ou seja, a especificação dos endpoints que as implementações de SOAP devem ter.

A serialização de objetos, que acabamos de descrever, tem como objetivo formar uma mensagem SOAP, composta pelo objeto denominado envelope SOAP, ou **SOAP-ENV**.

Dentro desse envelope existem mais dois componentes: o **header SOAP**, que possui informações de atributos de metadados da requisição como, por exemplo, IP de origem e autenticação; e o **SOAP body**, que possui as informações referentes à requisição, como o nome dos métodos que deseja se invocar e o objeto serializado que será enviado como payload da requisição.

Na **Listagem 1** apresentamos um exemplo de um envelope SOAP simplificado representando uma requisição SOAP. Nele podemos identificar claramente os elementos **body** e **header** e também ter uma breve visão da sintaxe do SOAP. Dentro do **body** ainda se nota a descrição do método para o qual esse envelope deve ser direcionado (GetRevista) e o payload específico que foi enviado a essa função.

Listagem 1. Exemplo de um envelope SOAP que transporta um objeto com o nome "RevistaNome".

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetRevista xmlns:m="http://www.example.org/revista">
      <m:RevistaNome>Java Magazine</m:ResvistaNome>
    </m:GetRevista>
  </soap:Body>
</soap:Envelope>
```

Como verificado, não existe muito segredo no transporte de objetos através do SOAP. No caso do nosso exemplo, a requisição ao método **GetRevista** é enviada para o servidor juntamente com o objeto **RevistaNome**, que servirá de parâmetro de entrada ao nosso método e tem, como conteúdo, a **String** "Java Magazine". Ao chegar ao servidor, essa informação é parseada e a aplicação realiza a lógica necessária para processá-la.

Como boa prática, esse serviço também tem seu funcionamento e formato de dados de entrada especificados pelo protocolo SOAP. Essa definição de como os métodos que compõem um web service SOAP devem funcionar é feita através de um documento XML chamado WSDL, abreviação de *Web Service Description Language*, um XML que descreve o formato dos dados que o serviço aceita, seu funcionamento e comportamento (se é assíncrono ou não, por exemplo) e os dados de saída do método, incluindo desde o formato até o envelope SOAP que deve ser retornado.

Na **Listagem 2** apresentamos um exemplo de WSDL especificando nosso serviço **GetRevista**, que aceita uma **String** como entrada e devolve, como retorno, outra **String**.

Listagem 2. Exemplo de WSDL para o serviço getRevista.

```
<definitions name="HelloService"
     targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

     <message name="GetRevistaRequest">
        <part name="RevistaNome" type="xsd:string"/>
     </message>
     <message name="GetRevistaResponse">
        <part name="RevistaNome" type="xsd:string"/>
     </message>

     <portType name="Revista_PortType">
        <operation name="GetRevista">
           <input message="tns:GetRevistaRequest "/>
           <output message="tns:GetRevistaResponse "/>
        </operation>
     </portType>

     <binding name="Revista_Binding" type="tns:Revista_PortType">
     <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="GetRevista">
        <soap:operation soapAction="GetRevista"/>
        <input>
           <soap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              use="encoded"/>
        </input>
        <output>
           <soap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              use="encoded"/>
        </output>
     </operation>
     </binding>

     <service name="GetRevista">
        <documentation>WSDL File for HelloService</documentation>
        <port binding="tns:Revista_Binding" name="Revista_Port">
           <soap:address
              location="http://www.examplo.com/Revista/">
        </port>
     </service>
  </definitions>
```

Conforme podemos verificar, o WSDL apresenta diversos atributos referentes ao serviço, como a localização do endpoint, nomes de cada um dos serviços, nomes de cada parâmetro e seu respectivo tipo.

Através dessa definição, qualquer sistema externo pode se comunicar com o web service que criamos no WSDL, enviando a requisição através do protocolo SOAP.

Entre os elementos do WSDL que apresentamos na **Listagem 2**, podemos observar que a definição de um serviço é basicamente composta por quatro partes principais.

A primeira delas, definida pela tag **<message>**, é onde definimos o formato de nossos dados, podendo ser esses de entrada e saída. Nesse ponto, podemos utilizar tanto atributos primitivos, como no exemplo o uso de **String**, como também utilizar schemas XMLs (chamados de XSDs) para definir nossos objetos.

Uma vez definido o formato de nossas informações, a segunda parte de um WSDL é especificar as operações que desejamos expor e suas respectivas entradas e saídas. Isso é feito na tag **<portType>**, onde definimos a operação **GetRevista** e determinamos, também, os dados de entrada e saída, baseados nas tags **<message>** que criamos anteriormente.

Em seguida, especificamos a tag do WSDL chamada de **<binding>** e, dentro desta, a nossa tag **portType** (criada no passo anterior) e suas respectivas operações a um serviço. Esse serviço, que representará a implementação de nosso web service, é determinado na tag **<service>**, onde explicitamos o endereço do endpoint através da tag **<soap:address location>**.

Além de tudo isso, o SOAP também possibilita funcionalidades bastante interessantes, como é o caso do WS-Addressing.

Esse tipo de tecnologia permite que, uma vez enviada alguma informação dentro de um envelope SOAP, seja possível o envio de um parâmetro extra definindo um endpoint de callback para ser chamado logo que a requisição termine de ser processada.

Dessa forma, uma vez que termine a requisição, o web service é capaz de chamar essa URL de callback, passando em seu conteúdo o resultado do processamento e permitindo, dessa forma, o trabalho assíncrono entre sistemas.

Modelo arquitetural REST

Ao introduzir e adotar essas padronizações, muitos sistemas começaram a implementar esse modelo na criação de seus web services, fazendo com que o protocolo SOAP se tornasse uma das tecnologias essenciais no desenvolvimento de web services em sistemas corporativos.

No entanto, o protocolo SOAP trazia também diversas desvantagens. A primeira delas (e talvez uma das mais importantes) é o fato de que ao transportar todas as informações dentro de um envelope SOAP em XML, o conteúdo dos dados enviados de um sistema para outro se torna muitas vezes maior que o necessário, elevando tanto o consumo da banda de rede como o tempo de processamento dos dados.

Em segundo lugar, a utilização de web services SOAP não faz o uso correto dos verbos HTTP. Isso se deve ao fato de que todas as requisições SOAP são feitas através do POST de um XML, que contém o envelope SOAP.

Porém, em grande parte das requisições, o mais adequado seria outro verbo, como um GET ou PUT, de acordo com a funcionalidade exposta pelo serviço. Graças a essa limitação do protocolo, o SOAP acaba por contrariar alguns dos princípios de uma boa modelagem HTTP, ferindo as boas práticas dessa especificação.

Por esses e outros motivos, desenvolvedores e pesquisadores vêm adotando outra abordagem para criar seus web services, chamada de serviços REST.

A modelagem por trás de um serviço REST parte do princípio de seguir as boas práticas da criação de serviços HTTP e utilizar esses padrões para desenvolver web services simples e performáticos.

Um dos pontos cruciais nesse modelo arquitetural é o uso correto dos métodos disponibilizados pelo HTTP. Ao contrário do SOAP, que só utiliza o método POST para transmitir os dados, uma arquitetura REST prevê que, dentro de um cenário ideal, o método HTTP a ser utilizado seja diretamente relacionado à funcionalidade do serviço a ser consumido.

Portanto, serviços de busca de informações são feitos através de métodos **GET**, serviços de atualização de informação através de métodos **PUT**, serviços de criação de dados através do método **POST**, serviços de deleção através do **DELETE** e assim por diante. A partir do correto uso dos verbos HTTP ganhamos também a vantagem de não termos diversas URLs para cada um dos nossos serviços, podendo expor somente uma URL e, conforme o método, executar uma ação diferente.

Como exemplo, vamos imaginar um simples serviço CRUD (Criação, atualização, leitura e deleção de dados) para clientes. Dentro de uma arquitetura REST, esse serviço pode ser definido somente com uma URL, por exemplo: *http://nosso.exemplo.com/client/12345*. Para realizar uma consulta ao cliente *12345*, basta realizarmos uma requisição **GET** à URL mencionada e para deletá-lo, realizar uma requisição **DELETE**.

O mesmo pode se aplicar às requisições **PUT** e **POST**, onde utilizamos o corpo da requisição que nos foi enviado para atualizar ou criar uma nova entidade em nosso sistema, respectivamente.

Deste modo, é fácil observar que, utilizando os métodos corretamente, os web services REST se tornam bastante simples de serem desenvolvidos e altamente intuitivos, sem a necessidade de descrições, como os WSDLs ou documentações extensas, para explicar sua funcionalidade.

Outra vantagem do REST em relação ao SOAP se refere ao formato do dado a ser trafegado.

Enquanto o SOAP se restringe a utilizar XMLs, o REST não impõe restrições quanto a isso, ou seja, qualquer um pode ser enviado em suas requisições.

Diretamente relacionado a isso, como boa prática, uma requisição REST deve sempre conter o header **Content/Type** para explicitar o formato do dado em sua requisição e, para o retorno, o header **Accepts**, determinando o formato de dado que a aplicação espera.

Como exemplo, caso uma aplicação deseje enviar um dado em XML e receber, como retorno, um JSON, basta ela explicitar ambos os formatos nos respectivos headers e o web service REST deve ser capaz de resolver a requisição de acordo com essas necessidades.

Dentro do REST, outra diferença é o chamado HATEOAS, abreviação para *Hypermedia as the Engine of Application Status*. Considerado um dos principais benefícios dessa arquitetura, o HATEOAS define que, dentro de uma aplicação REST, o estado da aplicação e da interação do usuário deve ser controlado através das URLs (Hypermedia) retornadas pelo serviço. Isso implica que, em um serviço REST, quando a primeira URL dessa aplicação é acessada, todos os possíveis "caminhos" que o cliente pode seguir a partir dali são retornados dentro da resposta da requisição, como as URLs dos serviços que podem ser invocados no passo seguinte.

Através dessa técnica evitamos diversos problemas em nossos clientes, uma vez que eles só precisam ter a primeira URL de nossa aplicação hardcoded em seu código, possibilitando que as URLs seguintes sejam capturadas durante a própria execução do web service.

Aplicações RESTful

As características apresentadas anteriormente são somente algumas das principais definições do modelo REST. Podemos dizer que, uma aplicação que segue todas as definições e funcionalidades desse modelo, é chamada de RESTful. Essas especificações arquiteturais, denominadas de **Architectural Constraints**, são divididas em cinco grupos principais, cada um referenciando as características mais importantes do modelo REST.

A primeira funcionalidade desse grupo exige que um sistema RESTful siga o modelo Cliente-Servidor. Nesse modelo, as responsabilidades de cada uma das partes ficam claramente divididas, deixando todo ou grande parte do código do lado do servidor, que expõe essas funcionalidades através de uma interface uniforme ao cliente (web services) e, consequentemente, minimiza problemas na aplicação provenientes de código no cliente.

Em segundo lugar, serviços RESTful devem ser stateless, ou seja, não devem guardar estado. Isso explicita que, dada uma requisição, todas as informações necessárias para o processamento desta devem estar presentes no payload enviado ao serviço.

Também como uma necessidade importante, é essencial que os serviços REST sejam cacheados, ou seja, o cliente deve ser capaz de acessar um cache das respostas, evitando que sejam feitas requisições seguidas a recursos que não sofreram alteração e permitindo, portanto, um menor consumo de banda. Além disso, quando um desses recursos for atualizado, esse cache deve ser capaz de automaticamente notar isso e responder a requisição atualizada ao cliente.

Outro ponto importante para o cliente é que os serviços REST devem poder ser separados em camadas de forma transparente ao usuário. Isso implica que, se quisermos colocar um serviço intermediário ou um Load Balancer entre o serviço final e o consumidor desse sistema, não deve haver grandes alterações em ambos os lados, sendo essa transição o mais transparente possível.

Por fim, um sistema RESTful deve ser capaz de prover uma interface uniforme através de endpoints HTTP para cada um dos recursos disponíveis possibilitando, também, a implementação dos itens que mencionamos anteriormente, incluindo o HATEOAS, uso correto das URLs e dos métodos HTTP e a capacidade de descrever o tipo de dado através de headers (**Content/Type** e **Accept**).

Criando os nossos web services em Java

Agora que entendemos os principais conceitos por trás desses dois modelos de web service, iremos apresentar como criá-los dentro de um projeto Java. Para isso, vamos introduzir duas bibliotecas bastante importantes: a JAX-WS, para projetos SOAP; e a JAX-RS, para projetos REST.

Configurando nosso projeto

O primeiro passo é realizar o download de um servidor que implementa as bibliotecas da Java EE 6, permitindo assim o uso das funcionalidades do JAX-WS e do JAX-RS. Como escolha para esse artigo, decidimos adotar o servidor WildFly, considerado o sucessor do famoso JBoss AS (veja o endereço para download na seção **Links**).

Feito o download do WildFly, para realizar a instalação basta descompactá-lo em alguma pasta de seu computador e, em sua IDE, criar a configuração de um novo servidor e identificar, dentro das propriedades, o caminho do diretório em que os arquivos do mesmo foram descompactados.

Neste exemplo, iremos utilizar a IDE Eclipse Kepler para desenvolver nossos serviços e o plugin do JBoss Tools para configurar nosso servidor.

> **Nota**: A instalação e configuração do plugin do JBoss, juntamente com a instalação do servido r na IDE, podem ser visualizadas com mais detalhes no endereço indicado na seção **Links**.

Configurado o servidor, podemos agora criar o projeto na IDE. Para isso, basta clicarmos em *New > Dynamic Web Project*, nomearmos nosso projeto e escolher, como runtime de nossa aplicação, o servidor recém-configurado, o WildFly 8. Na **Figura 1** mostramos essa configuração.

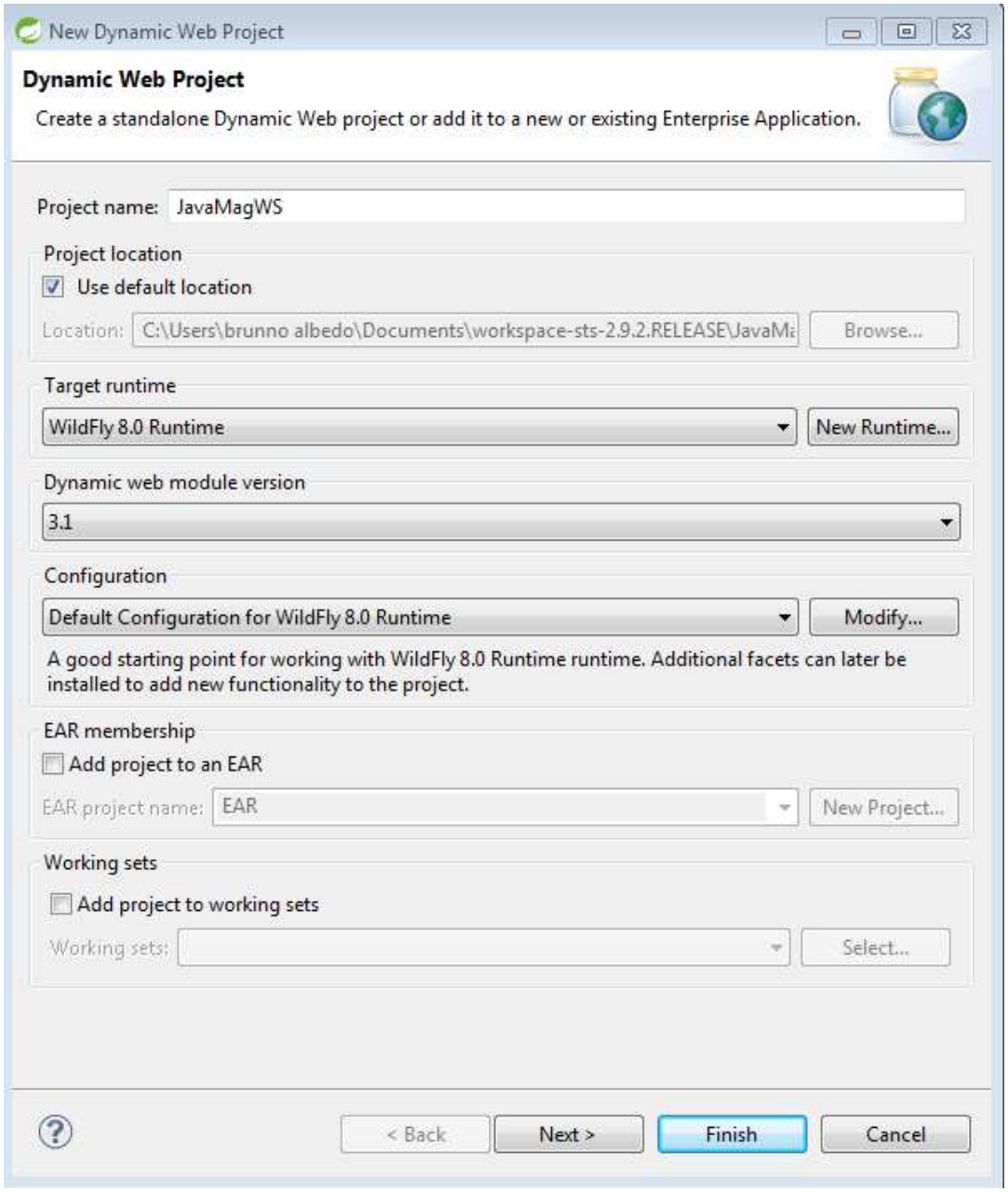

Figura 1. Configuração de nosso projeto.

Feito isso, precisamos configurar o projeto como sendo do tipo Maven. Essa etapa pode ser feita com a ajuda do plugin do Maven (que pode ser baixado pelo próprio eclipse), bastando adicionar o *pom.xml* apresentado na **Listagem 3** na raiz do projeto e clicar com o botão direito no projeto e escolher *Configure* > *Convert to Maven Project*.

Listagem 3. Pom.xml a ser adicionado ao projeto.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>com.java.mag</groupId>
  <artifactId>WS</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>3.0.6.Final</version>
    </dependency>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jackson-provider</artifactId>
        <version>3.0.6.Final</version>
    </dependency>
  </dependencies>
</project>
```

Como pode ser visto no *pom.xml*, basicamente definimos duas dependências ao nosso projeto: a dependência relacionada ao **resteasy-jaxrs** e a dependência relacionada ao **resteasy-jackson**. Essas bibliotecas servem, respectivamente, para implementar o redirecionamento das requisições HTTP aos respectivos serviços REST, e, também, para permitir o processamento de objetos em formato JSON.

Criando nosso serviço SOAP

Com o projeto configurado, podemos demonstrar os passos para a criação de um web service SOAP. Para viabilizar esse exemplo, iremos utilizar a biblioteca JAX-WS e, também, as *Annotations* do Java, para declarar quais serão nossos métodos que irão representar os serviços expostos na web.

Assim, uma vez que a aplicação seja colocada no servidor, o JAX-WS é responsável por verificar quais das classes estão anotadas com **@Webservice** e informar ao servidor quais são as URLs de conexão para cada uma dessas classes, criando também os respectivos WSDLs.

Na **Listagem 4** apresentamos a classe **JavaMagazineSoap**, onde criamos um serviço SOAP expondo o método **hello()**, responsável por receber uma **String** e imprimir a mensagem **Hello** no console.

Listagem 4. Criando uma classe para expor um web service SOAP.

```
@WebService
 public class JavaMagazineSoap {

        public String hello(String hi){
              System.out.println("Hello!!!");
              return hi;
       }
 }
```

Como demonstrado nesse código, a criação de web services SOAP é bastante simples, sendo necessário apenas anotar a classe que queremos para que seus métodos sejam expostos como serviços.

Assim que finalizar o desenvolvimento dessa classe, você pode subir nosso projeto no WildFly e então observar o WSDL gerado pelo JAX-WS. Para verificar esse arquivo, basta acessar o diretório indicado no log (exibido no console de saída de sua IDE), na linha *WSDL published to:*. No caso do projeto desse artigo, a linha de log gerada foi: *WSDL published to: file: /Users/brunnoattorre1/Documents/servers/Wildfly8.2/standalone/data/wsdl/JavaMagazineWS.war/JavaMagazineSoapService.wsdl*.

> **Nota**: O caminho será gerado de acordo com o nome do seu projeto. No nosso caso, como o n ome do projeto é JavaMagazineWS, o contexto do caminho ficou com esse nome. Caso seu pr ojeto tenha outro nome, basta substituir o caminho de acordo com a respectiva nomenclatura que será apresentada.

Também é importante notar que, no desenvolvimento dos serviços SOAP, podemos utilizar objetos dentro dos parâmetros de entrada e saída. Na **Listagem 5** apresentamos um exemplo que utiliza objetos como parâmetros de entrada, representados pelos objetos **user1** e **user2**, e saída do serviço, representado pelo objeto retornado pelo método **hello()**. Além disso, também implementamos a classe **User**, que representará o objeto que servirá de parâmetro para o nosso web service.

Listagem 5. Implementação do web service SOAP JavaMagazineSoapObject.

```
@WebService
public class JavaMagazineSoapObject {

	public User hello(User user1, User user2){
		System.out.println(user1.getName());
		return user2;
	}

}

class User{
	private String name;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Casos de uso do SOAP

Para apresentarmos um caso de uso real de web services SOAP e contextualizarmos o que mostramos até aqui, vamos introduzir nesse tópico uma situação em que devemos desenvolver um sistema capaz de receber notificações externas de diversas aplicações.

A escolha do SOAP, nesse caso, se dá pelo fato de que, em nosso cenário fictício, trabalharemos com sistemas das mais diversas linguagens e, muitos deles, não possuem suporte ao REST. Numa situação semelhante a essa, o SOAP se mostra como uma alternativa mais viável, uma vez que viabiliza um suporte mais extenso principalmente para plataformas mais antigas.

O exemplo dessa implementação se encontra na **Listagem 6**, onde apresentamos o código da classe **NotificationSoap**, responsável por receber uma notificação através do protocolo SOAP.

Listagem 6. Classe NotificationSoap implementando um web service de notificação.

```
@WebService
public class NotificationSoap {

	@Oneway
	public void notificate(Message message){
	     System.out.println("Recebeu a mensagem "+ message.getPayload());
	}

}

class Message implements Serializable{
	private String payload;
	private String error;
	public String getPayload() {
	      return payload;
	}
	public void setPayload(String payload) {
	      this.payload = payload;
	}
	public String getError() {
	      return error;
	}
	public void setError(String error) {
	      this.error = error;
	}
}
```

Nessa listagem, além das anotações que já apresentamos anteriormente, introduzimos a anotação **@Oneway**. Essa anotação define que o serviço referente ao método que for anotado será apenas de notificação, ou seja, ele não terá retorno, servindo somente para enviar mensagens.

Neste caso, mesmo que ocorra alguma exceção no processamento da mensagem, essa exceção não será retornada ao cliente. Deste modo, é válido ressaltar que, quando desejamos criar serviços de notificação SOAP, é sempre importante tratarmos corretamente a exceção no próprio serviço, evitando que erros não tratados aconteçam.

Uma vez implementada essa classe, basta subirmos o servidor e acessarmos o WSDL gerado (da mesma forma que acessamos no exemplo anterior) para observarmos as características do serviço criado e, caso seja necessário testar, basta utilizarmos um cliente capaz de realizar requisições SOAP.

> **Nota**: O aplicativo SoapUI é uma alternativa viável para desenvolvedores que precisam testar seus web services. Ela permite, através de um WSDL previamente gerado, realizar requisições com um conteúdo definido pelo usuário. Na seção de **Links** adicionamos mais algumas inform ações de como realizar o download e utilizar essa ferramenta para testar web services.

Criando serviços REST

Agora que entendemos o básico por trás da criação de serviços SOAP, vamos introduzir a biblioteca JAX-RS, responsável por habilitar a criação de serviços utilizando o modelo REST. Para isso, partiremos do mesmo princípio da biblioteca JAX-WS, ou seja, implementaremos, dentro de uma classe Java comum, os métodos que desejamos expor via web service REST.

Uma vez implementada, utilizaremos annotations do Java para indicar a classe e os métodos que desejamos disponibilizar via REST, referenciando também outras informações como o tipo do verbo HTTP para uma determinada requisição, o conteúdo produzido por determinado serviço e a URI de cada um dos serviços.

Além do JAX-RS, iremos utilizar a biblioteca RESTEasy, que declaramos anteriormente como dependência de nosso projeto. Esta será responsável por tratar as requisições REST e direcioná-las ao serviço correto. A configuração dessa biblioteca é feita no arquivo *web.xml*, sendo o arquivo de nosso exemplo apresentado na **Listagem 7**.

Listagem 7. Conteúdo do arquivo web.xml com a configuração do RESTEasy.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  id="WebApp_ID" version="3.1">
    <display-name>JavaMagazineWS</display-name>

     <context-param>
     <param-name>resteasy.scan</param-name>
      <param-value>true</param-value>
    </context-param>

    <context-param>
      <param-name>resteasy.servlet.mapping.prefix</param-name>
      <param-value>/rest</param-value>
    </context-param>

    <listener>
      <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
    </listener>
```

```
  <servlet>
    <servlet-name>resteasy-servlet</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>resteasy-servlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Nessa listagem podemos observar a configuração de alguns itens importantes. O primeiro deles, indicado pelo **context-param** de nome **resteasy.scan**, permite referenciar se queremos que o RESTEasy rastreie em nosso projeto todas as classes que possuem a anotações **@Path** do JAX-RS e, em seguida, disponibilize essas classes como serviços REST.

No caso de nosso projeto, indicamos o valor do parâmetro **resteasy.scan** como **true** e possibilitamos ao RESTEasy realizar esse trabalho automaticamente.

Em segundo lugar, configuramos o listener **ResteasyBootstrap** e a servlet **HttpServletDispatcher**, responsáveis por, para cada requisição, fazer o redirecionamento à classe correta, que foi escaneada no passo anterior.

Por fim, definimos também qual serão as URLs que servirão de endpoints para nossas requisições REST. Essas são indicadas tanto no parâmetro **resteasy.servlet.mapping.prefix** como, também, no **servlet-mapping** do nosso servlet **HttpServletDispatcher**, ambas colocando, como endereço da URL, o endpoint com o caminho **/rest/***.

Uma vez terminadas as configurações dentro do *web.xml*, basta anotarmos uma classe com as anotações do JAX-RS para podermos indicar, ao RESTEasy, que essa classe servirá como um serviço REST. Para demonstrar o uso dessas anotações, na **Listagem 8** apresentamos o código da classe **JavaMagazineRest**, que recebe uma requisição GET e devolve um HTML com a mensagem "**Hello, World!!**".

Listagem 8. Código da classe JavaMagazineRest, que implementa um web service REST.

```
@Path("/rest")
public class JavaMagazineRest {

       @GET
       @Produces("text/html")
       public String example(){
              return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";
      }
}
```

Conforme podemos verificar, a criação da classe **JavaMagazineRest** e o uso das anotações do JAX-RS podem ser explicados em três etapas principais. A primeira delas, definida pela anotação **@Path** aplicada em um método, permite que indiquemos o caminho pelo qual o serviço REST irá responder. No caso desse exemplo, qualquer requisição à URI */rest* será redirecionada ao nosso serviço.

Logo em seguida, explicitamos o método HTTP no qual o conteúdo da requisição deve ser enviado pelo cliente. Assim, uma vez que essa requisição chegue à aplicação, será respondida pelo método anotado de nossa classe **JavaMagazineRest**. No nosso exemplo, isso é feito através da anotação **@GET**, especificando que o recurso anotado só responderá às requisições do tipo GET.

Por fim, a anotação **@Produces** indica o tipo do dado de retorno (no nosso exemplo, HTML). Caso recebêssemos algum parâmetro de entrada, poderíamos fazer uso da anotação **@Consumes**, que indica o tipo de dado aceito como input para o serviço.

Com a classe desenvolvida, para observarmos o REST em funcionamento, basta subir nossa aplicação no WildFly, acessar a URL *http://localhost:8080/JavaMagazineWS/rest/rest* e então verificar a mensagem "Hello, World".

Casos de uso do REST

Agora que aprendemos os conceitos básicos por trás do desenvolvimento de serviços REST, apresentaremos nesse tópico uma análise simplificada de um caso de uso real dessa opção de web service para um sistema comercial. Para isso, suponha um cenário onde vamos desenvolver um serviço de CRUD para um cadastro de usuários.

A construção desse serviço é feita a partir do código da **Listagem 9**, onde implementamos a classe **CrudRest**. Esta define os endpoints REST que servirão para expor os endereços de criação, atualização, recuperação e deleção de instâncias da classe **User**, que representa os usuários de nosso sistema.

Listagem 9. Web services REST para o CRUD do objeto User.

```
@Path("/user")
public class CrudRest {

	@Path("/{id}")
	@GET
	@Produces("application/json")
	public Response getUser(@PathParam("id") String id) {
		// Simulando a busca por um usuário
		System.out.println("buscando usuario "+ id);
		User user = new User();
		user.setName("user Test");
		return Response.status(200).entity(user).build();
	}

	@POST
	@Consumes("application/json")
	@Produces("application/json")
	 public Response createUser( User user) {
```

```
            // Simula uma inserçao no banco
            System.out.println("Criando usuario ");
            return Response.status(200).entity(user).build();


        }
        @Path("/{id}")
        @PUT
        @Consumes("application/json")
        @Produces("application/json")
         public Response updateUser(@PathParam("id")String id,User user) {
           // Simula uma atualizacao no banco
           System.out.println("Atualizando usuario "+ id);
            return Response.status(200).entity(user).build();
         }


         @Path("/{id}")
         @DELETE
         public Response deleteUser(@PathParam("id") String id){
               //Simula uma delecao
               System.out.println("deletando usuario "+ id);

               return Response.status(200).build();
         }
}

class User{
       private String name;

         public String getName() {
               return name;
         }

         public void setName(String name) {
               this.name = name;
         }
}
```

Nesse código, podemos observar que seguimos alguns dos princípios primordiais do REST. O primeiro deles está relacionado ao caminho de nossas URLs: todas seguem o mesmo padrão e nomenclatura, somente mudando o tipo do método HTTP que será utilizado.

Esse tipo de abordagem permite que os serviços se tornem mais intuitivos, ou seja, não é necessária uma extensa documentação para descrever o que cada um faz, pois o próprio método HTTP já nos indica sua funcionalidade.

Além disso, também relacionado às URLs expostas, implementamos o conceito de trabalhar com as entidades do sistema na forma de **recursos** nos links e operações REST.

Esse conceito permite que, em uma aplicação REST, sempre trabalhemos com a referência dos recursos que desejamos utilizar nas próprias URLs (por exemplo, em nosso caso, será uma palavra que servirá de identificador único para cada uma de nossas entidades).

Contextualizando essa funcionalidade para o nosso exemplo, se quisermos fazer qualquer ação em cima do objeto **User** com **id** igual a "345abd", a URL de destino seria sempre a representada pelo endereço */user/345abd*.

A partir daí, qualquer ação que desejamos tomar sobre esse recurso (**User**) deve ser definida pelos métodos HTTP correspondentes. Consequentemente, uma requisição do tipo **DELETE** a essa URL iria remover o usuário cujo id é igual "345abd", enquanto uma requisição do tipo **GET** iria trazer, como retorno, os dados desse usuário.

Além das URLs, podemos observar que também definimos em nossa classe quais são os tipos de dados de entrada e saída para cada um dos serviços. Nas anotações **@Consumes** e **@Produces**, especificamos, respectivamente, qual o tipo de dado de entrada e saída. Em nosso caso serão as informações e objetos serializados no formato JSON. É importante notar que, uma vez que chegue uma requisição a esses serviços, os headers **Content-Type** (tipo de dado de envio) e **Accept** (tipo de dado que aceita como retorno) devem ter, como conteúdo, o mesmo valor dos definidos em ambas as anotações (**@Consumes** e **@Produces**).

Por fim, para observarmos o funcionamento desses serviços REST recém-desenvolvidos, basta subir nossa aplicação e realizar as respectivas requisições (com os verbos HTTP corretos e os headers de **Content-Type** e **Accept** claramente descritos). Uma vez que essas requisições sejam feitas, podemos verificar o comportamento de cada um de nossos métodos REST e, também, observar os resultados que serão retornados.

Nota: Como sugestão para testar esses web services, o plug-in do Google Chrome chamado Postman possibilita que sejam feitas as mais diversas requisições, permitindo explicitar os verbos HTTP a serem usados e também definir os headers "Content-type" e "Accept". Para mais informações, adicionamos uma referência na seção **Links**.

Considerações finais a respeito do uso de REST x SOAP

Com os principais conceitos por trás de web services, tanto do protocolo SOAP como do modelo REST, analisados, é importante entendermos as diferenças e vantagens na utilização de cada um.

Dito isso, a primeira diferença que precisamos ter em mente é que web services SOAP seguem um protocolo específico, enquanto os serviços REST seguem um modelo arquitetural. Essa diferença é importante pois, enquanto o SOAP exige alguns padrões para funcionar, ou seja, se não seguir todos os padrões não conseguimos expor nem consumir um serviço desse tipo; o REST indica algumas boas práticas e aconselha um modelo arquitetural, ficando a critério do desenvolvedor utilizar todos ou somente os que atendem à necessidade do projeto.

Outro ponto que o REST se difere do SOAP está no fato de ser uma solução mais leve e de fácil implementação, acarretando em um ganho considerável de velocidade de processamento, sendo assim bastante prático e indicado para projetos onde a performance é essencial.

O SOAP, no entanto, tem funcionalidades muito úteis em situações específicas, trazendo um leque de artefatos para auxiliar o desenvolvedor. Soluções que envolvem autenticação, manter a sessão entre requisições, WS-Addressing (URLs de callback), entre outras, já vêm embutidas no SOAP e podem ser muito úteis em determinadas situações;

Vale a pena lembrarmos também que o desenvolvimento de aplicações REST bem arquitetadas depende muito do time por trás de sua implementação. Os padrões SOAP, por sua vez, já são engessados e limitam a liberdade de customização por parte do desenvolvedor. Devido a isso, o REST é mais aberto e permite que sejam desenvolvidas as mais diversas combinações de recursos dentro de seu modelo arquitetural.

No entanto, isso acarreta tanto vantagens como desvantagens. Vantagens pois, para uma equipe mais experiente, a customização e utilização das funcionalidades necessárias se tornam mais fáceis e práticas de serem implementadas.

Como desvantagem, está o fato de que, caso não se tenha um cuidado especial na modelagem e na implementação dos serviços, uma arquitetura REST pode levar o projeto ao fracasso. Com o intuito de auxiliar a escolha entre as tecnologias REST e SOAP, na **Tabela 1** resumimos as principais diferenças, vantagens e desvantagens de cada uma.

	SOAP	**REST**
Transporte de dados	Mais pesado e custoso, pois necessita do uso do envelope SOAP.	Mais leve e dinâmico, pois não possui um padrão específico para serialização de dados.
Tipo de dados	Sempre trabalha com XML	Pode trabalhar com qualquer tipo de dado, desde que seja especificado na implementação do serviço.
Requisições HTTP	Suporta somente POST	Suporta (e aconselha) o uso de todos os verbos HTTP.
Modelo arquitetural	É um protocolo.	É um conjunto de boas práticas na modelagem de web services.
Suporte em outras linguagens	É suportado basicamente por todas as linguagens, principalmente linguagens mais antigas.	É melhor suportado por linguagens mais modernas e voltadas ao desenvolvimento para web (JavaScript, por exemplo).
Funcionalidades	Possui diversas funcionalidades mais avançadas em sua implementação padrão, como o WS Addressing.	Estipula somente o básico dos serviços, sendo necessária a implementação de funcionalidades mais avançadas.
Uso dentro do Java	Possui diversas	Exige a criação e implementação

	funcionalidades, como a criação de classes a partir de WSDLs, que facilitam o desenvolvimento.	dos métodos que receberão as chamadas HTTP e permite uma maior liberdade na escolha das funcionalidades que deseja implementar.

Tabela 1. Principais diferenças entre SOAP e REST.

Enfim, ambas as tecnologias apresentam vantagens e desvantagens e possuem situações específicas nas quais se encaixam melhor para solucionar um determinado problema. Portanto, para escolher a tecnologia ideal para o seu projeto, é aconselhável a opinião de profissionais mais experientes e uma análise de todo o contexto de execução em que os serviços irão rodar.

Links

Especificação W3C SOAP.
http://www.w3.org/TR/soap/

Definições do REST pela W3C.
http://www.w3.org/2001/sw/wiki/REST

Página com instruções de instalação do JBoss Tools e do WildFly 8.
http://www.mastertheboss.com/wildfly-8/configuring-eclipse-to-use-wildfly-8

Cliente do Postman.
https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm

Cliente do SOAPUI.
http://www.soapui.org

Uso do SOAPUI para testes SOAP.
http://www.soapui.org/Getting-Started/your-first-soapui-project.html

por Brunno Fidel
Expert em Java e programação Web