

How to REST?!

By Nima Goudarzi (nima@javadev.org) - July, 2007

This is an article about REST (Representational State Transfer) which gives you the knowledge of developing enterprise applications with SOA as a loosely coupled approach but without getting involved with SOAP and its complexities.

This article consists on the following sections:

- Definitions
- Example: Authentication Service
- Authentication Web Service
- Authentication Client
- XML Transformation
- RESTing Without JAX-WS

* The complete source codes and binary versions of the examples used in this article are available at: <http://www.javadev.org/files/rest.zip>

Definitions

What is REST?

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term was introduced in the doctoral dissertation in 2000 by Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification, and has come into widespread use in the networking community.

REST strictly refers to a collection of network architecture principles that outline how resources are defined and addressed. The term is often used in a looser sense to describe any simple interface that transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking via HTTP cookies.

Principles

REST's proponents argue that the Web enjoyed the scalability and growth that it has had as a direct result of a few key design principles:

- Application state and functionality are divided into **resources**
- Every resource is uniquely addressable using a **universal syntax** for use in **hypermedia links**
- All resources share a **uniform interface** for the transfer of state between client and resource, consisting of

- A constrained set of **well-defined operations**
- A constrained set of **content types**, optionally supporting **code-on-demand**
- A protocol that is:
 - **Client/Server**
 - **Stateless**
 - **Cacheable**
 - **Layered**

REST's client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries--proxies, gateways, and firewalls--to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching.

REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cache ability. —*Roy Fielding*

REST vs. RPC

A RESTful web application requires a different design approach from an RPC (Remote procedure call) application. An RPC application is exposed as one or more network objects, each with an often unique set of functions that can be invoked. Before a client communicates with the application it must have knowledge of the object identity in order to locate it and must also have knowledge of the object type in order to communicate with it.

RESTful design constrains the aspects of a resource that define its interface (the verbs and content types). This leads to the definition of fewer types on the network than an RPC-based application but more resource identifiers (nouns). REST design seeks to define a set of resources that clients can interact with uniformly, and to provide hyperlinks between resources that clients can navigate without requiring knowledge of the whole resource set. Server-provided forms can also be used in a RESTful environment to describe how clients should construct a URL in order to navigate to a particular resource.

—*Wikipedia*

Example: Authentication Service

As an example we are going to develop an authentication service. This service receives an xml document as input and returns an xml document as output. The input xml document contains username and password which needs to be authenticated and the output xml contains a simple XML message to return the result of authentication to the client.

As the first step, we need to define an interface for our service to format the incoming and outgoing XML documents.

Almost all the SOAP-based SOA applications use WSDL to define their interfaces, but in the RESTful application there is no WSDL to use. So we need to find another way for defining our service interfaces. One of the most widely used approaches is using XML Schema.

In this example we use a simple XML Schema to restrict the input and output xml messages to our desired format.

So:

1. XML documents are used to exchange messages between applications.
2. XML Schema documents define the application interfaces.

Example of incoming XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<auth xmlns="http://www.javadev.org/auth"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.javadev.org/auth
http://javadev.org/rest/auth/auth.xsd">
  <username>foo</username>
  <password>foo</password>
</auth>
```

Example of the schema used as the interface:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:auth="http://www.javadev.org/auth"
targetNamespace="http://www.javadev.org/auth" elementFormDefault="qualified">
  <element name="username" type="xs:string"/>
  <element name="password" type="xs:string"/>
  <element name="auth">
    <complexType>
      <sequence>
        <element ref="auth:username"/>
        <element ref="auth:password"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Client applications send XML messages to the server in the defined format. This transition is done via the HTTP protocol but the way we use to do this transition varies:

JAX-WS : JAX-WS is a fundamental technology for developing SOAP based and RESTful Java Web services. JAX-WS is designed to take the place of JAX-RPC in Web services and Web applications. We use JAX-WS technology to send/receive XML messages to/from web services. In this approach XML

messages are transferred as StreamSource objects and connection to the web service is done via javax.xml.ws.Service and by using Dispatches. This approach will be introduced in more details.

HttpURLConnection: another way to transfer messages between RESTful web services is using HttpURLConnection. In this approach the Web service is accessed with an HTTP GET request. The client application needs to issue the HTTP GET, and process the HTTP response stream that contains the XML document.

Depends on our requirements, we can use one of the approaches mentioned above. In this article we use JAX-WS API, however we will take a look at the other approach to find out differences.

Our example consists of two parts:

- **Web Service:** Web Service receives an XML document which needs to be authenticated and returns an XML message which indicates that the authentication was passed or failed.

- **Client:** Client is an application, It can be a simple java application, another web service or any other kind of application. Client is responsible for sending the authentication XML message and receiving the result XML message.

Now we explain each part in more details.

Authentication Web Service: AuthProvider.java

Our web service class starts with three annotations:

```
@ServiceMode(value = Service.Mode.PAYLOAD)
@WebServiceProvider(serviceName = "authService")
@BindingType(value = HTTPBinding.HTTP_BINDING)
```

In the SOAP web services **@ServiceMode** is set to MESSAGE which indicates that you want to work with the entire SOAP envelop, but in the RESTful web services it is set to PAYLOAD to indicate that we just need the SOAP body of the message.

@WebServiceProvider annotation is required for deploying a RESTful web service and its serviceName attribute is used to deploy our web service with a desired name. This name is used in the client application to access the web service.

The **@BindingType** annotation (javax.xml.ws.BindingType) is also defined by the JAX-WS 2.0 specification and is used to specify the binding that should be employed when publishing an endpoint. The property value indicates the actual binding. In this case, you can see that the value is specified as follow:

```
value=HTTPBinding.HTTP_BINDING
```

This indicates that the XML/HTTP binding should be used, rather than the default SOAP 1.1/HTTP. This is how REST endpoints are specified in JAX-WS 2.0—by setting the **@BindingType**. If one were to leave the

@BindingType annotation off this example, the Java EE 5 container would deploy it as a service that expects to receive a SOAP envelope, rather than straight XML over HTTP.

The web service class should implement the javax.xml.ws.Provider.

This interface enables you to create a web service that works directly with the XML message as an instance of javax.xml.transform.Source.

The web service class receives the XML message as an instance of javax.xml.transform.Source. So the first step toward authentication is to convert this object to an XML document, then we can parse the XML document to extract our required information.

The transformation is done in the parse(Source src) method which get an Source object as input parameter and returns the parsed XML document.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
StreamResult res = new StreamResult(baos);
try {
    TransformerFactory.newInstance().newTransformer()
        .transform(src, res);
} catch (Exception e) {
    e.printStackTrace();
}
```

This code transforms the Source object to an OutputStream object. Then this OutputStream object is converted to an InputStream object and passed to the DomParser to be converted to the parsed XML document (I used the Apache Xerces-J 2 for XML parsing).

```
ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());

try {
    InputSource is = new InputSource(bais);

    // Create a DOM Parser
    DOMParser parser = new DOMParser();

    // Parsing the incoming file
    parser.parse(is);

    // Obtain the document
    doc = parser.getDocument();

} catch (IOException ioe) {

    ioe.printStackTrace();

} catch (SAXException sax) {
```

```
saxe.printStackTrace();  
}
```

Once we have the parsed XML document, we can extract the required information needed for the authentication.

```
NodeList root = doc.getElementsByTagName("auth");  
Element rootEL = (Element) root.item(0);  
String username = getStringVal(rootEL, "username");  
String password = getStringVal(rootEL, "password");
```

Now we have the passed in Username and Password which needs to be authenticated. After authenticating Username and Password we must return the appropriate message. This message will be an XML message but as a StreamSource object (as the message passed into the web service).

```
if (auth(username, password)) {  
    String message = "";  
  
    // Write the authentication result to an XML message  
    message += "<?xml version='1.0'?>";  
    message += "<result>";  
    message += "<message>authenticated</message>";  
    message += "</result>";  
  
    // Create an InputStream with the authentication info  
    ByteArrayInputStream bais = new ByteArrayInputStream(message.getBytes());  
    return new StreamSource(bais);  
} else {  
    String message = "";  
  
    // Write the authentication result to an XML message  
    message += "<?xml version='1.0'?>";  
    message += "<result>";  
    message += "<message>failed</message>";  
    message += "</result>";  
  
    // Create an InputStream with the authentication info  
    ByteArrayInputStream bais = new ByteArrayInputStream(message.getBytes());  
    return new StreamSource(bais);  
}
```

The client will use the same transformation we did here to parse this XML message and find out the authentication result.

Deploying the Web Service

By using **@WebServiceProvider** annotation, deploying our web service is as easy as putting the .war file in the deploy directory of the application server. The application server detects it as a web service and deploys it automatically.

Authentication Client: Client.java

We are using a simple java application as our client.

The client is using the JAX-WS API to communicate with the web service.

It first generates an XML message which contains information need to be authenticated.

```
info += "<?xml version=\"1.0\"?>";

info += "<auth xmlns=\"http://www.javadev.org/auth\"
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" \n" +
"xsi:schemaLocation=\"http://www.javadev.org/auth\n" +
"http://www.javadev.org/rest/auth/auth.xsd\">";

info += "<username>foo</username>";
info += "<password>foo</password>";

info += "</auth>";
```

As you can see, we are generating an XML message with the Username and Password which must be authenticated in the web service.

In the client, `Service` is used to create an instance of `javax.xml.ws.Dispatch<Source>`, which enables XML message-level interaction with the target Web service. `Dispatch` is the low-level JAX-WS 2.0 API that requires clients to construct messages by working directly with the XML, rather than with a higher-level binding such as JAXB 2.0 schema derived program elements. For many REST proponents, however, this is exactly the programming paradigm they want—direct access to the XML request and response messages.

The client uses the `Service.addPort()` method to create a port within the `Service` instance that can be used to access the RESTful web service.

Next, the `Service.createDispatch()` method is invoked to create an instance of `Dispatch<Source>`—a `Dispatch` instance that enables you to work with XML request/response messages as instances of `javax.xml.transform.Source`.

The `Dispatch.invoke()` method then packages the XML request—per the JAX-WS 2.0 HTTP Binding—and sends it to the RESTful service. The `invoke()` method waits for the response before returning.

The service processes the HTTP GET and sends an HTTP response that includes the XML.

The `invoke()` method returns the response XML message as an instance of `Source`.

```
// Create an InputStream with the authentication info
    ByteArrayInputStream bais = new ByteArrayInputStream(info.getBytes());

    QName svcQName = new QName("http://rest", "svc");
    QName portQName = new QName("http://rest", "port");
    Service svc = Service.create(svcQName);
    svc.addPort(portQName, HTTPBinding.HTTP_BINDING, url);
    Dispatch<Source> dis =
        svc.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
    StreamSource result = (StreamSource) dis.invoke(new StreamSource(bais));
```

Notice that you have to create QName instances for the Service instance and the “port” that corresponds to the RESTful Web service. In a SOAP scenario, these qualified names would correspond to the WSDL definitions for the wsdl:service and wsdl:port. Since there is no WSDL when invoking a RESTful service, these QName instances are gratuitous. They are required by the API, but not used to invoke the RESTful service.

The URL used in the addPort method is the URL of your web service for example we used `http://localhost:8080/auth/authService` which:

localhost is the name/IP of your running application server.

8080 is the port port of your application server.

auth is the context path of your web service which in our example is defined.

authService is the value of serviceName attribute of the `@WebServiceProvider` annotation in the web service.

As you can see the authentication result is returned as an Source object. So we should parse it and extract the message as we did in the web service.

XML Transformation

In the previous sections, we sent and received XML messages to/from web service.

In this communication the client and web service received the XML message in the formats that they were expecting. These formats were defined in the XML schema files. But sometimes we need to adapt ourselves with different formats of information. For example some services may use email instead of username to identify their customers. So we must provide a way for different XML formats to be acceptable by our RESTful web service.

Here we need to transform the incoming xml message to an acceptable format. We can use XSLT for this reason.

XSLT makes sense as the transformation tool of choice within SOA integration frameworks, because it is a universally accepted standard and the transformation engines that interpret XSLT to perform data transformations keep getting better and faster.

As an example assume that the authentication message came from the client is in the following format:

```
<?xml version="1.0"?>
<auth xmlns="http://www.javadev.org/mail"
xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xsi:schemaLocation="http://www.javadev.org/mail
http://javadev.org/rest/auth/mail.xsd">
  <email>foo</email>
  <password>foo</password>
</auth>
```

As you can see <username> is replaced with <email>, for the authentication service there is no difference between username and email, because both of them are unique. But the main point of the consideration is that currently, when we are parsing the incoming xml document, we expect the <username> element but we are facing an <email> element.

To transform the incoming XML document to the expected format we write an XSLT and use it in the JAXP API.

The XSLT document is as bellow:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:mail="http://www.javadev.org/mail">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:template match="mail:auth">
    <auth xmlns="http://www.javadev.org/auth"
xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xsi:schemaLocation="http://www.javadev.org/auth
http://javadev.org/rest/auth/auth.xsd">
      <xsl:apply-templates/>
    </auth>
  </xsl:template>
  <xsl:template match="mail:email">
    <xsl:apply-templates select="./mail:email"/>
  </xsl:template>
  <xsl:template match="mail:password">
    <xsl:apply-templates select="./mail:password"/>
  </xsl:template>
  <xsl:template match="mail:email">
    <username>
      <xsl:value-of select="."/>
    </username>
  </xsl:template>
  <xsl:template match="mail:password">
    <password>
      <xsl:value-of select="."/>
    </password>
  </xsl:template>
```

```
</password>
</xsl:template>
</xsl:stylesheet>
```

This XSLT transformer reads the source XML file and converts it to the destination XML format. After creating the XSLT file, we need to apply it to the incoming xml file before parsing it.

```
InputStream xslt = this.getClass().getResourceAsStream("mail_to_user.xslt");
try {
    TransformerFactory.newInstance().newTransformer(new StreamSource(xslt))
        .transform(src, res);
} catch (Exception e) {
    e.printStackTrace();
}
```

The only difference is that we pass the XSLT document as an StreamSource object to the Transformer to use it before converting the incoming StreamSource object to the equivalent OutputStream. So we can convert any format of incoming messages to our desired format before extracting information.

RESTing Without JAX-WS

During our example we used JAX-WS API to communicate with the web service. Although this is the best way, but there are other ways as well.

One of these ways is using HttpURLConnection:

1. The client uses the URL.openConnection() method to create an instance of HttpURLConnection representing a connection to the Web service's URL.
2. HttpURLConnection.connect() sends the HTTP GET request that has been configured using the Web service's URL.
3. The Web service processes the request and writes the appropriate XML document to the HTTP response stream.
4. The HttpURLConnection's InputStream is used to read the HTTP response's XML document.

Exmample of a non-JAX-WS Client

```
URL url = new URL(""); // the web service URL
URLConnection con = (URLConnection) url.openConnection();
con.setRequestMethod("GET");
con.connect();
InputStream in = con.getInputStream();
byte[] b = new byte[1024]; // 1K buffer
int result = in.read(b);
while (result != -1) {
    System.out.write(b,0,result);
    result =in.read(b);
}
in.close();
con.disconnect();
```

Note that the in this case the web service does not return the XML message directly in the StreamSource format but it writes the StreamSource result to the response OutputStream.

References:

- *SOA Using Java™ Web Services by Mark D. Hansen*
- *Wikipedia*