



Enterprise Tech Tips

Get tips on using enterprise Java technologies and APIs, such as those in Java Platform, Enterprise Edition (Java EE).



« [Using WS-Trust Suppo...](#) | [Main](#) | [Tech Tips Quiz](#) »

Configuring JSON for RESTful Web Services in Jersey 1.0

By edort on Oct 27, 2008

by [Jakub Podlesak](#)

Note: The API for configuring JSON in Jersey has changed with the recently released 1.0.2 version. Please see http://blogs.sun.com/japod/entry/configuring_json_for_restful_web.

Jersey 1.0 is an open-source, production-ready reference implementation of JAX-RS, the Java API for RESTful Web Services (JSR-311). Jersey makes it easy to create RESTful web services in Java.

In an earlier Tech Tip, [Implementing RESTful Web Services in Java](#), Paul Sandoz and I introduced RESTful Web Services, JAX-RS, and Jersey, and showed how to write RESTful web services in Java that conform to the JAX-RS specification. In this tip you will learn how to configure data in JSON (JavaScript Object Notation) using Jersey 1.0. JSON is a lightweight data-interchange format that is based on the object notation of the JavaScript language. Because of its simple text format, JSON provides a good alternative to other data interchange formats such as XML and is particularly attractive as a data interchange format for RESTful web services.

In this tip you will build a Jersey-based web application that provides information about printer status. The application returns the information in JSON format. To build the application, you will use the Maven 2 software project management tool. For more information about Maven, see [Welcome to Maven](#) and [Building Web Applications with Maven 2](#).

Creating a Jersey-Based Web Application With Maven 2

The first step in creating a Jersey-based web application with Maven 2 is to create a Maven 2 project. You can do this by running the following Maven 2 archetype plugin:

```
mvn archetype:generate -DarchetypeCatalog=http://download.java.net/maven/2
```

In response, Maven 2 will prompt you to choose an archetype, that is, a Maven 2 project template, from the archetypes listed in the archetype catalog, archetype-catalog.xml, at URL <http://download.java.net/maven/2>:

About

edort

Search

Enter search term:

☐ Search only this blog

Recent Posts

Security Token Service and Identity Delegation with Metro

DataSource Resource Definition in Java EE 6

Asynchronous Support in Servlet 3.0

POST-REDIRECT-GET and JSF 2.0

Using CDI and Dependency Injection for Java in a JSF 2.0 Application

Locking and Concurrency in Java Persistence 2.0

A Sampling of EJB 3.1

A Common Ant Build File for Metro-Based Services and Clients

Jersey and Spring

Tech Tips Quiz

Choose archetype:

```
1: http://download.java.net/maven/2 -> jersey-quickstart-grizzly (Archetype for cre
2: http://download.java.net/maven/2 -> jersey-quickstart-webapp (Archetype for crea
Choose a number: (1/2):
```

Choose 1 (jersey-quickstart-grizzly). You will then be asked to enter a groupid and some other inputs for the web application. You may use the following values:

```
Define value for groupId: : com.example
Define value for artifactId: : printer-status-webapp
Define value for version: 1.0-SNAPSHOT: :
Define value for package: com.example: :
Confirm properties configuration:
groupId: com.example
artifactId: printer-status-webapp
version: 1.0-SNAPSHOT
package: com.example
Y: :
```

After you confirm the inputs, Maven 2 creates a new subdirectory called `printer-status-webapp`, which contains a template for your new Jersey-based web application. [Figure 1](#) shows the expanded file structure of the `printer-status-webapp` subdirectory.



Figure 1. Web Application Template Structure Created By Maven 2

Maven 2 also creates a Project Object Model (POM) file, which contains an XML representation of the Maven project. You can find the `pom.xml` file in the `printer-status-webapp` subdirectory.

Adding REST Resources

Top Tags

.net 109 196 2.0 299 3.0
314 330 annotation api
application beans
components composite
developer ee ejb
enterprise faces
faces glassfish java
javabeans javaserver
jax-rs jaxb jersey jmake
jpa jsf jsr metro
mysql performance
persistence phobos quiz
rest security
services servlet sip
spring tech technology
tips ui view web wsit

Categories

Enterprise Java technology
Enterprise JavaBeans
Technology
JAX-RS
Java Persistence
JavaServer Faces (JSF)
technology
Metro
OpenSolaris
Personal
Phobos
Servlet API
Session Initiation Protocol
(SIP)
Students
Sun

If you navigate below the printer-status-webapp subdirectory to src/main/java/com/example, you'll see a Java class named MyResource. This class identifies the URI for a resource that represents a simple message.

```
package com.example;

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

// The Java class will be hosted at the URI path "/myresource"
@Path("/myresource")
public class MyResource {

    // TODO: update the class to suit your needs

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getIt() {
        return "Got it!";
    }
}
```

Notice the JAX-RS annotations in the file. The `@Path` annotation identifies the URI path for which the `MyResource` class will serve requests. Recall that an important concept in REST is the existence of resources, each of which can be referred to using a global identifier, that is, a URI. In order to manipulate these resources, components of the network, clients and servers, communicate using a standardized interface such as HTTP and exchange representations of these resources. The `@GET` annotation indicates that the `getIt()` method responds to HTTP GET requests. The `@Produces` annotation indicates that the `getIt()` method returns plain text.

To demonstrate JSON functionality in Jersey, let's add some additional REST resources. In order to do so, you need to enable JSON support in the application. Edit the `pom.xml` file, and uncomment the following part:

```
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-json</artifactId>
  <version>${jersey-version}</version>
</dependency>
```

[datasource](#)
[jMaki](#)
[performance](#)
[quiz](#)
[security](#)
[web services](#)

Archives

◀ November 2012

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Today

Bookmarks

[BigAdmin blog](#)
[Core Java Technologies Tech Tips](#)
[Enterprise Java Technologies Tech Tips Archive](#)
[Java EE](#)
[Java Technology Fundamentals](#)
[SDN Program News](#)
[blogs.sun.com](#)
[java.com](#)
[java.net](#)
[opensolaris.org](#)

Menu

You also need to update the jersey-version property in the pom.xml file to the currently available stable version, 1.0:

```
<properties>
  <jersey-version>1.0</jersey-version>
</properties>
```

You're now ready to add JSON resources. To do that, let's create some [Java Architecture for XML Binding \(JAXB\)](#) beans to model status information for a printer. Then you'll make the beans accessible as REST resources.

First, in the src/main/java/com/example subdirectory, create a file StatusInfoBean.java with the following code:

```
package com.example;

import java.util.Collection;
import java.util.HashSet;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class StatusInfoBean {

    public String status = "Idle";
    public int tonerRemaining = 25;
    public final Collection<JobInfoBean> jobs = new HashSet<JobInfoBean>();
}
```

Next, in the src/main/java/com/example subdirectory create a file JobInfoBean.java with the following code:

```
package com.example;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "job")
public class JobInfoBean {
    public String name;
    public String status;
    public int pages;

    // just to make JAXB happy
```






















[Blogs](#) [Home](#)

[Weblog](#)



[Login](#)

Feeds

RSS

-  [All](#)
-  [/Enterprise Java technology](#)
-  [/Enterprise JavaBeans Technology](#)
-  [/JAX-RS](#)
-  [/Java Persistence](#)
-  [/JavaServer Faces \(JSF\) technology](#)
-  [/Metro](#)
-  [/OpenSolaris](#)
-  [/Personal](#)
-  [/Phobos](#)
-  [/Servlet API](#)
-  [/Session Initiation Protocol \(SIP\)](#)
-  [/Students](#)
-  [/Sun](#)
-  [/datasource](#)
-  [/jMaki](#)
-  [/performance](#)
-  [/quiz](#)
-  [/security](#)
-  [/web services](#)
-  [Comments](#)

Atom

-  [All](#)
-  [/Enterprise Java technology](#)

```

public JobInfoBean(){};

public JobInfoBean(String name, String status, int pages) {
    this.name = name;
    this.status = status;
    this.pages = pages;
}

}

```

To make the beans accessible as REST resources, you need to update the `MyResource.java` file in the `src/main/java/com/example` subdirectory as follows:

```

package com.example;

import com.sun.jersey.spi.resource.Singleton;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Singleton
@Path("/status")
public class MyResource {




















    StatusInfoBean statusInfoBean = new StatusInfoBean();

    {{
        statusInfoBean.jobs.add(new JobInfoBean("sample.doc", "printing...", 13));
    }}

    @GET
    @Produces("application/json")
    public StatusInfoBean getStatus() {
        return statusInfoBean;
    }

    @PUT
    @Consumes("application/json")

```

-  /Enterprise JavaBeans Technology
-  /JAX-RS
-  /Java Persistence
-  /JavaServer Faces (JSF) technology
-  /Metro
-  /OpenSolaris
-  /Personal
-  /Phobos
-  /Servlet API
-  /Session Initiation Protocol (SIP)
-  /Students
-  /Sun
-  /datasource
-  /jMaki
-  /performance
-  /quiz
-  /security
-  /web services
-  Comments

```

        public synchronized void setStatus(StatusInfoBean status) {
            this.statusInfoBean = status;
        }
    }
}

```

Notice that the `getStatus()` method returns data in JSON format to the client and the `setStatus()` method processes data in JSON format sent by the client:

```

@Produces("application/json")
public StatusInfoBean getStatus() {...}

@Consumes("application/json")
public synchronized void setStatus(StatusInfoBean status) {...}

```

Running the Application

You can run the web application with the following Maven 2 lifecycle command in the `printer-status-webapp` subdirectory:

```
mvn clean compile exec:java
```

The command cleans out all Maven 2-generated files and compiles the Java source. It also starts the [Grizzly](#) container with the web application running within it.

```

C:\printer-status-webapp>mvn clean compile exec:java
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] -----
[INFO] Building printer-status-webapp
[INFO]    task-segment: [clean, compile, exec:java]
[INFO] -----
[INFO] [clean:clean]
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
Downloading: http://download.java.net/maven/2//com/sun/jersey/jersey-server/1.0/
jersey-server-1.0.pom
...
[INFO] [compiler:compile]
[INFO] Compiling 4 source files to C:\printer-status-webapp\target\classes
[INFO] Preparing exec:java

```

```
[INFO] No goals needed for project - skipping
[INFO] [exec:java]
Starting grizzly...
Jersey app started with WADL available at http://localhost:9998/application.wadl
Hit enter to stop it...
```

Open your browser and point it to <http://localhost:9998/status>. You will see the following:

```
{"status":"Idle", "tonerRemaining":"25", "jobs":[{"name":"sample.doc", "status":"printi
```

This confirms that the Jersey-based web application returns data in JSON format.

Improving the Application

Although the application returns data in JSON format, there are some improvements you might want to make. For example, if you add another printer job and rerun the application, it would return something like the following:

```
{"status":"Idle", "tonerRemaining":"25", "jobs":[{"name":"sample.ps", "status":"print
```

The jobs object in the JSON data is an array, something that wasn't evident in the output for a single printer job. You might want to change the application so that the result is seen as an array even if it contains just a single element. Otherwise it could cause some problems when the JSON is processed by the client.

There is another thing you might consider changing. The values for tonerRemaining and pages are currently listed as strings. You probably want to represent those results as numbers.

To make these improvements, create a file named `MyJAXBContextResolver.java` in the `src/main/java/com/example` subdirectory with the following content:

```
package com.example;

import com.sun.jersey.api.json.JSONJAXBContext;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import javax.ws.rs.ext.ContextResolver;
import javax.ws.rs.ext.Provider;
import javax.xml.bind.JAXBContext;

@Provider
public class MyJAXBContextResolver implements ContextResolver<JAXBContext> {
```

```

private JAXBContext context;
private Class[] types = {StatusInfoBean.class, JobInfoBean.class};

public MyJAXBContextResolver() throws Exception {
    Map props = new HashMap<String, Object>();
    props.put(JSONJAXBContext.JSON_NOTATION, JSONJAXBContext.JSONNotation.MAPPE
    props.put(JSONJAXBContext.JSON_ROOT_UNWRAPPING, Boolean.TRUE);
    props.put(JSONJAXBContext.JSON_ARRAYS, new HashSet<String>(1){add("jobs");
    props.put(JSONJAXBContext.JSON_NON_STRINGS, new HashSet<String>(1){add("pa
    this.context = new JSONJAXBContext(types, props);
}

public JAXBContext getContext(Class<?> objectType) {
    return (types[0].equals(objectType)) ? context : null;
}
}

```

Restart the application with:

```
mvn clean compile exec:java
```

Note that clean is optional here.

You should now see the returned JSON with the results shown as an array -- even for a single print job -- and with the tonerRemaining and pages results as numbers.

```
{"status":"Idle","tonerRemaining":25,"jobs":[{"name":"sample.doc","status":"printin
```

JSON Configuration Parameters

If you examine the content of the MyJAXBContextResolver class, you'll notice that it includes the following line:

```
props.put(JSONJAXBContext.JSON_NOTATION, JSONJAXBContext.JSONNotation.MAPPED);
```

The option JSONJAXBContext.JSON_NOTATION specifies the type of JSON notation that Jersey 1.0 will generate from the JAXB beans in the web application. In this case, it specifies that the mapped type of JSON notation will be used. The mapped notation is the default

JSON notation in Jersey 1.0. For this notation, Jersey 1.0 takes a JAXB bean specification such as this:

```
@XmlRootElement
public class StatusInfoBean {

    public String status = "Idle";
    public int tonerRemaining = 25;

}
```

And generates the following JSON data:

```
{"status":"Idle", "tonerRemaining":"25"}
```

The following lines in MyJAXBContextResolver specify configuration properties for the default mapped notation:

```
props.put(JSONJAXBContext.JSON_ROOT_UNWRAPPING, Boolean.TRUE);
props.put(JSONJAXBContext.JSON_ARRAYS, new HashSet<String>(1){{add("jobs");}});
props.put(JSONJAXBContext.JSON_NON_STRINGS, new HashSet<String>(1){{add("pages"); a
```

In addition to these three properties, a fourth configuration property, JSONJAXBContext.JSON_ATTRS_AS_ELEMS is available. [Table 1](#) describes the four configuration properties for the mapped notation.

Table 1: Configuration Properties for the Mapped Notation			
Property	Type	Description	Example
JSONJAXBContext.JSON_ROOT_UNWRAPPING	Boolean	If set to true (default value), root elements corresponding to the XML root element will be stripped out.	Boolean.TRUE
		Contains	

JSONJAXBContext.JSON_ARRAYS	Collection<String>	elements which should be treated as array. Such elements will be delimited with braces.	new HashSet<String>(1){}
JSONJAXBContext.JSON_NON_STRINGS	Collection<String>	Contains elements which should not be delimited with double-quotes.	new HashSet<String>(1){}
JSONJAXBContext.JSON_ATTRS_AS_ELEMS	Collection<String>	Contains attributes which should be encoded as elements in JSON (XML attribute names start with @ otherwise).	new HashSet<String>(1){}

Jersey 1.0 supports two JSON options in addition to JSONJAXBContext.JSON_NOTATION. These are:

- JSONNotation.MAPPED_JETTISON. This specifies the Jettison convention. For the Jettison convention, Jersey 1.0 generates the following JSON data for the StatusInfoBean:

```
{"StatusInfoBean":{"status":"Idle","tonerRemaining":"25"}}
```

Jersey currently supports the JSONJAXBContext.JSON_XML2JSON_NS property to configure namespace mapping for the Jettison convention.

- JSONNotation.BADGERFISH. This specifies the BadgerFish convention. For the BadgerFish convention, Jersey 1.0 generates the following JSON data for the StatusInfoBean:

```
{"StatusInfoBean":{"status":{"$":"Idle"},"tonerRemaining":{"$":"25"}}
```

Further Reading

- [Implementing RESTful Web Services in Java](#)
- [Better JSON Available In Jersey](#)
- [Welcome to Maven](#)
- [Building Web Applications with Maven 2](#)

About the Author

Jakub Podlesak is a member of the Jersey project team. Previously, he participated in the development of Metro, the GlassFish v2 web services stack, as a member of the WS-Policy team.

Category: Enterprise Java technology ::: Tags: [jax-rs](#) [jaxb](#) [jersey](#) [json](#) [rest](#) [restful services](#) [web](#) :::
[Permanent link to this entry](#) :::

« [Using WS-Trust Suppo...](#) | [Main](#) | [Tech Tips Quiz](#) »

Comments:

Thanks for the interesting article. Replacing XML with JSON certainly makes network payload lighter and hence better scalability and response time. But do we have enough parsers out there to read/write JSON? What about environments which doesn't understand JavaScript?

Posted by [DominoMill](#) on October 30, 2008 at 03:04 AM PDT <#>

Hi DominoMill, you can look at the second half of <http://www.json.org/> page for a list of languages/libraries supporting JSON. And if you are happy with Jersey, you can use it for processing JSON also on the client side with Jersey client API (http://blogs.sun.com/sandoz/entry/jersey_client_api)

Posted by [Jakub](#) on October 31, 2008 at 02:34 AM PDT <#>

@DominoMill: JSON is important because it is easier for JavaScript clients to consume, namely JavaScript engines embedded in browsers making AJAX calls.

I really cannot say what the performance differences are between producing and consuming JSON and XML. It depends on so many factors. So i would not base my pick on JSON over XML based on performance unless i had empirical data for my particular application indicating that JSON was faster than XML.

Paul.

Posted by **Paul Sandoz** on October 31, 2008 at 05:55 AM PDT <#>

There is a typo:

```
{"status":"Idle","tonerRemaining":"25",  
"jobs":[{"name":"sample.ps","status":"printing...","pages":"13"}, {"name":"second.pdf","st  
in queue","pages":"2"}]}
```

There are suppose to be two job objects attached to the jobs property. Looks like a '{' is missing.

Posted by **Dan D** on November 04, 2008 at 03:54 AM PST <#>

Thanks for pointing out the typo. It's now fixed.

Posted by **Edward Ort** on November 04, 2008 at 08:25 AM PST <#>

@Paul: I didn't mention 'performance' either. XML is more verbose than JSON, hence the message size in bytes is higher. That means it will take longer to push it over network, means more waiting time for user. So the gain is on 'network' side, not on 'CPU'.

Posted by **DominoMill** on November 05, 2008 at 03:24 AM PST <#>

The API for configuring JSON in Jersey has changed with recently released 1.0.2 version. Please see http://blogs.sun.com/japod/entry/configuring_json_for_restful_web for details.

Posted by **Jakub** on February 12, 2009 at 10:33 PM PST <#>

I have tried everything that I can think of, and when it translates to JSON, it does not use the JSONJAXBContext created. I made my web service class extend PackagesResourceConfig and implement ContextResolver, and I can see that it gets picked up as a @Provider when it deploys, and the methods run fine, but nothing I do changes the JSON output, and print statements for the Context never show. Please tell me what I am missing, as I would really like to get the more "natural" JSON output.

Posted by **Gene** on April 09, 2009 at 11:16 AM PDT <#>

@Gene: Thanks for trying things out. The tip was published almost half a year ago and covers Jersey 1.0. Currently available Jersey archetypes use Jersey version 1.0.2. This might be the reason, why things do not work for you, even if i tried to keep backward compatibility. From your comment, it is hard to say, what concrete thing is wrong. You might want to try the example at <http://download.java.net/maven/2/com/sun/jersey/samples/json-from-jaxb/1.0.2/json-from-jaxb-1.0.2-project.zip> to get something working out of the box. If you still facing issues, please let me know at users@jersey.dev.java.net mailing list. I would be happy to help you.

Posted by **Jakub** on April 13, 2009 at 06:40 PM PDT <#>

Hi,
I m a newbie.My question can I pass a xml string as a argument to @GET method.I tried using a xml string but it dint let me do it.though it accepts a simple string.Can I use JSON object instead of an XML?Can you suggest some links where I can take a look at the samples.

Thanks in advance,
Ravi

Posted by **ravindran** on July 22, 2009 at 05:06 AM PDT <#>

@Ravi: You can download a number of Jersey examples as a zip archive at <http://download.java.net/maven/2/com/sun/jersey/samples/jersey-samples/1.1.1->

[ea/jersey-samples-1.1.1-ea-project.zip](#) . JSON could be used instead of XML (see the json-from-jaxb example) but i do not understand, how you use that in the context of GET method, where entity body is not allowed.

Posted by [Jakub](#) on July 22, 2009 at 04:58 PM PDT <#>

thank you for your article...that is a nice one...i often visit this blog and i enjoy reading all posts...

Posted by [dizi izle](#) on January 22, 2010 at 06:50 PM PST <#>

Nice article.

One point i would like to add is, if the bean's field is annotated like @XmlElement(name = "Jobs"), then you should use this name while setting the JSONJAXBContext.JSON_ARRAYS property.

Posted by [Dan](#) on October 19, 2010 at 09:58 PM PDT <#>

Post a Comment:

Comments are closed for this entry.

The views expressed on this blog are those of the author and do not necessarily reflect the views of Oracle. [Terms of Use](#) | [Your](#)