JSON Support

JSON Overview

JSON is a textual data format for data exchange. JSON stands for Javascript Object Notation and, as the name implies, is itself Javascript. Here is a small sample:

```
{
  "customer" : {
    "name" : "Jane Doe",
    "company" : "Acme Enterprises"
}
```

One of the advantages of JSON is that is very easy for Javascript developers to use - it simply needs to be evaluated and it immediately becomes a javascript object. For more information on this see the JSON website.

JSON is supported in CXF through Jettison. Jettison is a StAX implementation that reads and writes JSON. Jettison intercepts calls to read/write XML and instead read/writes JSON.

JSON/XML Conventions

To understand how to create a JSON service, you first need to understand how your XML structures will be converted into the JSON format. JSON does not have many concepts found in XML such as namespaces, attributes or entity references. Because of this we must adopt *conventions* on how to convert between the two.

Jettison supports two conventions currently. These are explained in more detail in the Jettison user's guide, but a quick summary of each follows.
1. The "mapped" convention. In this convention namespaces are mapped to json prefixes. For instance, if you have a namespace "http://acme.com" you could map it to the "acme" prefix. This means that the element <customer xmlns="http://acme.com"> would be mapped to the "acme.customer" JSON tag.
2. The BadgerFish convention. This convention provides a mapping of the full XML infoset to JSON. Attributes are represented with an @ sign - i.e. "@attributeName" : "value". Textual data is represented with the "$" as the tag. Example:

```
{ "customer" : { "name" : { "$" : "Jane Doe" } } }
```

Configuring Jettison

Creating a Jettison service is like creating any other service, except that you set a couple extra properties on your service's endpoint. First you'll want to create a service and make sure it is working with XML before continuing here. Currently its recommended that you use the HTTP Binding for your JSON endpoints as javascript developers will be more familiar with the RESTful style of interaction. It is also easier for javascript developers as they will not have to formulate a request for ever interaction.

Using ServerFactoryBeans

This example shows how to set up Jettison using a ServerFactoryBean, such as the JaxWsServerFactoryBean. First you must create a properties HashMap and set the StAX XMLInputFactory and XMLOutputFactory:

```
Map<String,Object> properties = new HashMap<String,Object>();

// Create a mapping between the XML namespaces and the JSON prefixes.
// The JSON prefix can be "" to specify that you don't want any prefix.
HashMap<String, String> nstojns = new HashMap<String,String>();
nstojns.put("http://customer.acme.com", "acme");

MappedXMLInputFactory xif = new MappedXMLInputFactory(nstojns);
properties.put(XMLInputFactory.class.getName(), xif);

MappedXMLOutputFactory xof = new MappedXMLOutputFactory(nstojns);
properties.put(XMLOutputFactory.class.getName(), xof);
```

You must also tell CXF which Content-Type you wish to serve:

```
// Tell CXF to use a different Content-Type for the JSON endpoint
// This should probably be application/json, but text/plain allows
// us to view easily in a web browser.
properties.put("Content-Type", "text/plain");
```

Last, you'll want to actually create your service:

```
// Build up the server factory bean
JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
sf.setServiceClass(CustomerService.class);
// Use the HTTP Binding which understands the Java Rest Annotations
sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
sf.setAddress("http://localhost:8080/json");
sf.setServiceBean(new CustomerServiceImpl());

sf.setProperties(properties);

sf.create();
```