

# **Web Services Server for Niagara AX Use Guide**

Version 1.0  
2009-11-23

## DESCRIPTION

This module can be used to turn a Niagara AX station into a web-services server that understands SOAP requests and is capable of serving WSDL service descriptions. It has a set of extensions for making data points readable and writeable via SOAP-based operations. It also contains tools for running custom web services with a more complex behavior.

The module is written in Java 1.5 SE, so it will run only on the JACEs that support that version of the language (soft JACEs, workbenches, and JACEs with Niagara 3.5+).

## SETUP

After you've loaded `korsWsServer` module to the station, you need to drop Server component into a folder under the "config" category. You also need to change the component's Servlet Name to some meaningful value (we will use "services" throughout this guide). The Server will act as a point of entry for all the web service calls. It will be accessible via addresses like <http://127.0.0.1:80/services> where `127.0.0.1` is the IP or host name of the Niagara station, `80` is the port number of the station's web server and `services` is the servlet name you entered for your Server component. If you open that page in a browser, you should see the WSDL document that describes the web services available on that particular station.

It's worth noting that web services use Niagara's web server, so their behavior, performance and security depend on the way it's configured. `Config/Services/WebService` should allow you to change HTTP ports and see whether the web server is enabled at all. `Config/Services/UserService` is the component that is going to control authentication.

## USING EXTENSIONS

`korsWsServer` comes with a three-point extensions: `WSBooleanExtension`, `WSStringExtension` and `WSNumericExtension`. Each of them can be used to make a control point of the appropriate type readable and writeable from the web. To do that, you would simply open `korsWsServer` palette and drag-and-drop the icon of the extension to the target component (either in Nav panel or in Property Sheet view).

Each extension has an "Id Format" field. It is a standard Baja format field and can use Niagara's formatting syntax. Once passed by the component, the field content will resolve to a string displayed in the "id" slot. Whatever you see there is the token the extension's parent point can be referred to in SOAP requests.

When you add the first extension, it will generate two operations depending on the type. For example, `WSBooleanExtension` will create `SetBooleanPoint` and `GetBooleanPoint` operations. Those operations and all the message types used with them will appear in server-generated WSDL file.

## CREATING CUSTOM WEB SERVICES

Creating a custom web service is done by:

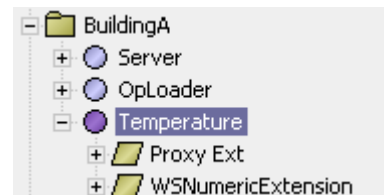
1. Extending `com.korsengineering.ws.microwss.ComplexType` class to create request and response message types.
2. Extending `com.korsengineering.ws.microwss.Operation` to define operation logic using the types created.
3. Packaging the resulting classes as a Niagara module.
4. Loading the operation and registering it with the Server component.

An example for #1 and #2 can be found in sample implementation of [MicroWSS](#). To make the last step easier, `korsWsServer` package provides `OpLoader` component that automatically loads and registers web-service operations with the Server. All you have to do is `instantiate` one (doesn't matter where on the station it resides), and supply it with a module name and a full class name of the operation you wrote. If the operation is successfully loaded, `OpLoaders`'s "Loaded" slot will be set to true. The new operation and messages will be described in Server's WSDL.

## EXTENSIONS TUTORIAL

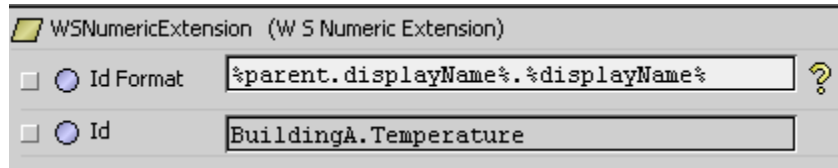
This tutorial will walk you through a process of making one `NumericWriteable` point web-enabled. We assume that the target point is `Configs/BuildingA/Temperature`. We also assume that the station in question is accessible via 127.0.0.1 IP. The tutorial ignores everything related to authentication, since it is covered in Niagara documentation.

1. Install `korsWsService` module on the station via platform software manager.
2. Open the palette for `korsWsService` in the palette area of the workbench.
3. If you haven't done this before, drag and drop `Server` component to `Configs/BuildingA` folder on the station. (It could be any folder within the station's hierarchy.) Make sure that its `Enabled` field is set to true. Also, type "wss" into `Servlet Name` field and save the component configuration.
4. Go to `Configs/Services/WebService` and take a note of the `Http Port` attribute. We will assume it is 81.
5. Open <http://27.0.0.1:81/wss> in any browser. You should see some XML there. (To be more precise, you should see a WSDL document that doesn't yet list any operations.)
6. Drag and drop `WDNumericExtension` to the `Temperature` component. After that, <http://27.0.0.1:81/wss> should change to something similar to code listing 1. It now has two operations (`GetNumericPoint`, `SetNumericPoint`) and four message types (`GetPointRequest`, `GetNumericPointResponse`,



SetNumericPointRequest, SetPointResponse).

- Open the property view of the extension you've just created. It should have Id Format equal to “%parent.displayName%.%displayName%” (the default value). That resolves to “BuildingA.Temperature”. The resolved id string is how components are registered with the Server, and so the ids’ should be unique among the components of the same type.



- At this point, you should be able to use a SOAP client (or anything that is capable of sending an HTTP POST response with the appropriate body) to get and set Temperature. You could, for example, test it with [SoapUI](#). Set the point value to 11 in the workbench. Code listings 2 and 3 show the request you can send and the response you will get.

Using Niagara's format field to generate point ids makes the extensions extremely flexible without adding much complexity to their use. The ids are regenerated on copy, so if you copy BuildingA folder and name the copy BuildingB, the copy of the Temperature inside will be immediately accessible as BuildingB.Temperature.

## CODE LISTINGS

### Code listing 1: WSDL after adding a numeric point extension to the system

```
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="targetNs" xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" targetNamespace="targetNs"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
<s:schema targetNamespace="targetNs" elementFormDefault="qualified">
<s:element name="GetNumericPointRequest" type="tns:GetPointRequest"/>
<s:element name="GetNumericPointResponse" type="tns:GetNumericPointResponse"/>
<s:element name="SetNumericPointRequest" type="tns:SetNumericPointRequest"/>
<s:element name="SetNumericPointResponse" type="tns:SetPointResponse"/>
<s:complexType name="SetPointResponse">
<s:sequence>
<s:element name="error" type="s:string"/>
</s:sequence>
</s:complexType>
<s:complexType name="GetNumericPointResponse">
<s:sequence>
<s:element name="value" type="s:double"/>
<s:element name="error" type="s:string"/>
</s:sequence>
</s:complexType>
<s:complexType name="SetNumericPointRequest">
<s:sequence>
```

```

    <s:element name="pointId" type="s:string"/>
    <s:element name="value" type="s:double"/>
  </s:sequence>
</s:complexType>
<s:complexType name="GetPointRequest">
  <s:sequence>
    <s:element name="pointId" type="s:string"/>
  </s:sequence>
</s:complexType>
</s:schema>
</types>
<message name="GetNumericPointIn">
  <part name="parameters" element="tns:GetNumericPointRequest"/>
</message>
<message name="GetNumericPointOut">
  <part name="parameters" element="tns:GetNumericPointResponse"/>
</message>
<message name="SetNumericPointIn">
  <part name="parameters" element="tns:SetNumericPointRequest"/>
</message>
<message name="SetNumericPointOut">
  <part name="parameters" element="tns:SetNumericPointResponse"/>
</message>
<portType name="myport">
  <operation name="GetNumericPoint">
    <input message="tns:GetNumericPointIn"/>
    <output message="tns:GetNumericPointOut"/>
  </operation>
  <operation name="SetNumericPoint">
    <input message="tns:SetNumericPointIn"/>
    <output message="tns:SetNumericPointOut"/>
  </operation>
</portType>
<binding name="soap12binding" type="tns:myport">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetNumericPoint">
    <soap12:operation soapAction="thisSoapAction" style="document"/>
    <input>
      <soap12:body use="literal"/>
    </input>
    <output>
      <soap12:body use="literal"/>
    </output>
  </operation>
  <operation name="SetNumericPoint">
    <soap12:operation soapAction="thisSoapAction" style="document"/>
    <input>
      <soap12:body use="literal"/>
    </input>
    <output>
      <soap12:body use="literal"/>
    </output>
  </operation>

```

```
</operation>
</binding>
<service name="WebService">
  <port name="WebServiceSoap12" binding="tns:soap12binding">
    <soap12:address location="http://127.0.0.1:81/wss"/>
  </port>
</service>
</definitions>
```

### Code listing 2: sample request

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:tar="targetNs">
  <soap:Header/>
  <soap:Body>
    <tar:GetNumericPointRequest>
      <tar:pointId>BuildingA.Temperature</tar:pointId>
    </tar:GetNumericPointRequest>
  </soap:Body>
</soap:Envelope>
```

### Code listing 3: sample response

```
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetNumericPointResponse xmlns="http://korsengineering.com/targetNs">
      <value>11.0</value>
    </GetNumericPointResponse>
  </soap12:Body>
</soap12:Envelope>
```