Demonstrating the Yahoo! Geocoder with GlassFish ESB V 1.2

thomas.barrett@sun.com
June 22, 2009

As a result of some proof-of-concept (POC) experience, I had a chance to explore interfacing with the Yahoo! Geocoder RESTful service from GlassFish ESB. My colleague Serge Blais (http://blogs.sun.com/sblais/) led the way on this part of the POC. I enjoyed learning from him and decided to write up a bit of what I think I learned. Serge used Approach 1 discussed in this document. After some pondering, I explored an alternative that I documented as Approach 2.

Demonstration Scenario:

You have database records that contain address information and you want to geocode these records by capturing longitude and latitude for each. Yahoo! Maps Web Services provides a Geocoding API. This RESTful API allows you to programmatically lookup latitude and longitude for any address. You can read about it at http://developer.yahoo.com/maps/rest/V1/geocode.html and feel that it offers a solution.

The service is available, free of charge, at: http://local.yahooapis.com/MapsService/V1/geocode. All you need to do is to register with Yahoo! to receive a free appID. Then, you can issue RESTful queries like:

http://local.yahooapis.com/MapsService/V1/geocode?
appid=[PutYourAppIDHere]&street=701+First+Ave&city=Sunnyvale&state=CA

And you'll receive XML back like this giving you the geocoding (latitude and longitude) for the specified location:

```
−<ResultSet xsi:schemaLocation="urn:yahoo:maps http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  −<Result precision="address">
      <Latitude>37.416397</Latitude>
      <Longitude>-122.025055</Longitude>
      <Address>701 1st Ave</Address>
      <City>Sunnyvale</City>
      <State>CA</State>
      <Zip>94089-1019</Zip>
      <Country>US</Country>
    </Result>
</ResultSet>
```

Your application can extract the latitude and longitude and store it along with the database address record or use it to plot data points in a mashup, for example. The task in this demonstration is to show how to interface with the Yahoo! service using GlassFish ESB.
We will explore two approaches. Both approaches create a SOAP web service that wraps the HTTP interaction to the RESTful Yahoo! web service. This shields your SOAP-focused, SOA developers

from having to deal with both REST. We can foster an "all SOAP shop." The SOAP web service we create here is simply a facade to hide the RESTful interaction.

The first approach (Approach 1) creates a solution featuring JBI. The BPEL service engine hosts the logic and the HTTP binding component (BC) provides connectivity to Yahoo!. This approach will be light on hardcore programming in favor of using the BPEL, WSDL and XSD editors in the NetBeans toolset. The second approach (Approach 2) creates a web service implemented as as a stateless EJB. This approach is more of a traditional Java EE and web services coding exercise. At the end of the document, I'll briefly comment on the pros and cons of the two approaches.

Environment:

While last reviewing my notes on the both approaches, I used the GlassFish ESB v2.1 build on Windows XP available from https://openesb.dev.java.net/Downloads.html.

Approach 1: Using JBI: BPEL SE and HTTP BC

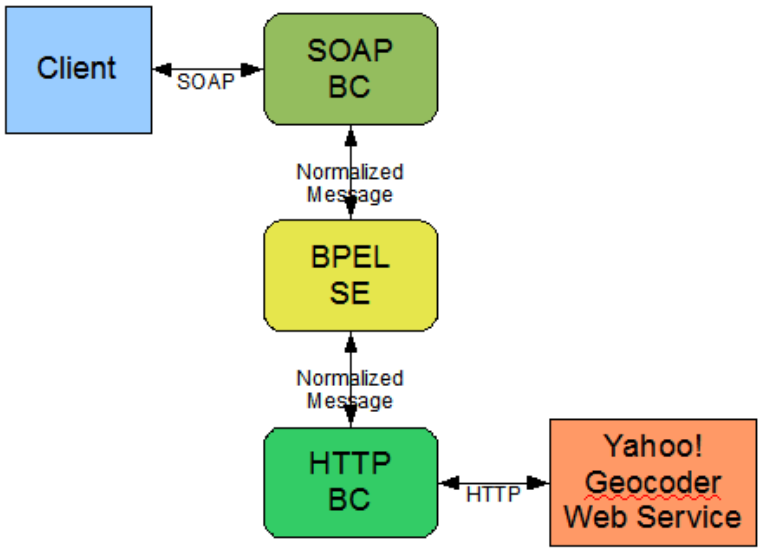
Overview:

At the outset of describing Approach 1, credit needs to go out to my colleague, Serge Blais (http://blogs.sun.com/sblais/), who worked with me on a proof-of-concept (POC) project at a customer site in early 2009. He worked up a demo showing how to build this BPEL-based solution and posted a screen cast here:
http://mediacast.sun.com/users/Serge_Blais/media/JBI_YahooGeoCode/details

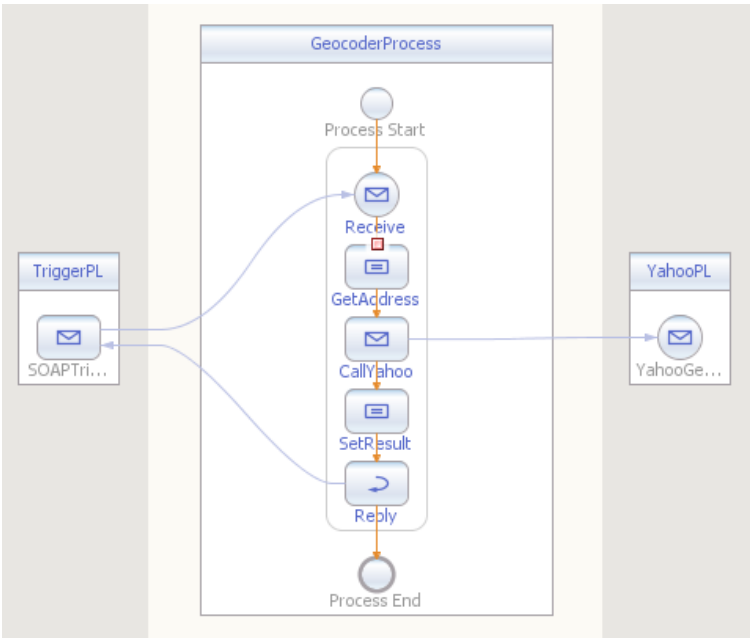
In this screencast, he shows you how to implement Approach 1. The notes that follow were inspired by what he did during the POC and by what he documented in his screen cast. In what follows, I took some technical writer license to reorder/elaborate/embellish his steps.

Here we'll use the HTTP binding component to provide the HTTP proxy to Yahoo!. We'll invoke it from a BPEL process that provides the SOAP web service wrapper. We'll expose the completed BPEL process via a SOAP WSDL.

The illustration below sketches out the approach. The BPEL service engine (SE) will host our BPEL process that will provide a "SOAP wrapper" that invokes the Yahoo! RESTful web service. The Client provides the trigger for the BPEL process by sending a SOAP message through the SOAP binding component (BC) communication proxy. The BPEL SE uses the HTTP BC to send the request to Yahoo! and receive the response bacl. The BPEL SE processes the returned geocoding and delivers the result it to the Client via the SOAP BC:



Here is the completed BPEL process that will be hosted by the BPEL SE:



The TriggerPL partner link on the left provides the SOAP interface to initiate the BPEL process and consume the result. This is hosted at runtime by the HTTP binding component. The YahooPL partner link at the right provides the HTTP interface to interact with the Yahoo! Geocoder RESTful web service. This communication proxy is hosted by the HTTP BC at runtime. Of course, the BPEL process itself runs in the BPEL service engine.

Demonstration:

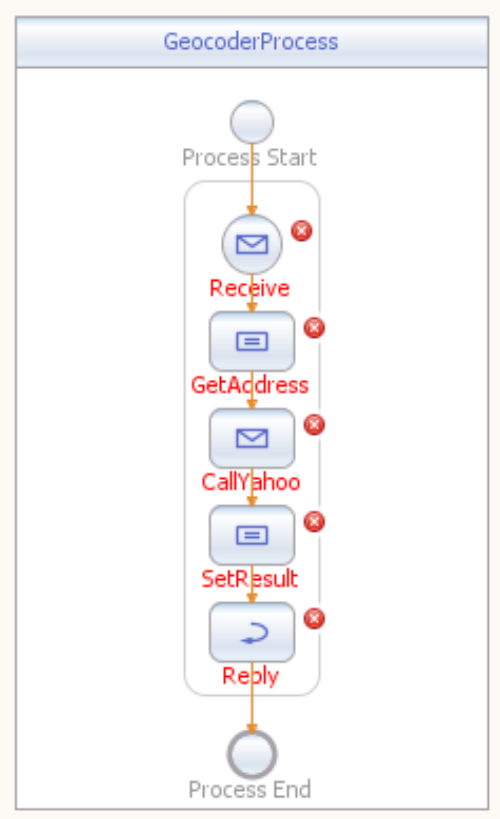
Get a Yahoo! Geocoder app id
- Go here: https://developer.yahoo.com/wsregapp/ and fill out the form
- Choose "Generic, No user authentication required" for Authentication method
- At end of registration, you will receive your Application ID. Save it in a text document so we can copy and paste it as needed.

Create a new SOA / BPEL module. Call it GeocoderBpelModule
- Project Location: C:\GeocoderBPELDemo

Let's sketch out our BPEL process so we have a framework within which to do all the detailed work.

Create a new BPEL process called GeocoderProcess

Add a receive, two assigns, invoke and reply and relabel as shown:

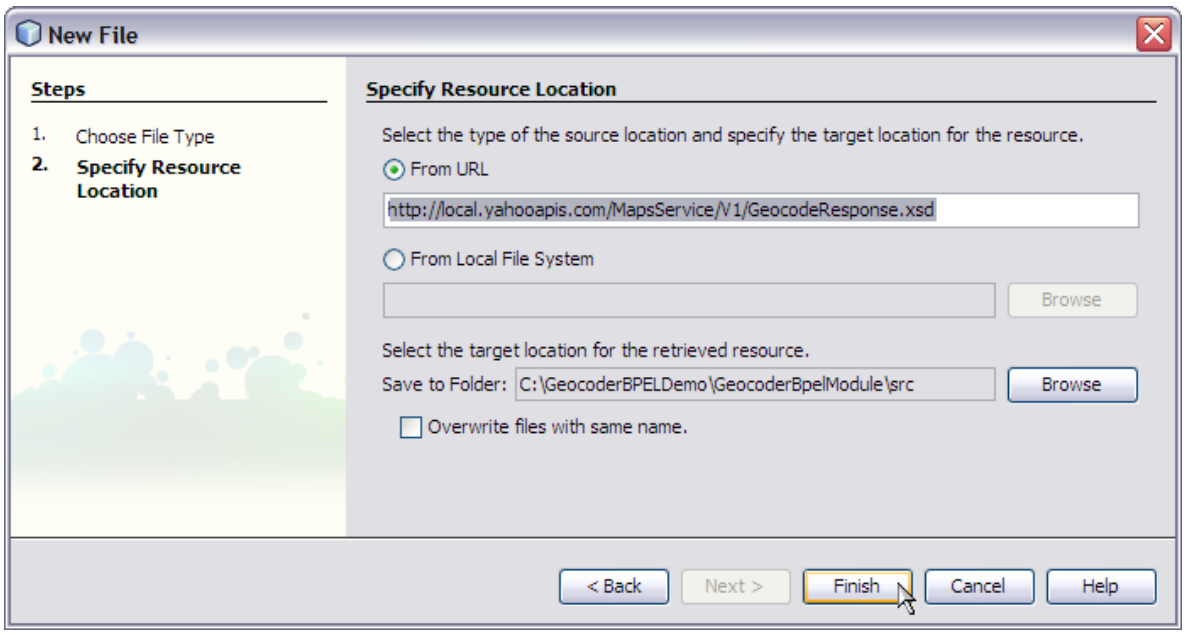


Our initial focus is to call out to Yahoo! Gecoder so let's work on the CallYahoo invoke first.

We need to create a new WSDL to describe the interface to the the Yahoo Geocoder service. We need to have access to the data structure that the Geocoder will return. We want to use that data structure definition in our WSDL construct, so we need to create an XML Schema Definition (XSD) for it.

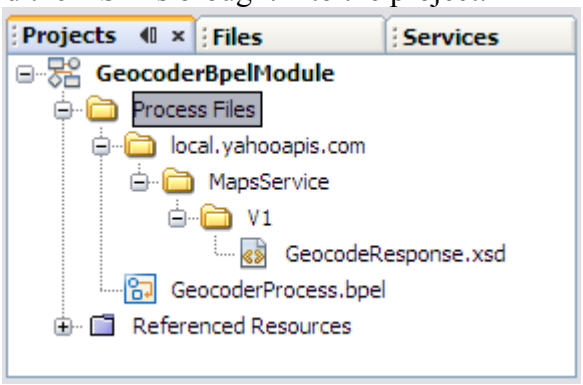
Fortunately, Yahoo! posts this on the web. We can just create the XML by providing the URL (see screen capture below). The XSD is "advertised" at: http://developer.yahoo.com/maps/rest/V1/geocode.html

Create a new XSD. Right-click on GeocoderBPELModule and select New / Other / XML / External XML Schema Document(s):



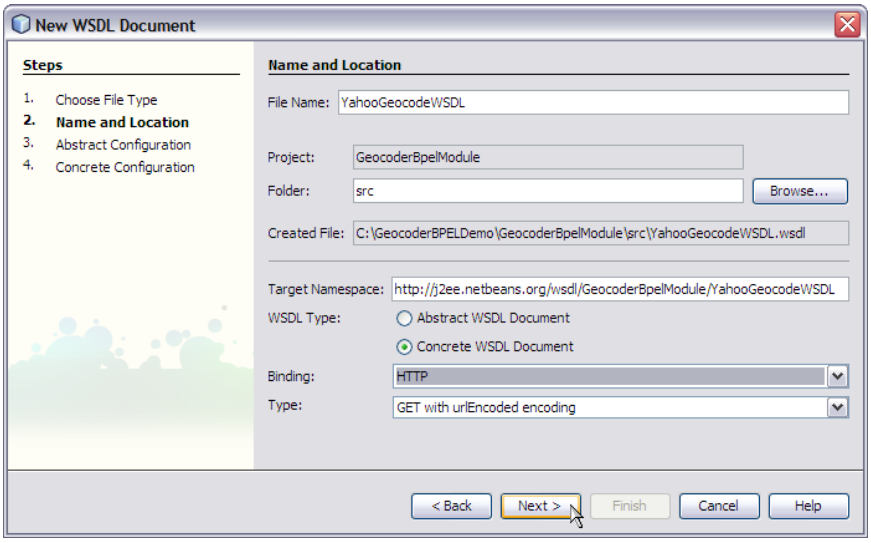
The URL is: http://local.yahooapis.com/MapsService/V1/GeocodeResponse.xsd

The Yahoo site is accessed and the XSD is brought into the project:



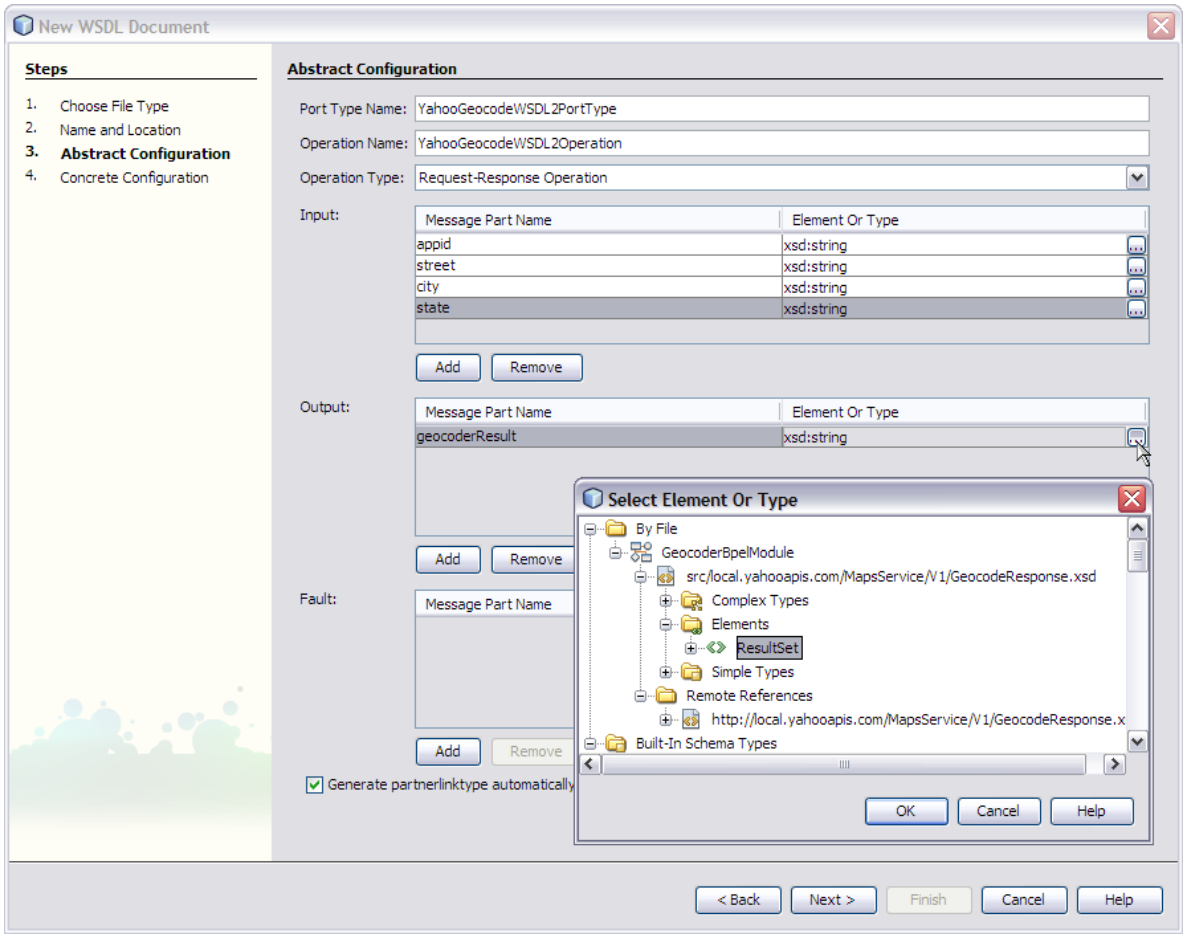
Now, create a new WSDL document to describe the interface to the the Yahoo Geocoder service. We will use an HTTP binding resulting in the HTTP BC being the host of this communication proxy. Name the WSDL: YahooGeocodeWSDL

Specify HTTP / Get with urlEncoded as the concrete binding:

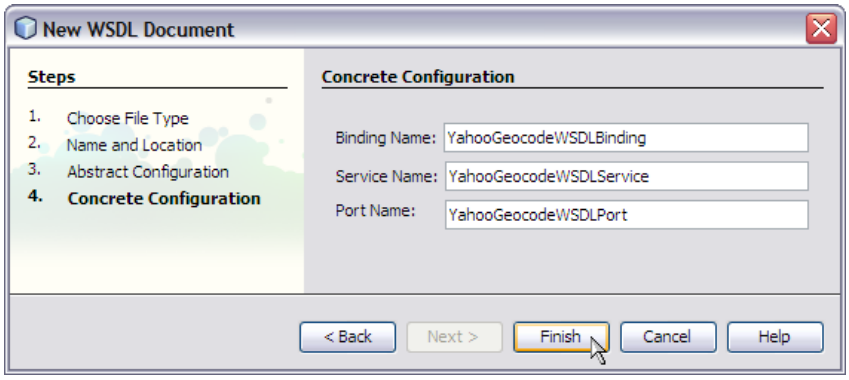


Specify a four-part input including (all of type string): appid, street, city, state

Define a one-part output called geocoderResult of type ResultSet from the GeocodeResponse.xsd:



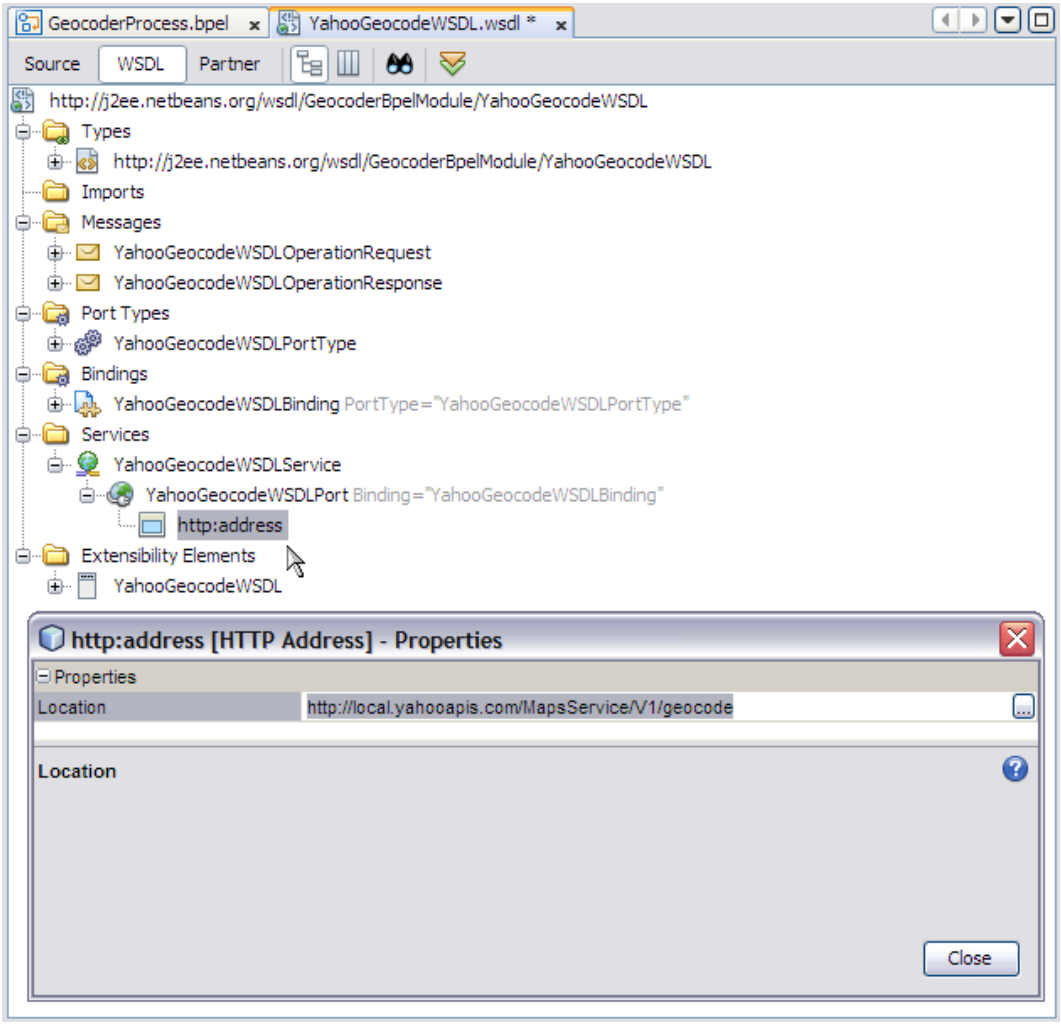
Take the defaults on the next wizard step:



We need to supply the URI where the Yahoo! Gecoder is "listening." At http://developer.yahoo.com/maps/rest/V1/geocode.html, you'll see that the address is:

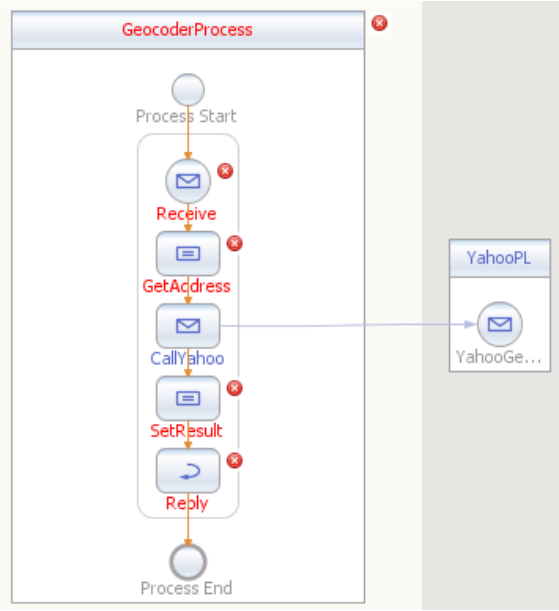
http://local.yahooapis.com/MapsService/V1/geocode

We need to edit the WSDL to reflect this endpoint address:



Save All your work.

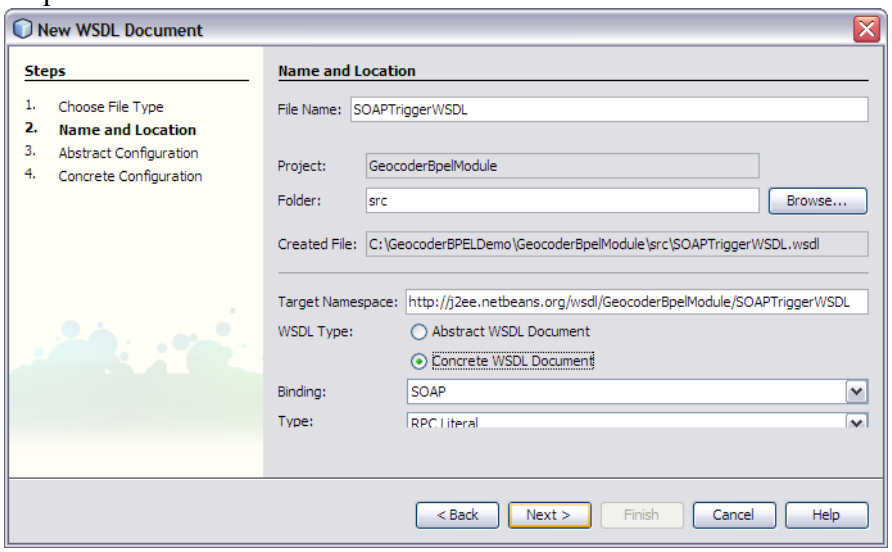
Let's create a partner link through which our CallYahoo invoke can call out to the Yahoo! Geocoder service. Drag and drop the YahooGeocodeWSDL.wsdl onto the right side of the BPEL model to create a partner link called: YahooPL Then, wire it up to the invoke:



We can call out of our BPEL process to access Yahoo!. Now, we need a way to call our BPEL process. We need a SOAP-based WSDL that specifies how to trigger the Receive and deliver the result through the Reply.

Create a new WSDL called SOAPTriggerWSDL

This new WSDL will provide a SOAP "wrapper" (SOAP / RPC Literal) around our BPEL process that we can use to call the process.

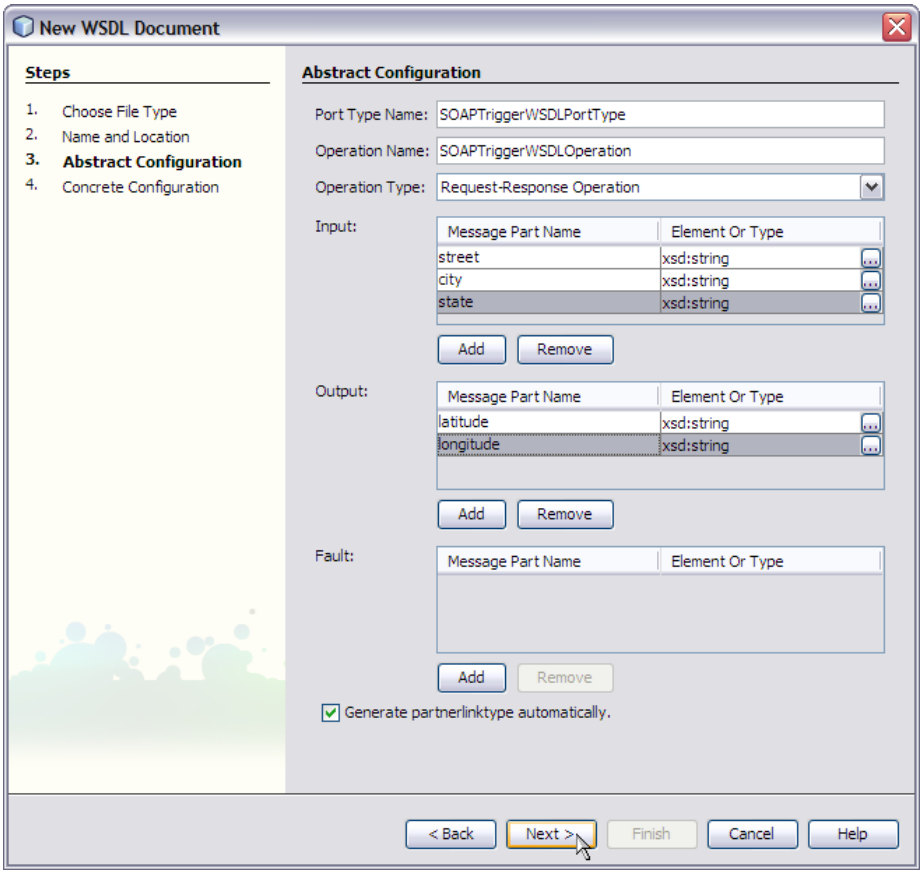


Specify a three-part input including (all of type string):

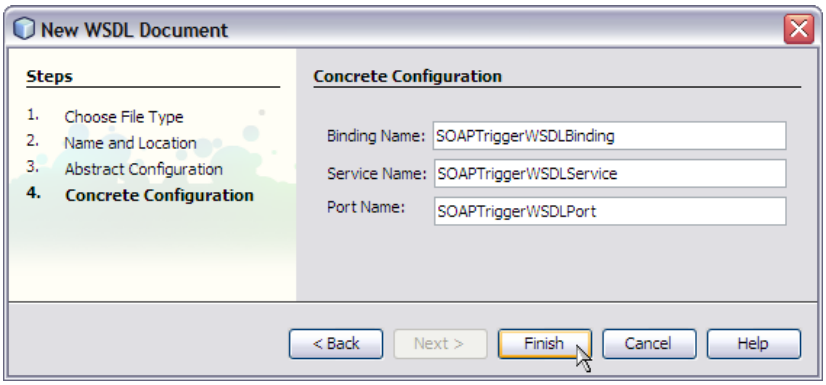
- street
- city
- state

Specify a two-part output including (all of type string):

- latitude
- longitude

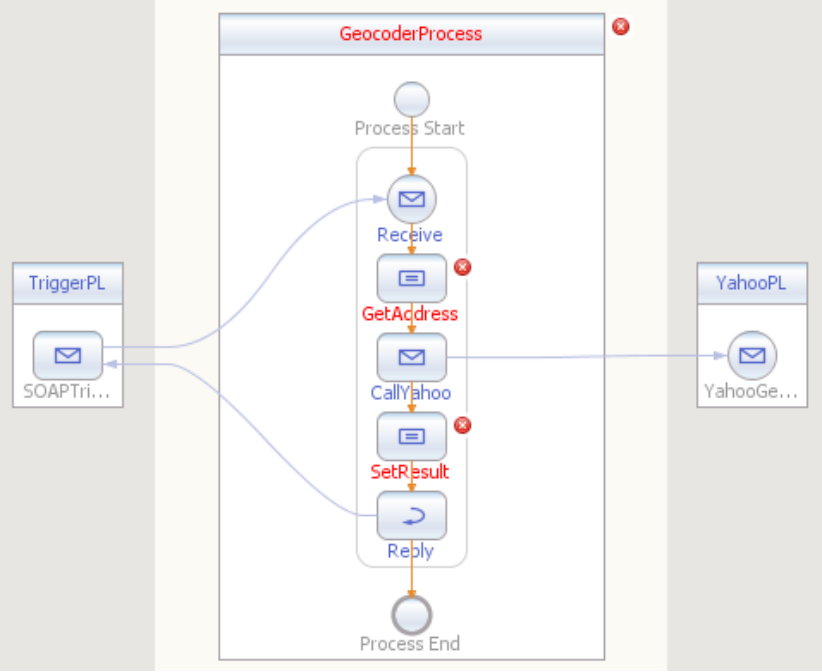


Take the defaults on the next wizard step:



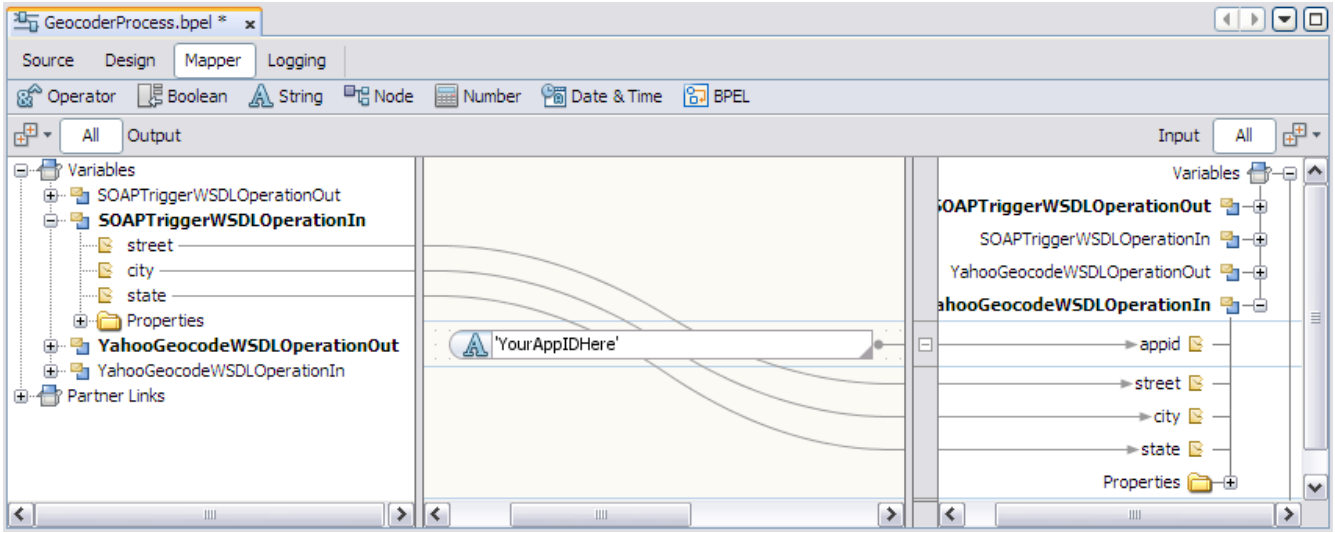
Drag the SOAPTriggerWSDL.wsdl onto the left side of the process to create a new partner link called TriggerPL

Wire up the new partner link to the Receive and Reply elements:



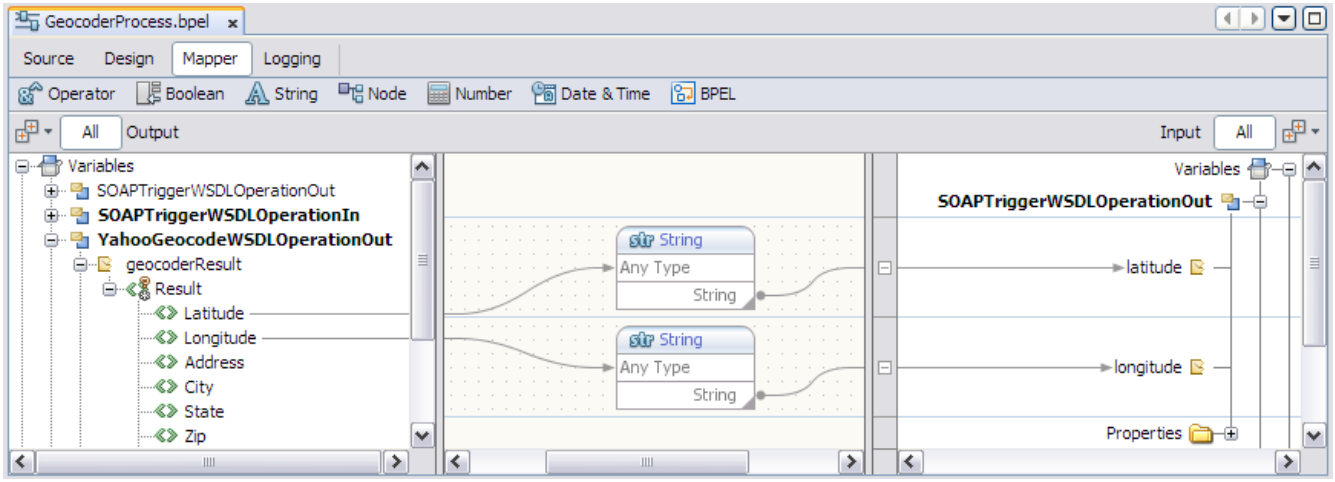
Now, we need to specify mappings for the GetAddress assign so that the inputs provided through the TriggerPL are mapped to inputs required for the CallYahoo invoke.

Double-click on the GetAddress assign to make these mappings:



Of course, you will replace “YourAppIDHere” with your actual Yahoo appID.

The last step to finish our BPEL process to map the results of the CallYahoo invoke to the response delivered via the TriggerPL. So, double-click on the SetResult assign and make the following mappings:



The String conversions noted above are necessary to "cast" the Latitude and Longitude (which are both of type decimal) to the string equivalents in our SOAP service output.

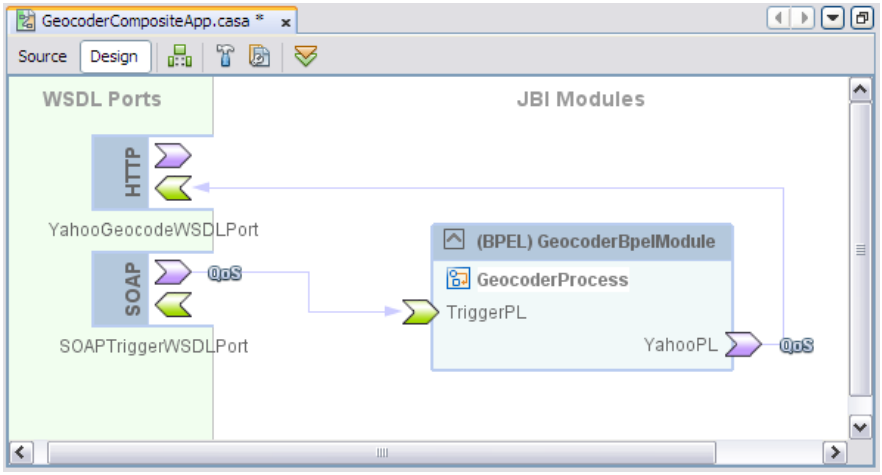
Save All your work.

Clean and Build your GeocoderBpELModule.

Create a new composite application named: GeocoderCompositeApp

When the CASA Editor canvas appears, drag and drop the GeocoderBpelModule into the JBI Modules area.

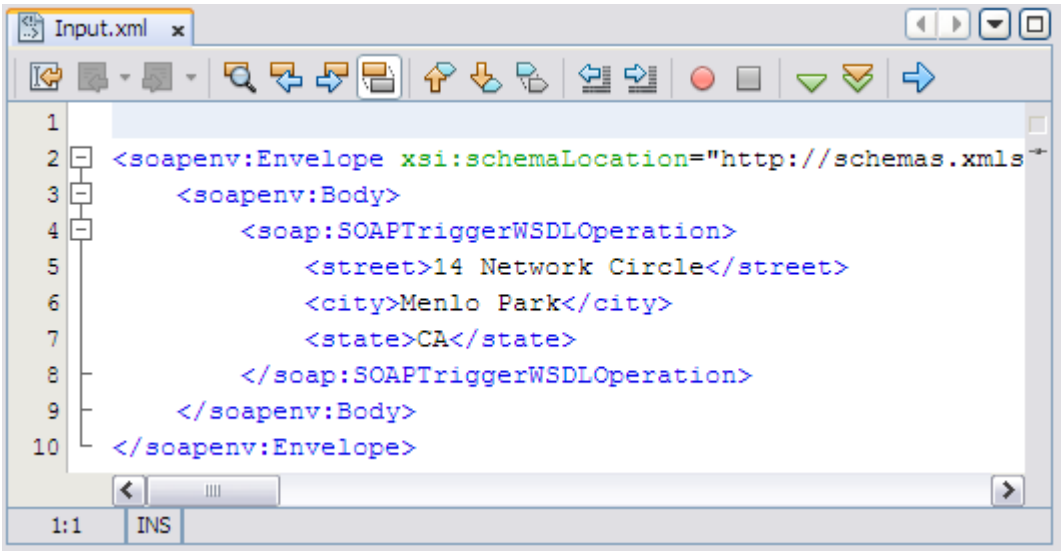
Click the Build Project (hammer) tool at the top of the canvas to build the service assembly:



Click the Deploy Project tool (the wrench) at the top of the canvas to deploy the service assembly to GlassFish.

Create a new Test Case for the GeocoderCompositeApp and specify SOAPTriggerWSDL.wsdl and the SOAPTriggerWSDLOperation as the endpoint to target.

Specify the test case as shown:

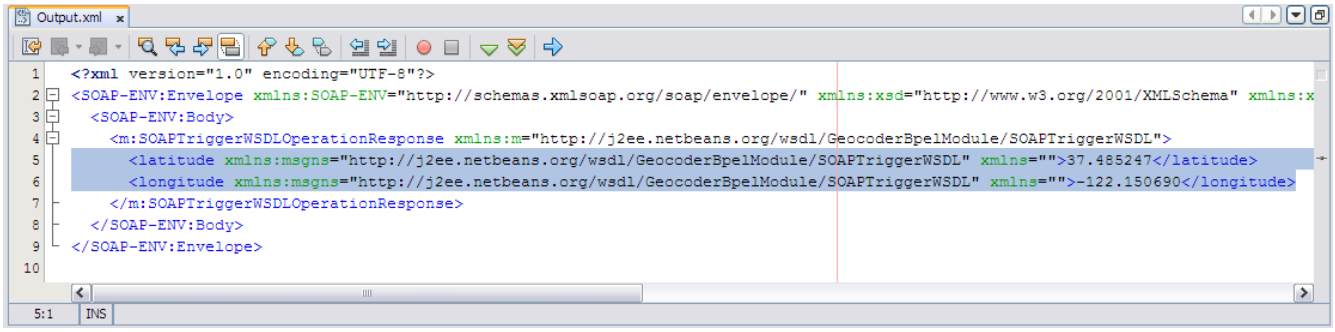


Here is the test address for Sun Microsystems:

<street>14 Network Circle</street>
<city>Menlo Park</city>
<state>CA</state>

Run the test case.

View the returned latitude and longitude in the test case output



Sun Microsystems is at:

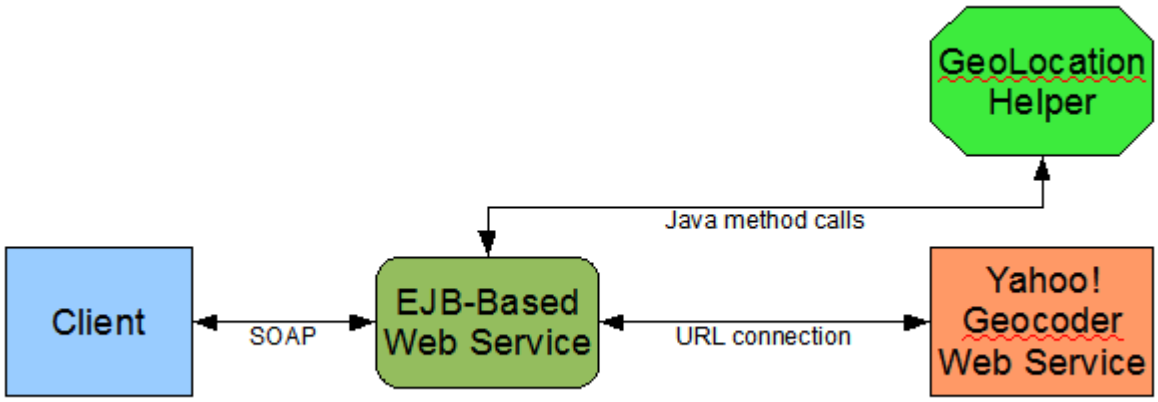
- latitude: 37.485247
- longitude: -122.150690

Approach 2: Using a EJB-Based Web Service

Overview:

In the second approach, there are no JBI service engines or binding components. We just create a stateless EJB-based web service that is invoked via a SOAP message. The EJB implement uses old-fashioned Java programming to open a connection to the Yahoo! Geocoder endpoint, send an HTTP request and consume the response. We make use of the Apache Commons Digester component to parse the returned XML result.

Here is a sketch of Approach 2:



The client will be soapUI for us. The two green components are just two Java classes. The Yahoo! Geocoder, of course, runs out on the Internet at Yahoo!.

Setup:

Make sure the soapUI Web Service Testing plugin is installed in your NetBeans. Use Tools / Plugins / Available Plugins to see if it is installed. If it isn't, install it. We'll use it at the end to test our web service. You'll need to restart your NetBeans in order to complete installation.

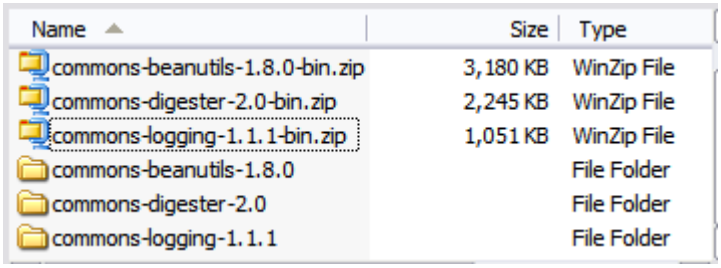
We will need some jar files from the Apache Commons Project. Go to the Apache Commons site: http://commons.apache.org/

Find the Components section and download the following:

- BeanUtils
- Digester
- Logging

Store the zip versions of the binary downloads in C:\GeocodeWebServiceDemo

Extract each of the three zip files:

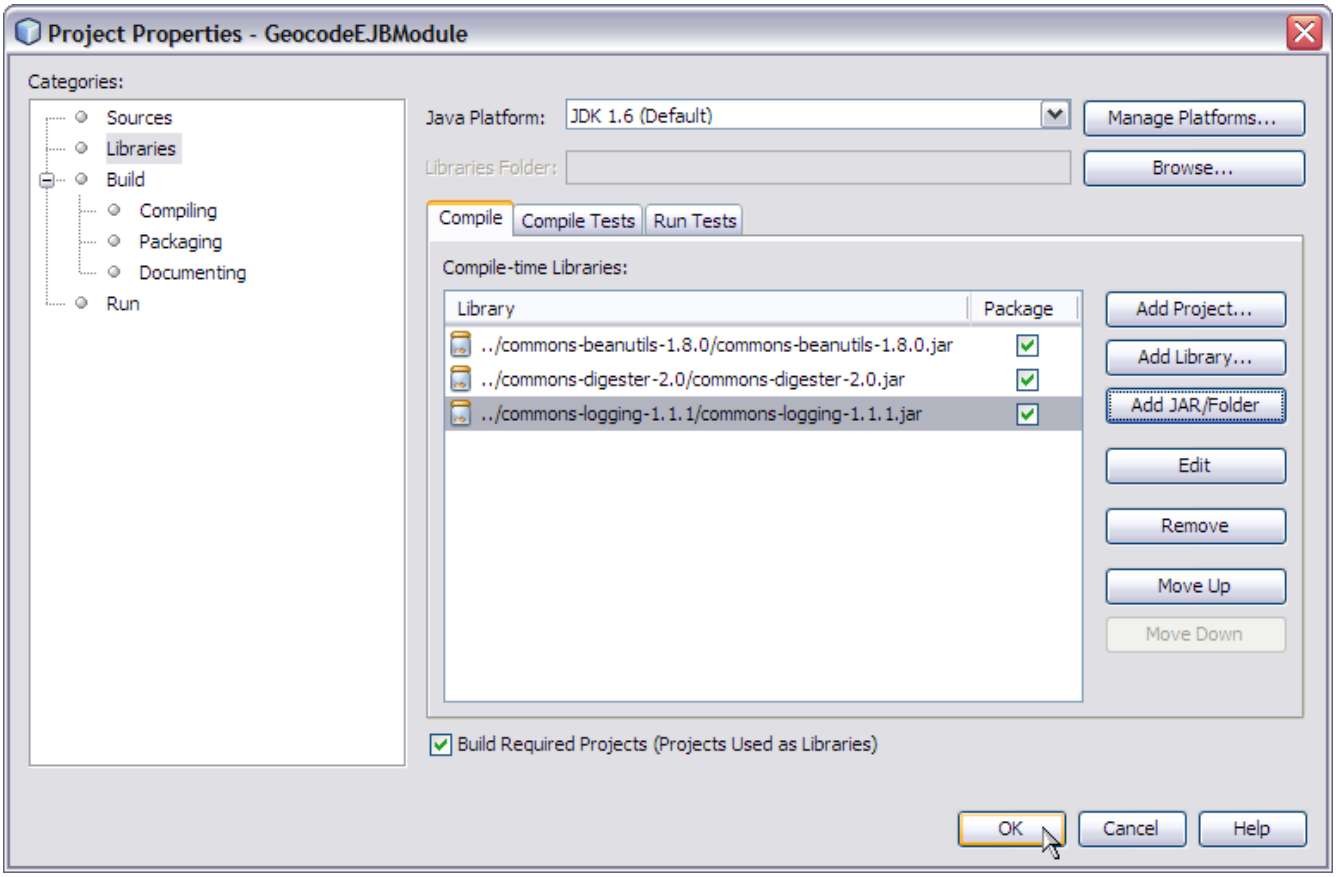


Create a new EJB module called GeocodeEJBModule

- Project Location: C:\GeocodeWebServiceDemo

We need to include these three jar files in the project. Right-click on GeocodeEJBModule and select Properties.

Click on Libraries and use the Add JAR/Folder button to add the three jar files:



Demonstration:

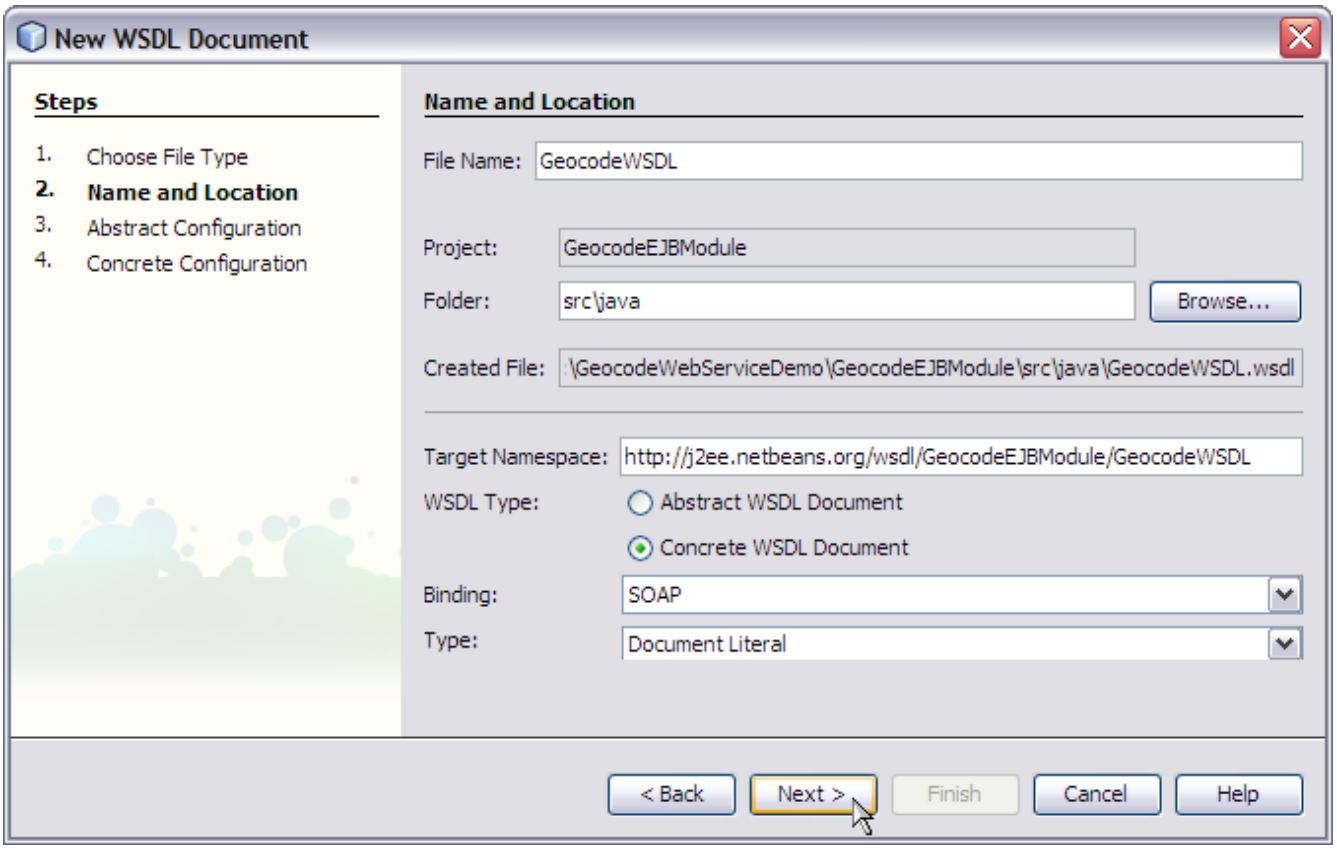
Create an XSD in the GeocodeEJBModule. Name the XSD: GeocodeXMLSchema

In Source mode, replace the skeleton XSD with the following (copy and paste):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xml.netbeans.org/schema/GeocodeXMLSchema"
        xmlns:tns="http://xml.netbeans.org/schema/GeocodeXMLSchema"
        elementFormDefault="qualified">
    <xsd:complexType name="GeoLocationInput">
        <xsd:sequence>
            <xsd:element name="query" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="GeoLocationOutput">
        <xsd:sequence>
            <xsd:element name="Latitude" type="xsd:decimal"/>
            <xsd:element name="Longitude" type="xsd:decimal"/>
            <xsd:element name="Address" type="xsd:string"/>
            <xsd:element name="City" type="xsd:string"/>
            <xsd:element name="State" type="xsd:string"/>
            <xsd:element name="Zip" type="xsd:string"/>
            <xsd:element name="Country" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="precision" type="xsd:string"/>
        <xsd:attribute name="warning" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <xsd:element name="GeoLocationRequest" type="tns:GeoLocationInput"/>
    <xsd:element name="GeoLocationResponse" type="tns:GeoLocationOutput"/>
</xsd:schema>
```

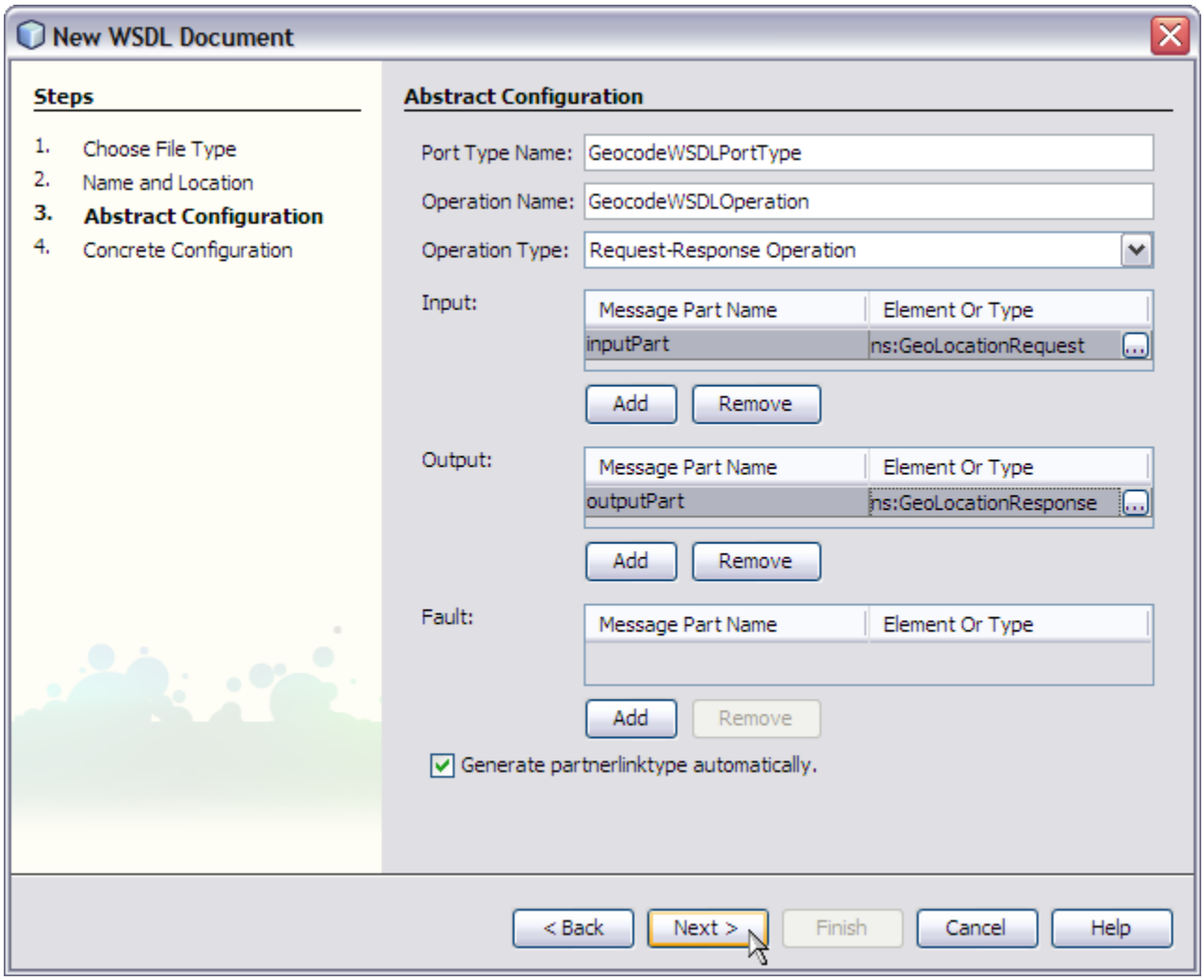
Save All.

Create a WSDL in the GeocoderEJBModule. Call the new WSDL file: GeocodeWSDL



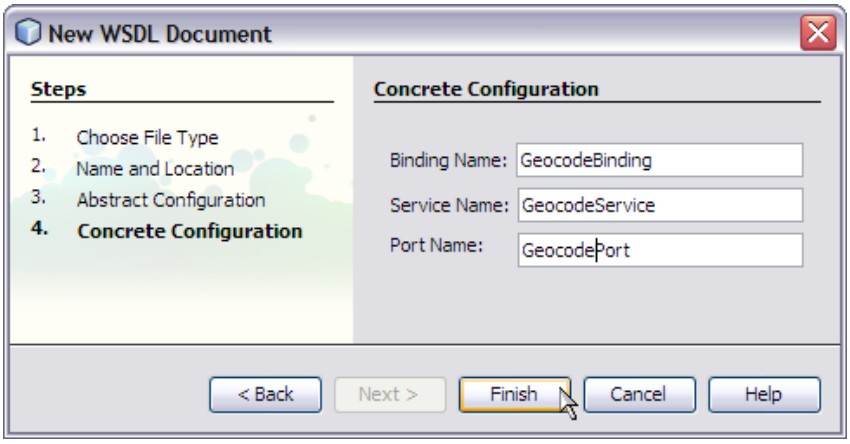
Note that we'll use the Document Literal style for our SOAP message interaction.

Specify the part names and part types:



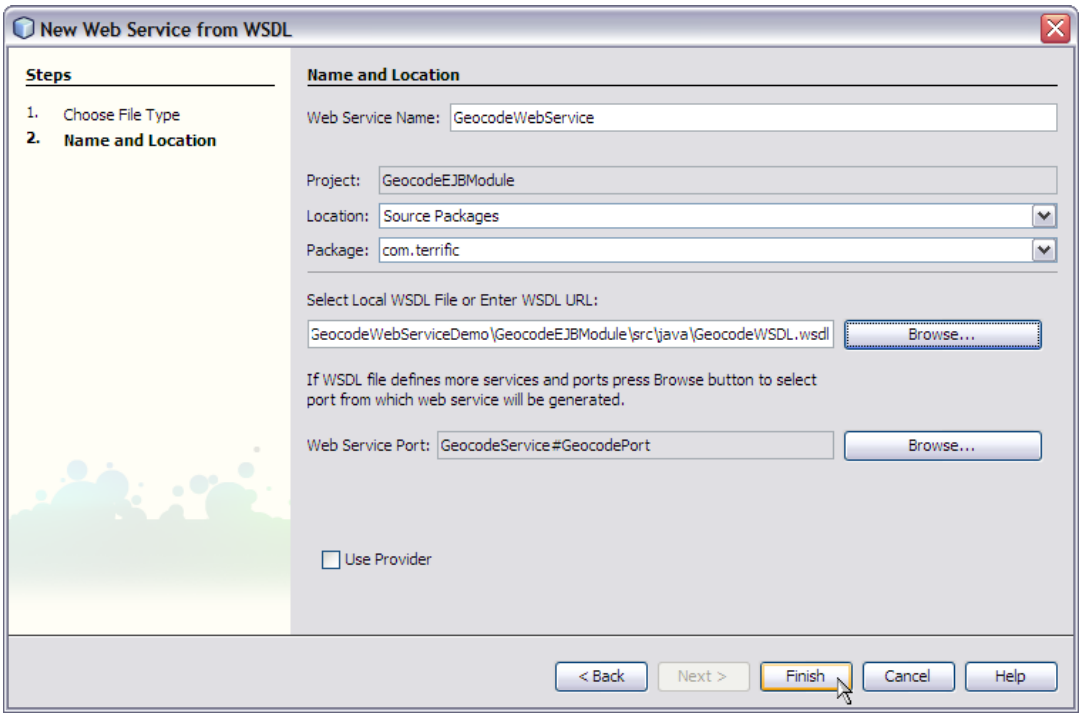
Note that we are using the data types defined in the XSD as types for the inputPart and outputPart.

Continue with the wizard sequence:

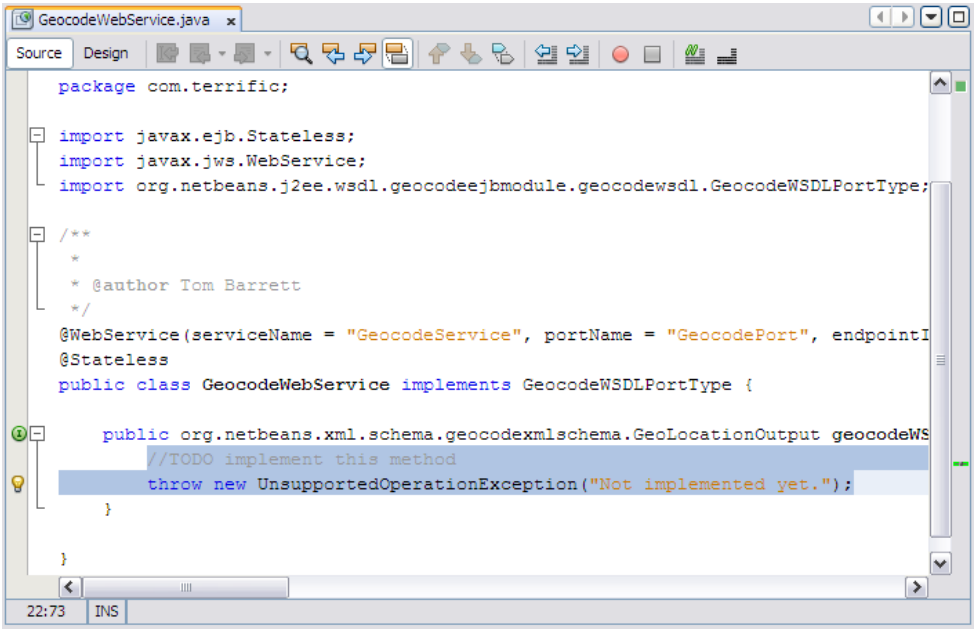


Create a New Web Service in the GeocodeEJBModule by selecting New / Other / Web Service / Web Service from WSDL. Name it: GeocodeWebService

- Package: com.terrific
- Browse to C:\GeocodeWebServiceDemo\GeocodeEJBModule\src\java\ and select GeocodeWSDL.wsdl



When the web service appears in the editor, enter Source mode and copy and paste the following code to replace the highlighted placeholder below:



Paste in this code:

```
// Thanks for guidance:  http://thepeninsulasedge.com/blog/?p=69
// Yahoo! Geocoder documentation at:
// http://developer.yahoo.com/maps/rest/V1/geocode.html
// RESTful endpoint that Yahoo! hosts is at:
String yahooRESTEndpoint = "http://api.local.yahoo.com/MapsService/V1/geocode";
// Here is my Yahoo!-provided appID
String appID = "YourAppIDHere";
String encodedQuery = null;
String query = inputPart.getQuery();
System.out.println("***** Received query: " + query);
try {
   // Encode location query so it can be delivered via a HTTP query string
   encodedQuery = URLEncoder.encode(query, "UTF-8");
   System.out.println("***** encodedQuery: " + encodedQuery);
} catch (UnsupportedEncodingException ex) {
   System.out.println("***** Sadness: " + ex.getMessage());
}
// Compose the HTTP request to send
String urlString = yahooRESTEndpoint + "?appid=" + appID + "&location=" + encodedQuery;
System.out.println("***** urlString: " + urlString);
GeoLocation geoLocation = null;
try {
   InputStream stream = null;
   // Get connection and send HTTP request
   URL url = new URL(urlString);
   URLConnection conn = url.openConnection();
   // Get the result from the RESTful endpoint
   stream = conn.getInputStream();
   // Use Apache Commons Digester to parse the resulting XML
   Digester digester = new Digester();
   digester.setValidating(false);
   // Parse the result based upon the XSD Yahoo! publishes at:
   // http://local.yahooapis.com/MapsService/V1/GeocodeResponse.xsd
   digester.addObjectCreate("ResultSet/Result", GeoLocation.class);
   digester.addBeanPropertySetter("ResultSet/Result/Latitude", "latitude");
   digester.addBeanPropertySetter("ResultSet/Result/Longitude", "longitude");
   digester.addBeanPropertySetter("ResultSet/Result/Address", "address");
   digester.addBeanPropertySetter("ResultSet/Result/City", "city");
   digester.addBeanPropertySetter("ResultSet/Result/State", "state");
   digester.addBeanPropertySetter("ResultSet/Result/Zip", "zip");
   digester.addBeanPropertySetter("ResultSet/Result/Country", "country");
   // Instantiate and populate GeoLocation using Digester
   geoLocation = new GeoLocation();
   geoLocation = (GeoLocation) digester.parse(stream);
   stream.close();
} catch (Exception ex) {
   System.out.println("***** Sadness: " + ex.getMessage());
}
// Populate the GeoLocationOutput data structure per our WSDL and XSD
org.netbeans.xml.schema.geocodexmlschema.GeoLocationOutput outputPart = new
  org.netbeans.xml.schema.geocodexmlschema.GeoLocationOutput();
outputPart.setLatitude(new BigDecimal(geoLocation.getLatitude()));
outputPart.setLongitude(new BigDecimal(geoLocation.getLongitude()));
outputPart.setAddress(geoLocation.getAddress());
outputPart.setCity(geoLocation.getCity());
```

```
        outputPart.setState(geoLocation.getState());
        outputPart.setZip(geoLocation.getZip());
        outputPart.setCountry(geoLocation.getCountry());
        // Return the element our WSDL and XSD say we are to return
        return outputPart;
```

Include the following statements right below the package statement at the top of the code:

```
import java.net.*;
import java.io.*;
import java.math.*;
import org.apache.commons.digester.Digester;
```

We still have errors involving the GeoLocation class. It can't be found. It's involved in a technique we'll use to parse the XML returned from the call to the Yahoo! Geocoder to get the fields values (like longitude and latitude).

Create a new Java class by right-clicking on GeocodeEJBModule and selecting New Java Class.

- For Class Name, use: GeoLocation
- Package: com.terrific

Replace the entire skeleton code with this:

```
package com.terrific;

public class GeoLocation {

    private String longitude;
    private String latitude;
    private String address;
    private String city;
    private String state;
    private String zip;
    private String country;

    public void GeoLocation() {
    }

    public void GeoLocation(String latitude, String longitude, String address, String city, String state, String zip, String country) {
        this.latitude = latitude;
        this.longitude = longitude;
        this.address = address;
        this.city = city;
        this.state = state;
        this.zip = zip;
        this.country = country;
    }

    public void setLatitude(String latitude) {
        this.latitude = latitude;
    }
```

```
    public String getLatitude() {
        return latitude;
    }
    public void setLongitude(String longitude) {
        this.longitude = longitude;
    }
    public String getLongitude() {
        return longitude;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getAddress() {
        return address;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCity() {
        return city;
    }
    public void setState(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
    public void setZip(String zip) {
        this.zip = zip;
    }
    public String getZip() {
        return zip;
    }
    public void setCountry(String country) {
         this.country = country;
    }
    public String getCountry() {
        return country;
    }
    public String toString() {
        return (latitude + "," + longitude + "," +
                address + "," + state + "," + zip + "," + country);
    }
}
```

Note: You could use the JAXB tooling support in NetBeans here as an alternative to the Apache Commons Digester component. JAXB provides very powerful marshaling and unmarshaling features between XML documents and Java objects. Using JAXB would certainly reduce the amount of code that you would have to write yourself.

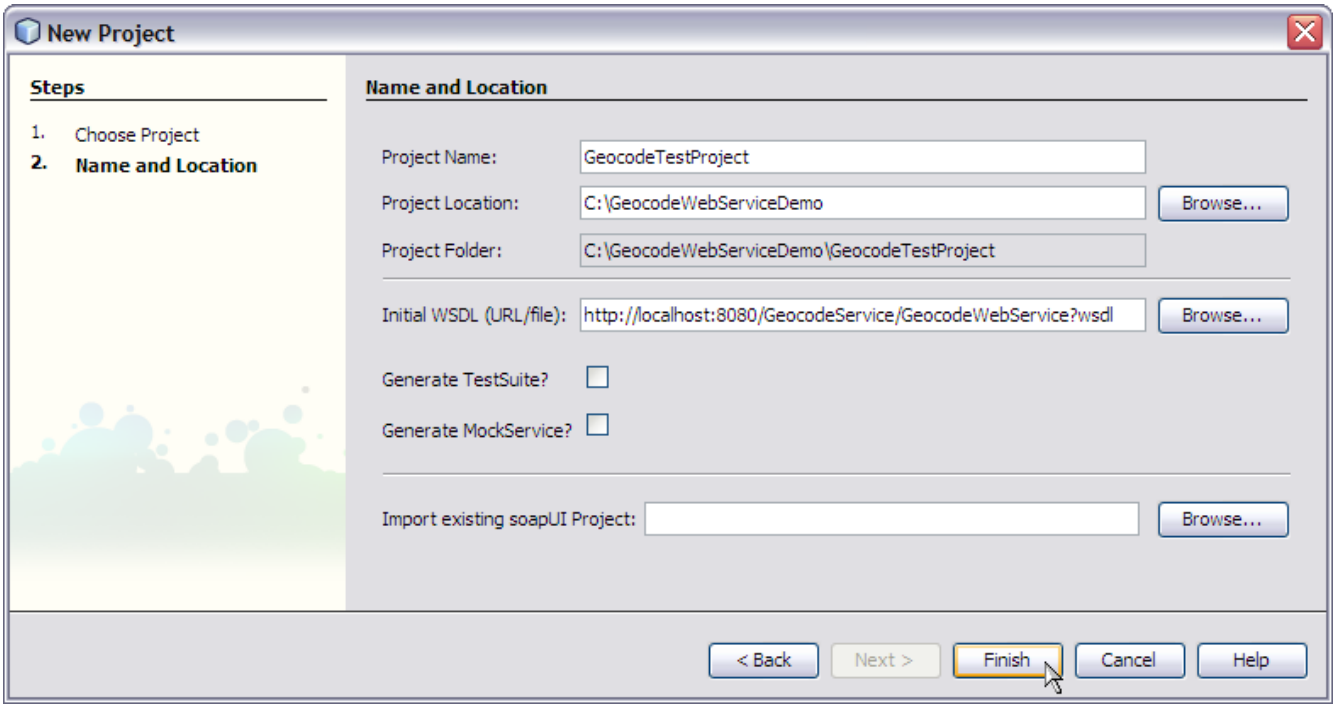
Save all

Clean and Build GeocodeEJBModule (Just Build if you get an error message that a directory can't be deleted during Clean and Build)

Deploy GeocodeEJBModule

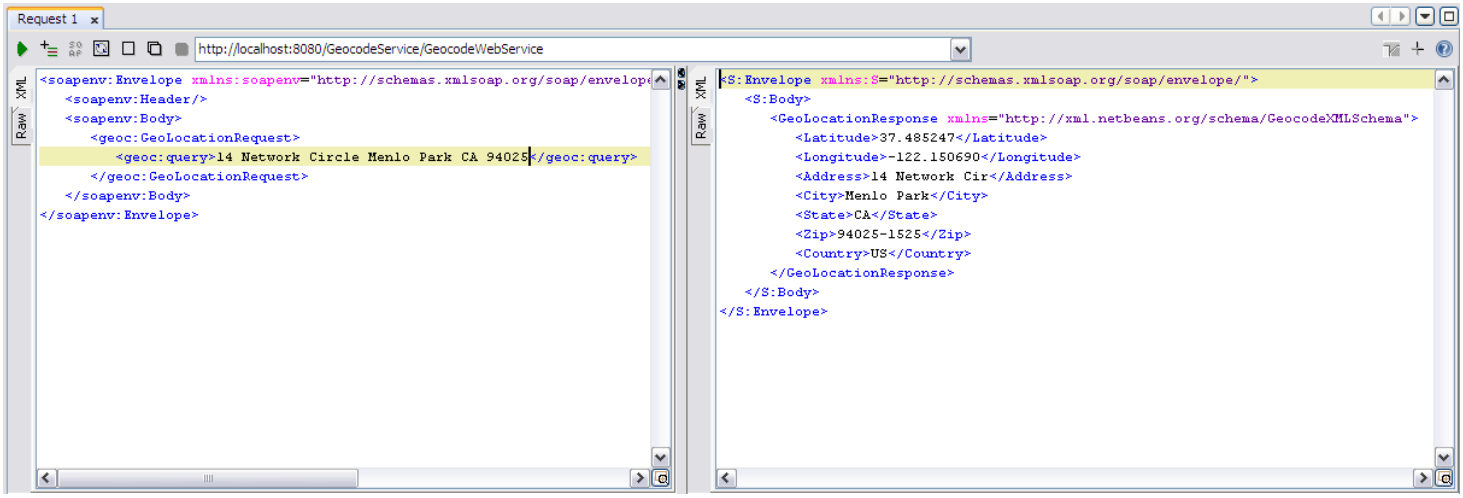
Create a new testing project: File / New Project / Enterprise / Web Service Testing Project (make sure you installed the soapUI plugin as noted above or you won't see this option)

- Project Name: GeocodeTestProject
- Project Location: C:\GeocodeWebServiceDemo
- Initial WSDL (URL/file): http://localhost:8080/GeocodeService/GeocodeWebService?wsdl



Open up the Request1 that was generated. For query, paste:
14 Network Circle Menlo Park CA 94025

View the successful response in the right-side panel:



You will see:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <GeoLocationResponse xmlns="http://xml.netbeans.org/schema/GeocodeXMLSchema">
      <Latitude>37.485247</Latitude>
      <Longitude>-122.150690</Longitude>
      <Address>14 Network Cir</Address>
      <City>Menlo Park</City>
      <State>CA</State>
      <Zip>94025-1525</Zip>
      <Country>US</Country>
    </GeoLocationResponse>
  </S:Body>
</S:Envelope>
```

Comparing and Contrasting the Two Approaches

Favoring Approach 1 - Using JBI: BPEL SE and HTTP BC

- BPEL provides a higher level of abstraction that might make maintenance easier especially if the maintainer is not the original creator.
- There is less work here that feels like programming. Are we approaching "near zero coding?" However, the Source view of the BPEL syntax is always available if you need it. You can make your changes in either the Source view or the Design (graphical) view.
- Tooling for BPEL provides a graphical display that makes it easier to see processing steps. The BPEL editor is advertised as business process modeling (BPM) though.
- BPEL can be a good prototyping approach to get something up and running quickly. Later, you could re-implement using a different technique.

Favoring Approach 2 - Using a EJB-Based Web Service

- The use case here is so simple that BPEL doesn't really value-add much on the clarity front.
- The BPEL engine is under-utilized. It has a powerful, long-lived transaction support that is not needed here. So, BPEL, for this scenario, may be like using a sledge hammer to pound a carpet tack.
- The BPEL engine adds some overhead so the EJB approach will likely be more performant and will likely handle more throughput.
- Use the right tool for the job. The use case simply asked for a SOAP -wrapped web service to allow a "SOAP shop" to easily use the RESTful Yahoo! geocoder service. Approach 2 delivers no more, no less -- Occam's Razor!
- Simpler is better. Approach 2 is very straightforward with fewer moving parts. If the user is not a programmer, this approach doesn't look simple though.

I would enjoy hearing your opinions: thomas.barrett@sun.com