Implementing OO based State-full HelloWorld

Version: 1.2 Date 10/03/10

This tutorial is the continuation of first tutorial "Converting a HelloWorld Web service into WSRF based HelloWorld" which can be found at http://tyne.dl.ac.uk/WOSE/Tutorial1.pdf. We will achieve nothing new in this tutorial except re-writing implementation according to the rules and learned lessons from OO techniques. Most of our deployment descriptors from previous tutorial are re-useable, but I have changed Web Service and target namespace from HelloWorldService1 to HelloWorldService2 and *http://tutorial1.wsrf.dl.ac.uk/helloworld to http://tutorial2.wsrf.dl.ac.uk/helloworld*. In fact based on each tutorial we will keep on modifying the deployment descriptors and configuration files referencing related tutorial number. These are although minor changes but can make difference later when we have to use stubs and skeleton generated by the Globus. If you remember from the last tutorial Globus uses target namespace to generate package statement for stubs and skeletons unless we provide namespace to package mapping in optional file "NStoPkg.properties". We will still not use ns2mapping.properties in this tutorial, but later we will investigate the utility of this mapping feature. If you are interested and keen right now to see the working of "NStoPkg.properties" then better check the tutorial by Borja Sotomayor in "The Globus Toolkit 4 Programmer's Tutorial". In this tutorial we are using target namespace http://tutorial2.wsrf.dl.ac.uk/helloworld therefore stubs and skeletons created by Globus will be in the package *"uk.ac.dl.wsrf.tutorial2.helloworld.*"*, *"uk.ac.dl.wsrf.tutorial2.helloworld.service.*"* and *"uk.ac.dl.wsrf.tutorial2.helloworld.bindings.*"*, it is very important to know default namespace to package conversion as later we have to use classes in our Web Service implementation and Client implementation even when those classes are not existing. More on this, later when, we will start implementation. Last tutorial was to convert HelloWorld Web Service into HelloWorld WSRF, this tutorial is about to convert HelloWorld WSRF into Object Oriented HelloWorld WSRF, the recommended way.

OO based State-full HelloWorld

We will start with simple WSRF *Service*, which, you have already seen in the last tutorial and discussing the changes which you need to made in this tutorial. I believe it is easy to understand and visualise when differences and similarities are represented in tabular way, rather than on different pages. I am Chemical Engineer, and due to my engineering background I always consider it is more time consuming to understand new concept in abstract way, as I am from school of thoughts "seeing is belief". Our simple WSRF *Service*, has two business methods, echo() and *getNameRP(), and one compulsory method GetResourceProperty(). If you see the implementation of simple WSRF Service, you will realise that it is fulfilling few different roles:*

1. *Responding to client's calls.*
2. *Declaring and creating Resource/s.*
3. *Initializing Resource/s.*

We need our Web Service to respond to client's call but we can other roles in separate class/es. This is what we are trying to achieve in this tutorial. In this tutorial we will have three classes:

1. HelloWorld.java i.e. Web Service to respond clients call.
2. HelloWorldResource.java i.e. Implementation of Resource/s to *declare and create Resource/s*
3. HelloWorldResourceHome.java i.e. Implementation to initialize Resource/s.

It is important to point out few things, we can combine last two classes i.e. HelloWorldResource.java and HelloWorldResourceHome.java, which is not a good approach. The approach which we are following is no one can access the Resource/s directly but it has to be done through HelloWorldResourceHome.java. There is nothing too complicated as long you can understand what we are trying to achieve, most of the code is very similar to previous example and borrowed as it is.

Simple WSRF (Tutorial 1)	HelloWorld.java	*Responding to client's calls*
		Declaring and creating Resource/s
		Initializing Resource/s

OO WSRF (Tutorial 2)	HelloWorld.java	*Web Service to respond clients call*
	HelloWorldResource.java	*Implementation of Resource/s for declaring & creating*
	HelloWorldResourceHome.java	*Implementation to initialize Resource/s*

Modifying WSDL File:

Just like previous tutorial we are starting with our WSDL file, which is similar to out last WSDL example with only change is different target name space, there are in total 4 changes highlighted by blue and bold font. There is no need to explain this copied WSDL file from our previous tutorial. This WSDL file is in the directory "schema\tutorial" relative to tutorial "src" directory with the name "helloworld2_port_type.wsdl", in last tutorial our WSDL file was helloworld1_port_type.wsdl.

```
<definitions name="HelloWorld"
  targetNamespace="http://tutorial2.wsrf.dl.ac.uk/helloworld"
  xmlns:tns="http://tutorial2.wsrf.dl.ac.uk/helloworld"
  xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
  xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
  xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
  xmlns:gtwsdl1="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-01.wsdl"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
  xmlns:wsntw="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
  xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
 <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-01.wsdl"
  location="../wsrf/faults/WS-BaseFaults.wsdl"/>
 <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
  location="../wsrf/lifetime/WS-ResourceLifetime.wsdl"/>
 <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
  location="../wsrf/properties/WS-ResourceProperties.wsdl"/>
 <import namespace="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
  location="../wsrf/notification/WS-BaseN.wsdl"/>
 <import namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-01.wsdl"
  location="../wsrf/servicegroup/WS-ServiceGroup.wsdl"/>


 <types>
  <schema
    targetNamespace="http://tutorial2.wsrf.dl.ac.uk/helloworld"
    xmlns:tns="http://tutorial2.wsrf.dl.ac.uk/helloworld"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <import namespace="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    schemaLocation="../ws/addressing/WS-Addressing.xsd"/>
  <import  namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-01.xsd"
    schemaLocation="../wsrf/servicegroup/WS-ServiceGroup.xsd"/>

   <element name="name" type="xsd:string"/>

   <element name="getNameRPRequest" >
    <complexType/>
   </element>
   <element name="getNameRPResponse" type="xsd:string"/>
```

Implementing OO based State-full HelloWorld

```
    <element name="HelloWorldResourcePropertiesSet">
     <complexType>
       <sequence>
         <element ref="tns:name"/>
       </sequence>
     </complexType>
    </element>

    <element name="echoRequest" type="xsd:string" />
    <element name="echoResponse" type="xsd:string" />

  </schema>
</types>

<message name="EchoRequest">
  <part name="EchoRequest"  element="tns:echoRequest" />
</message>
<message name="EchoResponse">
 <part name="EchoResponse" element="tns:echoResponse" />
</message>

<message name="GetNameRPRequest">
  <part name="GetNameRPRequest" element="tns: getNameRPRequest " />
</message>
<message name="GetNameRPResponse">
  <part name="GetNameRPResponse" element="tns:getNameRPResponse" />
</message>

<portType name="HelloWorldPortType"
        wsrp:ResourceProperties="HelloWorldResourcePropertiesSet">

  <operation name="GetResourceProperty">
    <input name="GetResourcePropertyRequest"
       message="wsrpw:GetResourcePropertyRequest"
       wsa:Action="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourceProperty"/>
   <output name="GetResourcePropertyResponse"
        message="wsrpw:GetResourcePropertyResponse"
        wsa:Action="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourcePropertyResponse"/>
   <fault  name="InvalidResourcePropertyQNameFault"
         message="wsrpw:InvalidResourcePropertyQNameFault"/>
    <fault name="ResourceUnknownFault" message="wsrpw:ResourceUnknownFault"/>
  </operation>

  <!--  name in input and output is optional-->
  <operation name="echo">
    <input  name="EchoRequest"  message="tns:EchoRequest" />
    <output name="EchoResponse" message="tns:EchoResponse" />
  </operation>

  <operation name="getNameRP">
    <input  name="GetNameRPRequest"  message="tns:GetNameRPRequest" />
    <output name="GetNameRPResponse" message="tns:GetNameRPResponse" />
  </operation>
```

```
  </portType>
</definitions>
```

Modified Implementation of Web Service:

Before we start implementing WSRF Service, I am writing simple utility class. This simple class is interface with few static and final variables. Interfaces can't be initialized and if interfaces have variables then they should be static and final, reason for static variables is very simple, as you can't create the object of interface therefore there is no way to access non-static variables and as there can be many objects using this interface and if anyone of them will change the value of static variable, change will be effective for all objects using this interface which may not be desired, thus keeping the variables final means that there is no possibility of even accidental change in value. In last tutorial where ever we had to use QName we have to create the instance of QName with target namespace URL and name of variable, which is cumbersome and error prone. This interface is just wrapping all QName used in our implementation. Below is the implementation of HelloQNames.java.

```
package uk.ac.dl.ws.service;

import javax.xml.namespace.QName;

public interface HelloQNames {

    public static final String NS = "http://tutorial2.wsrf.dl.ac.uk/helloworld";
    public static final QName RP_NAME = new QName(NS, "name");
    public static final QName RESOURCE_PROPERTIES = new QName(NS, "HelloWorldResourcePropertiesSet");

}
```

String NS is String value of our target namespace which is http://tutorial2.wsrf.dl.ac.uk/helloworld for tutorial 2. RP_NAME and RESOURCE_PROPERTIES are qualified name of our variable declared in our WSDL file. Only package we have to import is "*javax.xml.namespace.QName*" as this interface is utility interface and has nothing to do with WSRF implementation.

Now we start with HelloWorldResource.java which is implementation of Resource/s to declare and create resources, and in fact it is also initializing them, but no one will access it directly and all calls to initialize the tutorials should be through HelloWorldResourceHome.java. Below is the implementation of HelloWorldResource.java.

```
package uk.ac.dl.ws.service;

import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;

public class HelloWorldResource  implements Resource, ResourceProperties {

  /* Resource Property set */
  private ResourcePropertySet propSet;

  /* Resource properties */
  private String name;
```

```
 public void initialize() throws Exception {
  /* Create RP set */
  this.propSet = new SimpleResourcePropertySet(
     HelloQNames.RESOURCE_PROPERTIES);

  /* Initialize the RP's */
  try {
   ResourceProperty nameRP = new ReflectionResourceProperty(
      HelloQNames.RP_NAME, "name", this);
   this.propSet.add(nameRP);
   setName("Asif Akram Tutorial 2");

  }
  catch (Exception e) {
   throw new RuntimeException(e.getMessage());
  }
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 /* Required by interface ResourceProperties */
 public ResourcePropertySet getResourcePropertySet() {
  return this.propSet;
 }

}
```

This class is taking most of the code from previous tutorial. So we start with similarities between HelloWorld.java from previous tutorial (tutorial 1) and HelloWorldResource.java.

1. Both classes import most of Globus WSRF implementation related classes, which are related to the management of Resource/s.
   - import org.globus.wsrf.Resource;
   - import org.globus.wsrf.ResourceProperties;
   - import org.globus.wsrf.ResourceProperty;
   - import org.globus.wsrf.ResourcePropertySet;
   - import org.globus.wsrf.impl.ReflectionResourceProperty;
   - import org.globus.wsrf.impl.SimpleResourcePropertySet;
2. Both classes are implementing Resource and ResourceProperties interfaces, as name tells both interfaces are related to Resource/s.
3. Both classes are declaring variable propSet of type ResourcePropertySet, which is used to wrap all resources related to our WSRF Web Service into one set.

   ```
   /* Resource Property set */
   private ResourcePropertySet propSet;
   ```
4. Both classes are declaring String variable “name” which is actually our resource.

Implementing OO based State-full HelloWorld

```
/* Resource properties */
private String name;
```

5. Both classes have variable nameRP of type ResourceProperty, the only difference is HelloWorld.java is declaring variable in its constructor, while HelloWorldResource.java is declaring in the method intialize() which technically doesn't makes any difference. Below is the code fragement:

```
public void initialize() throws Exception {
    /* Create RP set */
    this.propSet = new SimpleResourcePropertySet(
            HelloQNames.RESOURCE_PROPERTIES);

    /* Initialize the RP's */
    try {
        ResourceProperty nameRP = new ReflectionResourceProperty(
            HelloQNames.RP_NAME, "name", this);
        this.propSet.add(nameRP);
        setName("Asif Akram Tutorial 2");

    }
    catch (Exception e) {
        throw new RuntimeException(e.getMessage());
    }
}
```

We have one variable called "name" which will store the value passed to the echo method of our Web Service. This variable is Resource Property of our Web Service, this concept is very clearly and in detail explained by Borja Sotomayor in "The Globus Toolkit 4 Programmer's Tutorial" under the section Getting Started. Remember your WSRF Web Service can have many Resource Properties and all of them are placed in *ResourcePropertySet* for later retrieval, modification and deletion. It is always nice to follow good naming conventions which are self explanatory, data type which will encapsulate all Resources of our WSRF Web Services will ends with "*ResourcePropertiesSet" to indicate that it is ResourcePropertySet,* all individual resources will ends with RP (Resource Property) to keep things clear and simple.
We have initialized the *"propSet"* as object of *SimpleResourcePropertySet,* you may have figured out that *ResourcePropertySet* is interface and *SimpleResourcePropertySet* is concrete implementation of *ResourcePropertySet*. Constructor of *SimpleResourcePropertySet* takes QName i.e. the qualified name of our resource property as declared in the WSDL file. Similarly *ResourceProperty* is an interface and *ReflectionResourceProperty* is one of its implementation used to declare and create variable *nameRP* (name Resource Property). ReflectionResourceProperty has three different constructors and the constructor used in the above example is: *ReflectionResourceProperty(QName name, String propertyName, Object obj)*, QName should match the qualified name in the WSDL and "String propertyName" is the name of variable in our implementation. It can be called anything like myName.
Then we have to add this newly created *ResourceProperty* into our *ResourcePropertySet*, *this.propSet.add(nameRP)* and then assign initial value by calling utility set method for private variable name.

Note, we are using our interface "HelloQNames.java" wherever we have to pass Qualified Name (Qname)

```
this.propSet = new SimpleResourcePropertySet(HelloQNames.RESOURCE_PROPERTIES);
nameRP = new ReflectionResourceProperty(HelloQNames.RP_NAME, "name", this);
```

6. Both classes are implementing getResourcePropertySet() method which is requirement as it is declared in the interface *ResourceProperties* implemented by our Web Service.

```
public ResourcePropertySet getResourcePropertySet() {
        return this.propSet;
}
```

7. Both classes are implementing get/set methods for our private variable "String name".

```
public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}
```

Second class we have to implement is "HelloWorldResourceHome.java", there is nothing common between HelloWorld.java from previous tutorial (tutorial 1) and "HelloWorldResourceHome.java". If you remember "HelloWorldResourceHome.java" is only to initialize our "HelloWorldResource.java". Below is the code of complete "HelloWorldResourceHome.java":

```
package uk.ac.dl.ws.service;

import org.globus.wsrf.Resource;
import org.globus.wsrf.impl.SingletonResourceHome;

public class HelloWorldResourceHome extends SingletonResourceHome{

 public Resource findSingleton (){
  try {
   HelloWorldResource helloWorldResource = new HelloWorldResource();
   helloWorldResource.initialize();
   return helloWorldResource;
  } catch (Exception e){
   e.printStackTrace();
   return null;
  }
 }
}
```

"HelloWorldResourceHome.java" is extending SingletonResourceHome (Globus Provided Class) very simple, but why? We haven't used SingletonResourceHome in our first tutorial, then why we suddenly need this Globus dependent class. It is not confusing, in fact in first tutorial also we were using similar class, we were not using SingletonResourceHome class directly but we were using similar class with common functionality. If you remember "deploy-jndi-config.xml" deployment descriptor, which was the last requirement of our service we were using

```
<resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
```

ServiceResourceHome extends *SingletonResourceHome*, so we using directly/indirectly *ServiceResourceHome*. We will cover our modified later "deploy-jndi-config.xml", but you may have guessed that now we will use our own implementation in "deploy-jndi-config.xml"; which, will be like this:

```
<resource name="home" type="uk.ac.dl.ws.service.HelloWorldResourceHome">
```

Implementing OO based State-full HelloWorld

More on this when we will write “deploy-jndi-config.xml”, in fact nothing more to say as this is the only change in the “deploy-jndi-config.xml” for second tutorial.
Important thing to remember is that this class has only one method *findSingleton()* which overrides the method declared in super class and provides custom implementation. This method will be called by container, when ever we will create instance of *HelloWorldResource*. As the name shows we can have only one instance of the resource, therefore multiple clients will share the same resource.

Third class to implement is our *HelloWorld.java*, which is the class with business logic of our Web Service, this class has remaining contents of *HelloWorld.java* (from tutorial 2). Below is complete code from *HelloWorld.java* followed by the similarities between latest *HelloWorld.java* and *HelloWorld.java* from tutorial 1.

```
package uk.ac.dl.ws.service;

import java.rmi.RemoteException;
import org.globus.wsrf.ResourceContext;
import uk.ac.dl.wsrf.tutorial2.helloworld.*;

public class HelloWorld {

 public HelloWorld() throws RemoteException {
 }

 public String echo (String name) throws RemoteException {
  HelloWorldResource helloWorldResource = getResource();
  helloWorldResource.setName(name);
  return "Hello " + name + " !";
 }

 public String getNameRP(GetNameRPRequest params) throws RemoteException {
  HelloWorldResource helloWorldResource = getResource();
  return helloWorldResource.getName();
 }

 public HelloWorldResource getResource() throws RemoteException {
  Object resource = null;
  try {
   resource = ResourceContext.getResourceContext().getResource();
  } catch (Exception e) {
   throw new RemoteException("", e);
  }
  HelloWorldResource helloWorldResource = (HelloWorldResource) resource;
  return helloWorldResource;
 }

}
```

Once again I have outlined the similarities between HelloWorld.java from tutorial 2 and HelloWorld.java from tutorial 1. After discussing similarities I will discuss the differences between both classes.

1. Both classes have business methods echo() and getNameRP(GetNameRPRequest params), although the implementation between is different. Difference is due to fact that in tutorial 1, business methods and Resource/s were in the same class, so business methods can access the resources directly. In tutorial 2, business methods and Resource/s are in different classes so business methods have to access the resources

from different class. Below is the code, with Blue font showing additional code added in latest HelloWorld.java and Green font shows the modified statements which are in both classes but with different syntax.

```
public String echo (String name) throws RemoteException {
        HelloWorldResource helloWorldResource = getResource();
        helloWorldResource.setName(name);
        return "Hello " + name + " !";
}

public String getNameRP(GetNameRPRequest params) throws RemoteException {
        HelloWorldResource helloWorldResource = getResource();
        return helloWorldResource.getName();
}
```

2. Both classes are using uk.ac.dl.wsrf.tutorial2.helloworld.* package, where newly created stubs and skeletons will be placed.

There

> Be sure about the last import statement, these are classes in the new package which matches the target namespace mentioned in our WSDL file.
>
> *targetNamespace="http://tutorial2.wsrf.dl.ac.uk/helloworld"*
>
> We can change this mapping from this "targetNamespace" to any required package by providing a simple text file called NStoPkg.properties, which, I have avoided in this simple tutorial.
>
> Package uk.ac.dl.wsrf.tutorial2.helloworld.* has few classes which are used by client and service both, If you remember when we were changing the WSDL file, I talked about wrapper to represent void and empty parameter to method call. Our both methods *echo()* and *getNameRP()* return *String* but getNameRP() doesn't take any parameter and this is wrapped in the empty dataType "*getNameRPRequest*". Stub will be created representing this data type and the name of stub generated is GetNameRPRequest.java. It is important to understand as you have to import the related classes which are still not available and you should know the location and name of these classes by analysing the WSDL file at the time when Web Service is implemented.

are few differences between HelloWorld.java from tutorial 2 and HelloWorld.java from tutorial 1. Below is the list of differences between both classes. HelloWorld1 means HelloWorld.java from tutorial 1 and HelloWorld2 means HelloWorld.java from tutorial 2.

1. HelloWorld1 is importing Globus WSRF related package/classes, most of them are used in the new class HelloWorldResource.java, and so they are no more needed in HelloWorld2.
2. HelloWorld2 is importing a new class *org.globus.wsrf.ResourceContext*, which is responsible to get hold of *HelloWorldResourceHome.java*.
3. HelloWorld2 has utility method getResource(), which is missing in the HelloWorld1. This methods is borrowed from the tutorial by Borja Sotomayor. This method is responsible for obtaining the object of Resource/s and returns the object of type HelloWorldResource.java. Remember Resource/s will be created through HelloWorldResourceHome.java, Below is the implementation of the method.

```
public HelloWorldResource getResource() throws RemoteException {
        Object resource = null;
        try {
                resource = ResourceContext.getResourceContext().getResource();
        } catch (Exception e) {
                throw new RemoteException("", e);
        }
```

Implementing OO based State-full HelloWorld

```
        HelloWorldResource helloWorldResource = (HelloWorldResource) resource;
        return helloWorldResource;
    }
```

This method is using the class ResourceContext to get the Resource from the container, and container will parse "deploy-jndi-config.xml" to know which Class is responsible for Resource initialization and will call its *findSingleton ()*. In our case it is HelloWorldResourceHome. Container returns generic *Resource* object which has to be casted into the required Resource Type.

4. Each business method echo() and getNameRP(..) have to call utility method getResource() to get resource and then calling the appropriate methods on the resource.

Below is the table which describes all differences and similarities of Tutorial 1 and Tutorial 2 implementation.

Tutorial 1	Tutorial 2

package uk.ac.dl.ws.service;
 	
import java.rmi.RemoteException;
import javax.xml.namespace.QName;
import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import uk.ac.dl.wsrf.tutorial1.helloworld.;*
 	
public class HelloWorld implements Resource, ResourceProperties {
 	
/ Resource Property set */*
private ResourcePropertySet propSet;
 	
/ Resource properties */*
private String name;
 	
public HelloWorld() throws RemoteException {
/ Create RP set */*
this.propSet = new SimpleResourcePropertySet(
new QName("http://tutorial.wsrf.dl.ac.uk/helloworld", "HelloWorldResourceProperties"));
 	
/ Initialize the RP's */*
try {
ResourceProperty nameRP = new ReflectionResourceProperty(
new QName("http://tutorial.wsrf.dl.ac.uk/helloworld", "name"), "name", this);
this.propSet.add(nameRP);
setName("Asif"); // this is set method for private variable name
}
catch (Exception e) {
throw new RuntimeException(e.getMessage());
}
}
 	
public String getName() {
return name;
}
	package uk.ac.dl.ws.service;
	
	import org.globus.wsrf.Resource;
	import org.globus.wsrf.ResourceProperties;
	import org.globus.wsrf.ResourceProperty;
	import org.globus.wsrf.ResourcePropertySet;
	import org.globus.wsrf.impl.ReflectionResourceProperty;
	import org.globus.wsrf.impl.SimpleResourcePropertySet;
	
	public class HelloWorldResource implements Resource, ResourceProperties {
	
	/ Resource Property set */*
	private ResourcePropertySet propSet;
	
	/ Resource properties */*
	private String name;
	
	public void initialize() throws Exception {
	/ Create RP set */*
	this.propSet = new SimpleResourcePropertySet(
	HelloQNames.RESOURCE_PROPERTIES);
	
	/ Initialize the RP's */*
	try {
	ResourceProperty nameRP = new ReflectionResourceProperty(
	HelloQNames.RP_NAME, "name", this);
	this.propSet.add(nameRP);
	setName("Asif Akram Tutorial 2");
	
	}
	catch (Exception e) {
	throw new RuntimeException(e.getMessage());
	}
	}
	
	public String getName() {
	return name;
	}
	
	public void setName(String name) {
	this.name = name;
	}
	
	/ Required by interface ResourceProperties */*

Implementing OO based State-full HelloWorld

```
    public void setName(String name) {
        this.name = name;
    }

    public String echo(String name) {
        setName(name);
        return "Hello " + name + " !";
    }

    public String getNameRP(GetNameRP params) throws
          RemoteException {
        return name;
    }

    /* Required by interface ResourceProperties */
    public ResourcePropertySet getResourcePropertySet() {
        return this.propSet;
    }
}
```

NOTE:
Green Code shows similar code in HelloWorld.java from Tutorial 1 and HelloWorld.java from Tutorial 2.

Light Green Code shows code in HelloWorld.java from Tutorial 1 and HelloWorld.java from Tutorial 2 but with slightly different syntax, although performing similar function.

Blue Code shows similar code in HelloWorld.java from Tutorial 1 and HelloWorldResource.java from Tutorial 2.

Red Code in Tutorial 2 shows new code added.

```
    public ResourcePropertySet getResourcePropertySet() {
      return this.propSet;
    }

}
```

```
package uk.ac.dl.ws.service;

import org.globus.wsrf.Resource;
import org.globus.wsrf.impl.SingletonResourceHome;

public class HelloWorldResourceHome extends
SingletonResourceHome{

  public Resource findSingleton (){
    try {
      HelloWorldResource helloWorldResource = new
          HelloWorldResource();
      helloWorldResource.initialize();
      return helloWorldResource;
    } catch (Exception e){
      e.printStackTrace();
      return null;
    }
  }
}
```

```
package uk.ac.dl.ws.service;

import java.rmi.RemoteException;
import org.globus.wsrf.ResourceContext;
import uk.ac.dl.wsrf.tutorial2.helloworld.*;

public class HelloWorld {

  public HelloWorld() throws RemoteException {
  }

  public String echo (String name) throws RemoteException {
    HelloWorldResource helloWorldResource = getResource();
    helloWorldResource.setName(name);
    return "Hello " + name + " !";
  }

  public String getNameRP(GetNameRPRequest params) throws
      RemoteException {
    HelloWorldResource helloWorldResource = getResource();
    return helloWorldResource.getName();
  }

  public HelloWorldResource getResource() throws
RemoteException {
    Object resource = null;
    try {
      resource =
ResourceContext.getResourceContext().getResource();
    } catch (Exception e) {
      throw new RemoteException("", e);
    }
    HelloWorldResource helloWorldResource =
(HelloWorldResource) resource;
    return helloWorldResource;
```

	} }

Modified Deployment Descriptor:

Have we finished modification to our WSRF Web Service? Not yet … Don't forget we need Axis Engine dependent Deployment Descriptor to deploy our Web Service. This WSDD is very similar to previous WSDD only change is the change in the service name.

```
<deployment name="defaultServerConfig" xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:aggr="http://mds.globus.org/aggregator/types"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <service name="HelloWorldService2" provider="Handler" use="literal" style="document">
    <parameter name="providers" value="GetRPProvider"/>
    <parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
    <parameter name="scope" value="Application"/>
    <parameter name="allowedMethods" value="*"/>
    <parameter name="activateOnStartup" value="true"/>
    <parameter name="className" value=" uk.ac.dl.ws.service.HelloWorld "/>
    <wsdlFile>share/schema/tutorial/HelloWorld2_service.wsdl</wsdlFile>
  </service>
</deployment>
```

Other than just changing the name of our Service we have to mention the location of Final WSDL file which will be created by Globus when we will run Ant script. The final WSDL file will have similar name as our incomplete WSDL with suffix "_service.wsdl". In our WSDL file is helloworld2_port_type.wsdl therefore final WSDL file is HelloWorld2_service.wsdl. This is very important change; otherwise it will refer to WSDL file from previous tutorial and will give errors. In fact this is the name of WSDL file we have mentioned in the Ant script. If you remember we have changed the default script in last tutorial and this time the change will be:

```
<property name="binding.root" value="HelloWorld2"/>
```

Modified Deployment "jndi-config":

The last thing we need is "deploy-jndi-config.xml". This file is related to Tomcat Server and tells the server about the services and the class which provides the detail of services. This file is very simple just like previous tutorial when we were using custom implementation provided by Globus to wrap our WSRF Web Service.

"HelloWorldResourceHome.java" is extending *SingletonResourceHome* (Globus Provided Class) very simple, but why? We haven't used *SingletonResourceHome* in our first tutorial, then why we suddenly need this Globus dependent class. It is not confusing, in fact in first tutorial also we were using similar class, and we were not using *SingletonResourceHome* class directly. If you remember "*deploy-jndi-config.xml*" deployment descriptor, which was the last requirement of our service we were using

```
<resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
```

ServiceResourceHome extends SingletonResourceHome, so we using directly/indirectly *SingletonResourceHome*. In our modified *"deploy-jndi-config.xml"*, you may have guessed that now we will use our own implementation in "*deploy-jndi-config.xml"*; which, will be like this:

```
<resource name="home" type="uk.ac.dl.ws.service.HelloWorldResourceHome">
```

This is the only change in the *"deploy-jndi-config.xml"* for second tutorial.

```
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

  <service name="HelloWorldService">
    <resource name="home" type=" uk.ac.dl.ws.service.HelloWorldResourceHome ">
      <resourceParams>
        <parameter>
          <name>factory</name>
          <value>org.globus.wsrf.jndi.BeanFactory</value>
        </parameter>
      </resourceParams>
    </resource>
  </service>
</jndiConfig>
```

Tutorial 1: *"deploy-jndi-config.xml"*	Tutorial 2: *"deploy-jndi-config.xml"*

`<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">`
 	
`<service name="HelloWorldService">`
`<resource name="home"`
`type=" org.globus.wsrf.impl.ServiceResourceHome">`
`<resourceParams>`
`<parameter>`
`<name>factory</name>`
`<value>org.globus.wsrf.jndi.BeanFactory</value>`
`</parameter>`
`</resourceParams>`
`</resource>`
`</service>`
`</jndiConfig>`
	`<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">`
	
	`<service name="HelloWorldService">`
	`<resource name="home"`
	`type=" uk.ac.dl.ws.service.HelloWorldResourceHome">`
	`<resourceParams>`
	`<parameter>`
	`<name>factory</name>`
	`<value>org.globus.wsrf.jndi.BeanFactory</value>`
	`</parameter>`
	`</resourceParams>`
	`</resource>`
	`</service>`
	`</jndiConfig>`

Modified Implementation of our Client:

Last thing to do is to write Client to test our WSRF Web Service. Below is the complete code of Client1.java followed by discussing important parts, notice that Client1.java is in package "uk.ac.dl.ws.service.*client*" and our service HelloWorld.java is in package "*package* uk.ac.dl.ws.service". This is not the requirement of WSRF or Globus implementation of WSRF, in fact it is restriction due to the Ant "build.xml" file which will be used later to generate stubs, compile service and deploy to the container. You can change "build.xml" to adjust changes made in your package structure. We have to put all java classes and additional files like WSDL and "deploy-jndi-config.xml" in specific directories as required by Ant "build.xml". Ant "build.xml" is so handy and useful that following few restrictions is still worthy, as it hides all dirty work of setting class path, copying files from different locations and above all making all imported files in our WSDL available to our WSDL file.

```
package uk.ac.dl.ws.service.client;

import org.oasis.wsrf.properties.WSResourcePropertiesServiceAddressingLocator;
import org.oasis.wsrf.properties.GetResourceProperty;
import org.oasis.wsrf.properties.GetResourcePropertyResponse;

import org.apache.commons.cli.ParseException;
import org.apache.commons.cli.CommandLine;

import org.globus.wsrf.client.BaseClient;
```

```
import org.globus.wsrf.utils.AnyHelper;
import org.globus.wsrf.utils.FaultHelper;

import javax.xml.namespace.QName;
import javax.xml.rpc.Stub;
import uk.ac.dl.ws.service.HelloQNames;
import uk.ac.dl.wsrf.tutorial2.helloworld.service.*;
import uk.ac.dl.wsrf.tutorial2.helloworld.*;

public class Client1
    extends BaseClient {

  final static WSResourcePropertiesServiceAddressingLocator locator =
      new WSResourcePropertiesServiceAddressingLocator();

  public static void main(String[] args) {
    Client1 client = new Client1();

    // first, parse the commandline
    try {
      CommandLine line = client.parse(args);
    }
    catch (ParseException e) {
      System.err.println("Parsing failed: " + e.getMessage());
      System.exit(1);
    }
    catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      System.exit(1);
    }

    try {

      GetResourceProperty port =  locator.getGetResourcePropertyPort(client.getEPR());
      client.setOptions( (Stub) port);

      GetResourcePropertyResponse response =  port.getResourceProperty(HelloQNames.RP_NAME);
      System.out.println(AnyHelper.toSingleString(response));
      HelloWorld2ServiceAddressingLocator mylocator = new HelloWorld2ServiceAddressingLocator();

      HelloWorldPortType myPort = mylocator.getHelloWorldPortTypePort (client.getEPR());
      String result = myPort.echo("Rob Allan First Time");
      response = myPort.getResourceProperty(HelloQNames.RP_NAME);
      System.out.println(AnyHelper.toSingleString(response));
      result = myPort.echo("Rob Allan Second Time");
      response = myPort.getResourceProperty(HelloQNames.RP_NAME);
      System.out.println(AnyHelper.toSingleString(response));
    }
    catch (Exception e) {
      if (client.isDebugMode()) {
        FaultHelper.printStackTrace(e);
      }
      else {
        System.err.println("Error: " + FaultHelper.getMessage(e));
      }
    }
  }
}
```

Implementation of Client once again starts with importing couple of Globus dependent java classes along with importing few classes which are not yet available and will created by Ant build file. One of them is to locate our Web Service *"HelloWorldServiceAddressingLocator"* which is in the package

Implementing OO based State-full HelloWorld

"uk.ac.dl.wsrf.tutorial2.helloworld.service" and the *"HelloWorldPortType"* which is the interface with all three methods available in our main Web Service. *"HelloWorldPortType" is available in the package "import uk.ac.dl.wsrf.tutorial2.helloworld".* Our client implementation is using two different approaches to get hold of stub to communicate with Web Service i.e. first we are using Globus provided class *WSResourcePropertiesServiceAddressingLocator* through which we obtain generic stub *GetResourceProperty*, limitation of this approach is that we can only access one method which must be implemented by the Web Service that's is *getResourceProperty()* as shown in following code fragment;

```
WSResourcePropertiesServiceAddressingLocator locator= new
                        WSResourcePropertiesServiceAddressingLocator();
GetResourceProperty port = locator.getGetResourcePropertyPort(client.getEPR());
port.getResourceProperty(…….);
```

Second approach is to use HelloWorldServiceAddressingLocator to create custom stub HelloWorldPortType, through this stub we have access to all operations exposed by WSDL as shown in following code fragment;

```
HelloWorldServiceAddressingLocator mylocator =  new HelloWorldServiceAddressingLocator();
HelloWorldPortType myPort = mylocator.getHelloWorldPortTypePort (client.getEPR());
String result = myPort.echo("Rob Allan First Time ");
```

If you compare the implementation of Client1.java with the implementation of Client.java in tutorial 1, you will notice that we were importing too many packages, which were not used in the Client.java. We in fact don't need to import all those packages if we are just creating Singleton resource. Most of imported classes were related to querying the Resource, although in this simple tutorial we are not querying the resources, but later we will see those packages/classes in actions.

Modified "post-deploy":

Now everything is available and we have to build the WSRF Web Service. For build purposes we will use build tool "ANT" and the build.xml provided by the Globus. To make life easy to test web service we need last file "post-deploy.xml" which will create script to run the client, without setting class path, without worrying about the generated stubs and location of compiled stubs. This xml file is generated when everything goes well, below are the contents of file "post-deploy.xml".

```
<project default="all" basedir=".">
   <property environment="env"/>
   <property file="build.properties"/>
   <property file="${user.home}/build.properties"/>
   <property name="env.GLOBUS_LOCATION" value="."/>
   <property name="deploy.dir" location="${env.GLOBUS_LOCATION}"/>
   <property name="abs.deploy.dir" location="${deploy.dir}"/>
   <property name="build.launcher"
     location="${abs.deploy.dir}/share/globus_wsrf_common/build-launcher.xml"/>

   <target name="setup">
     <ant antfile="${build.launcher}"
        target="generateLauncher">
       <property name="launcher-name" value="helloworld2"/>
       <property name="class.name"  value="uk.ac.dl.ws.service.client.Client1"/>
     </ant>
  </target>
</project>
```

Implementing OO based State-full HelloWorld

This file is quite simple, and most of time remains same for most of clients. There are two adjustments to be made for each client and location of those adjustments is shown in bold. Name of the generated script: ***<property name="launcher-name" value="helloworld2"/>*** and location and name of java client of WSRF Web Service. In our case client is "*Client1.java*" in package "***uk.ac.dl.ws.service.client":***

```
<property name="class.name"    value="uk.ac.dl.ws.service.client.Client1"/>
```

Tutorial 1: *"post-deploy.xml"*	Tutorial 2: *"post-deploy.xml"*
<project default="all" basedir="."> <property environment="env"/> <property file="build.properties"/> <property file="${user.home}/build.properties"/> <property name="env.GLOBUS_LOCATION" value="."/> <property name="deploy.dir" location="${env.GLOBUS_LOCATION}"/> <property name="abs.deploy.dir" location="${deploy.dir}"/> <property name="build.launcher" location="${abs.deploy.dir}/share/globus_wsrf_common/build-launcher.xml"/> <target name="setup"> <ant antfile="${build.launcher}" target="generateLauncher"> <property name="launcher-name" value="helloworld"/> <property name="class.name" value="uk.ac.dl.ws.service.client.Client"/> </ant> </target> </project>	<project default="all" basedir="."> <property environment="env"/> <property file="build.properties"/> <property file="${user.home}/build.properties"/> <property name="env.GLOBUS_LOCATION" value="."/> <property name="deploy.dir" location="${env.GLOBUS_LOCATION}"/> <property name="abs.deploy.dir" location="${deploy.dir}"/> <property name="build.launcher" location="${abs.deploy.dir}/share/globus_wsrf_common/build-launcher.xml"/> <target name="setup"> <ant antfile="${build.launcher}" target="generateLauncher"> <property name="launcher-name" value="helloworld2"/> <property name="class.name" value="uk.ac.dl.ws.service.client.Client1"/> </ant> </target> </project>

Modified Ant Script:

Now we have to use Ant build file, but that Ant file also needs few changes related to our Web Service. These changes are very small and those changes are the last step required to finish the whole implementation.

Once again I have copied the complete build file and highlighted the changes related to Tutorial 2 in Blue Font:

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="all" name="globus_tutorial_stickynote">
    <description>
        WSRF MDS Aggregator Implementation
    </description>

    <!--
    Give user a chance to override without editing this file
    (and without typing -D each time it compiles it)
    -->

    <property environment="env"/>

    <property file="build.properties"/>
    <property file="${user.home}/build.properties"/>
    <!—it should be related to your Globus Server Installation -->
    <property location="../install" name="env.GLOBUS_LOCATION"/>
    <property location="${env.GLOBUS_LOCATION}" name="deploy.dir"/>
```

```
<property name="base.name" value="tutorial2_helloworld"/>
<property name="package.name" value="globus_${base.name}"/>
<property name="jar.name" value="${package.name}.jar"/>
<property name="gar.name" value="${package.name}.gar"/>
<property location="build" name="build.dir"/>
<property location="build/classes" name="build.dest"/>
<property location="build/lib" name="build.lib.dir"/>
<property location="build/stubs" name="stubs.dir"/>
<property location="build/stubs/src" name="stubs.src"/>
<property location="build/stubs/classes" name="stubs.dest"/>
<property name="stubs.jar.name" value="${package.name}_stubs.jar"/>
<property location="${deploy.dir}/share/globus_wsrf_common/build-packages.xml"
      name="build.packages"/>
<property location="${deploy.dir}/share/globus_wsrf_tools/build-stubs.xml" name="build.stubs"/>
<property name="java.debug" value="on"/>

<property name="test-reports.dir" value="test-reports"/>
<property name="junit.haltonfailure" value="true"/>

<property location="${deploy.dir}/share/schema" name="schema.src"/>
<property location="${build.dir}/schema" name="schema.dest"/>

<property name="garjars.id" value="garjars"/>
<fileset dir="${build.lib.dir}" id="garjars"/>

<property name="garschema.id" value="garschema"/>
<fileset dir="${schema.dest}" id="garschema" includes="tutorial/*"/>

<property name="garetc.id" value="garetc"/>
<fileset dir="etc" id="garetc"/>



<target name="init">
   <mkdir dir="${build.dir}"/>
   <mkdir dir="${build.dest}"/>
   <mkdir dir="${build.lib.dir}"/>

   <mkdir dir="${stubs.dir}"/>
   <mkdir dir="${stubs.src}"/>
   <mkdir dir="${stubs.dest}"/>

   <mkdir dir="${schema.dest}"/>

   <copy toDir="${schema.dest}">
      <fileset dir="${schema.src}" casesensitive="yes">
         <include name="wsrf/**/*"/>
         <include name="ws/**/*"/>
      </fileset>
   </copy>
   <copy toDir="${schema.dest}">
      <fileset dir="schema/" casesensitive="yes">
```

```
            <include name="tutorial/*"/>
        </fileset>
    </copy>

    <available file="${stubs.dest}/org.globus.wsrf.mds.aggregator" property="stubs.present" type="dir"/>
</target>

<target name="bindings" depends="init">
<ant antfile="${build.stubs}" target="generateBinding">
        <property name="source.binding.dir"
            value="${schema.dest}/tutorial"/>
        <property name="target.binding.dir"
            value="${schema.dest}/tutorial"/>
        <property name="binding.root" value="HelloWorld2"/>
        <property name="porttype.wsdl" value="helloworld2_port_type.wsdl"/>
    </ant>

</target>

<target name="stubs" unless="stubs.present" depends="bindings">
    <ant antfile="${build.stubs}" target="generateStubs">
        <property location="${schema.dest}/tutorial" name="source.stubs.dir"/>
        <property name="wsdl.file" value="HelloWorld2_service.wsdl"/>
        <property location="${stubs.src}" name="target.stubs.dir"/>
    </ant>
</target>


<target depends="stubs" name="compileStubs">
    <delete dir="${stubs.src}/org/apache"/>
    <javac debug="${java.debug}" destdir="${stubs.dest}" srcdir="${stubs.src}">
        <include name="**/*.java"/>
        <classpath>
            <fileset dir="${deploy.dir}/lib">
                <include name="*.jar"/>
            </fileset>
        </classpath>
    </javac>
</target>

<target depends="compileStubs" name="compile">
    <javac debug="${java.debug}" destdir="${build.dest}" srcdir="src">
        <include name="**/*.java"/>
        <classpath>
            <pathelement location="${stubs.dest}"/>
            <fileset dir="${deploy.dir}/lib">
                <include name="*.jar"/>
            </fileset>
        </classpath>
    </javac>
</target>
```

Implementing OO based State-full HelloWorld

```
<target depends="compileStubs" name="jarStubs">
  <jar basedir="${stubs.dest}" destfile="${build.lib.dir}/${stubs.jar.name}"/>
</target>


<target depends="compile" name="jar">
  <jar basedir="${build.dest}" destfile="${build.lib.dir}/${jar.name}"/>
</target>

<target depends="jar, jarStubs" name="dist">
  <ant antfile="${build.packages}" target="makeGar">
    <property name="gar.name" value="${build.lib.dir}/${gar.name}"/>
    <reference refid="${garjars.id}"/>
    <reference refid="${garschema.id}"/>
    <reference refid="${garetc.id}"/>
  </ant>
</target>

<target name="clean">
  <delete dir="tmp"/>
  <delete dir="${build.dir}"/>
  <delete file="${gar.name}"/>
  <delete dir="${test-reports.dir}"/>
</target>

<target depends="dist" name="deploy">
  <ant antfile="${build.packages}" target="deployGar">
    <property name="gar.name" value="${build.lib.dir}/${gar.name}"/>
    <property name="gar.id" value="${package.name}"/>
    <!-- <property name="noSchema" value="true"/> -->
  </ant>
</target>

<target name="undeploy">
  <ant antfile="${build.packages}" target="undeployGar">
    <property name="gar.id" value="${package.name}"/>
  </ant>
</target>

<target depends="deploy" name="all"/>

<target depends="compile" name="test">
  <mkdir dir="${test-reports.dir}"/>
  <junit fork="yes" haltonfailure="${junit.haltonfailure}" printsummary="yes" timeout="600000">
    <classpath>
      <pathelement location="${build.dest}"/>
      <pathelement location="${deploy.dir}"/>
      <pathelement location="${deploy.dir}/etc/wsrf-bundles.properties"/>
      <fileset dir="${deploy.dir}/lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
```

```
        <formatter type="xml"/>
        <batchtest todir="${test-reports.dir}">
          <fileset dir="${build.dest}">
            <include name="**/*Test.class"/>
          </fileset>
        </batchtest>
      </junit>
    </target>
</project>
```

There are only 4 minor changes in the Ant Script from Tutorial 1; those changes are highlighted in the Blue Font. Changes in the script are more related to changing the name of WSDL file created by us for Tutorial 2 i.e. "*helloworld2_port_type.wsdl*", and the name which Ant Script will use to generate complete WSDL file,

```
<property name="binding.root" value="HelloWorld2"/>
<property name="porttype.wsdl" value="helloworld2_port_type.wsdl"/>
```

Before we can run Ant script we have to put them at proper location where Ant build can locate them. HelloWorld140605B is the main directory containing all files under one umbrella, our Web Service implementation *HelloWorld.java* will be in the directory structure according to package statement i.e. "*HelloWorld140605B\source\src\uk\ac\dl\ws\service*" our *Client1.java* will be the directory "*HelloWorld140605B\source\src\uk\ac\dl\ws\service\client*" (2). Our WSDL file "*helloworld2_port_type.wsdl*" is located at "*HelloWorld140605B\source\schema\tutorial*" and source directory contains "deploy-server.wsdd", "deploy-jndi-config.xml" and "build.xml". File "post-deploy.xml", which; is used for creating client script is in directory "HelloWorld140605B*source\etc*".

Compiling and Deploying Modified WSRF Web Service:

I am using Windows Operating System and have installed only WS-Core component of Globus Toolkit. You have to set an environment variable called GLOBUS_LOCATION referencing the installation of Globus Toolkit. In my case it is:

```
set GLOBUS_LOCATION= E:\GlobusWeek\gt-install
```

Open two Command Prompt one for starting Globus Tomcat Server and other for deploying Web Service and testing client. In one window go to directory E:\GlobusWeek\HelloWorld140605B\source and run build file to create stubs, compile all java files, and deploy Web Service.

```
ant clean
ant deploy
```

If everything goes fine then after 10 seconds you will see the final outcome similar to:

```
generateWindows:
[echo] Creating Windows launcher script helloworld2
[copy] Copying 1 file to E:\GlobusWeek\gt-install\bin
[delete] Deleting: E:\GlobusWeek\HelloWorld140605B\source\tmp\gar\etc\post-deploy.xml
[delete] Deleting directory E:\GlobusWeek\HelloWorld140605B\source\tmp\gar
BUILD SUCCESSFUL
Total time: 8 seconds
```

In second Window go to directory "E:\GlobusWeek\gt-install" and run command,

```
bin\globus-start-container –nosec
```

which will start the container.

Testing Modified WSRF Web Service:

Implementing OO based State-full HelloWorld

If you remember in the file "post-deploy.xml" we have mentioned helloworld as name of script to be generated to run the client, which is confirmed by the final outcome of the Ant build file. Now it is time to run the client:

```
%GLOBUS_LOCATION%\bin\helloworld2 -s  http://localhost:8080/wsrf/services/HelloWorldService2
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Asif Akram Tutorial 2</ns1:name>
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Rob Allan First Time</ns1:name>
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Rob Allan Second Time</ns1:name>
```

"*Asif Akram Tutorial 2*" is the initial value assigned to our variable "name" and "*Rob Allan First Time*" is the value assigned to variable "name" from "Client1.java". Client1.java calls "echo" method twice and second time it passes the value "*Rob Allan Second Time*". If you will run the script again then the out come will be:

```
%GLOBUS_LOCATION%\bin\helloworld2 -s  http://localhost:8080/wsrf/services/HelloWorldService2
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Rob Allan Second Time</ns1:name>
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Rob Allan First Time</ns1:name>
<ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld">Rob Allan Second Time</ns1:name>
```

When we are running the client script for second time, our name variable has already value "*Rob Allan Second Time*" and we are assigning the "*Rob Allan First Time*" again, due to which it has printed *"Rob Allan Second Time"* twice.

"helloworld2" script is calling the service 5 times, and if you see the SOAP messages then there are 5 calls, thrice we are calling "getResourceProperty(..)" and twice we are calling "echo()" method. Call to "echo()" method is simple SOAP call, Tutorial 1 has the SOAP request and SOAP response from "getResourceProperty(..)" and "echo()" method, which are same for Tutorial 2. There is nothing too much complicated happening here. Most important thing to remember is that each resource in WSRP has unique ID to differentiate it from other resources, which along with the URL of Web Service is called End Point Reference but in the case of Singleton Resource there is only one Resource so it doesn't need any unique ID and all clients are sharing the same Resource. In this simple example, for all clients there is only one copy of the resource. In other words clients are sharing the resource but if multiple copies of the resource are possible then each copy will have unique ID which we will see in the later examples. When clients make a request, it sends its URL or location where Web Service can send the response. "http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous" is special instruction to Web Service indicating that there is no real endpoint available for this address and to keep the "http" connection open and use the same connection to send the response, which means we can have synchronously or asynchronously web services. Client sends the SOAP Request and SOAP Response can be sent to the URL provided, therefore client doesn't have to wait for the response, which is non-blocking call. Even the request can be sent to any other URL provided by the client. Requester and Receiver can be different. More on this later, when, we will see the WS-Addressing Specification in more detail. To understand the purpose and reason behind this new specification WS-Addressing there is a tutorial by Doug Davis on IBM website.

- http://www-106.ibm.com/developerworks/library/ws-address.html

Below is the table showing SOAP Request and SOAP Response to both methods in tabular form. I have done little editing to fit them in single table and two columns. *echo()* returns "Hello XXXX !" and value of the Resource "name" is "XXXX".

SOAP Request	Soap Response
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"> <soapenv:Header> <wsa:MessageID soapenv:mustUnderstand="0"> uuid:2a75e850-e16f-11d9-b628-fcb3cda304c3 </wsa:MessageID> <wsa:To soapenv:mustUnderstand="0"> http://localhost:8082/wsrf/services/HelloWorldService2 </wsa:To> <wsa:Action soapenv:mustUnderstand="0"> http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourceProperty </wsa:Action> <wsa:From soapenv:mustUnderstand="0"> <wsa:Address> http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous </wsa:Address> </wsa:From> </soapenv:Header> <soapenv:Body> <GetResourceProperty xmlns="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-Resource Properties-1.2-draft-01.xsd" xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld"> ns1:name </GetResourceProperty> </soapenv:Body> </soapenv:Envelope>	<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"> <soapenv:Header> <wsa:MessageID soapenv:mustUnderstand="0"> uuid:88709a20-dcc1-11d9-b359-f53ae5575587 </wsa:MessageID> <wsa:To soapenv:mustUnderstand="0"> http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous </wsa:To> <wsa:Action soapenv:mustUnderstand="0"> http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties/GetResourcePropertyResponse </wsa:Action> <wsa:From soapenv:mustUnderstand="0"> <wsa:Address> http://localhost:8082/wsrf/services/HelloWorldService </wsa:Address> </wsa:From> <wsa:RelatesTo RelationshipType="wsa:Reply" soapenv:mustUnderstand="0"> uuid:2a75e850-e16f-11d9-b628-fcb3cda304c3</wsa:RelatesTo> </soapenv:Header> <soapenv:Body> <GetResourcePropertyResponse xmlns="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"> <ns1:name xmlns:ns1="http://tutorial2.wsrf.dl.ac.uk/helloworld"> Asif Akram Tutorial 2 </ns1:name> </GetResourcePropertyResponse> </soapenv:Body> </soapenv:Envelope>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"> <soapenv:Header> <wsa:MessageID soapenv:mustUnderstand="0"> uuid:88c65b40-dcc1-11d9-ae7c-b8ea2ebe6b67 </wsa:MessageID> <wsa:To soapenv:mustUnderstand="0"> http://localhost:8082/wsrf/services/HelloWorldService2 </wsa:To> <wsa:Action soapenv:mustUnderstand="0"> http://tutorial2.wsrf.dl.ac.uk/helloworld/HelloWorldPortType/EchoRequest </wsa:Action> <wsa:From soapenv:mustUnderstand="0"> <wsa:Address> http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous </wsa:Address> </wsa:From> </soapenv:Header> <soapenv:Body> <echoRequest xmlns="http://tutorial2.wsrf.dl.ac.uk/helloworld"> Rob Allan First Time </echoRequest> </soapenv:Body> </soapenv:Envelope>	<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"> <soapenv:Header> <wsa:MessageID soapenv:mustUnderstand="0"> uuid:88c8cc40-dcc1-11d9-b359-f53ae5575587 </wsa:MessageID> <wsa:To soapenv:mustUnderstand="0"> http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous </wsa:To> <wsa:Action soapenv:mustUnderstand="0"> http://tutorial2.wsrf.dl.ac.uk/helloworld/HelloWorldPortType/EchoResponse </wsa:Action> <wsa:From soapenv:mustUnderstand="0"> <wsa:Address> http://localhost:8082/wsrf/services/HelloWorldService2 </wsa:Address> </wsa:From> <wsa:RelatesTo RelationshipType="wsa:Reply" soapenv:mustUnderstand="0"> uuid:88c65b40-dcc1-11d9-ae7c-b8ea2ebe6b67 </wsa:RelatesTo> </soapenv:Header> <soapenv:Body> <echoResponse xmlns="http://tutorial2.wsrf.dl.ac.uk/helloworld"> Hello Rob Allan First Time ! </echoResponse> </soapenv:Body> </soapenv:Envelope>

Implementing OO based State-full HelloWorld

Note: QName in HelloWorld.java and Client1.java should be same, otherwise it will be error, surprisingly if QName in HelloWorld.java and WSDL are different, client still works. My understanding is all three should be exactly same, better to test again before conclusion.