

**Module code and title: 5COSC010C-Client Service Architecture**  
**Tutorial Manual**

<b>Tutorial title</b>	<b>Data Persistency in Web Services</b>
<b>Tutorial type</b>	<b>Guided and indepenent and non-marked</b>
<b>Week Commencing</b>	<b>21/03/2020</b>

## Contents

<b>Intended Learning Outcomes .....</b>	<b>1</b>
<b>Lesson Plan .....</b>	<b>1</b>
<b>Indicative mapping of learning outcomes to tasks .....</b>	<b>1</b>
<b>TASKS to be Performed under the instruction of the Tutor .....</b>	<b>2</b>

## Intended Learning Outcomes

- ILO1: To **recall** the steps of creating a service producer and consumer
- ILO2: To **identify** the need of data persistency in service producer
- ILO3: To **implement** data persistency using serialization
- ILO4: To **evaluate** the effectiveness by testing the solution

## Lesson Plan

What we will learn today	5 mts
Explanation of the lab sheet	10 mts
Execute the steps in the lab sheet	60 mts
Feedback on implemented code	30 mts
Minute Paper	5 mts
Upload the code to Blackboard	5 mts

## Indicative mapping of learning outcomes to tasks

Tasks 1 to 27	= ILO1
Task 28 to 29	= ILO2
Task 30 to 37	= ILO3
Task 38 to 41	= ILO4

## **TASKS to be Performed under the instruction of the Tutor**

- 1) Start Netbeans (8.0.2 EE or 8.2 EE) in your system.
- 2) Create a new Web Application project in Netbeans (Figure 1 to 3). We will call this project ***Tutorial8***.

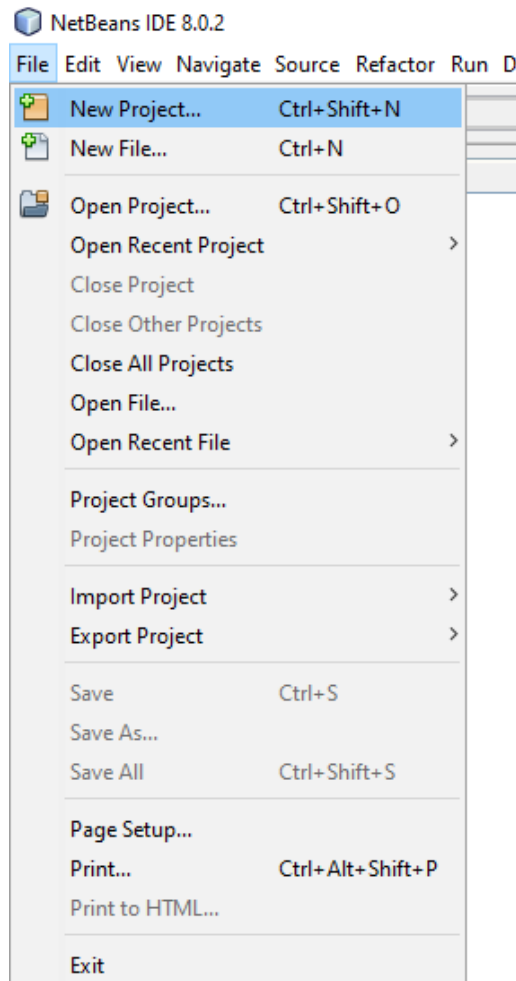


Figure 1, Create a Web Application Project in NetBeans (1)

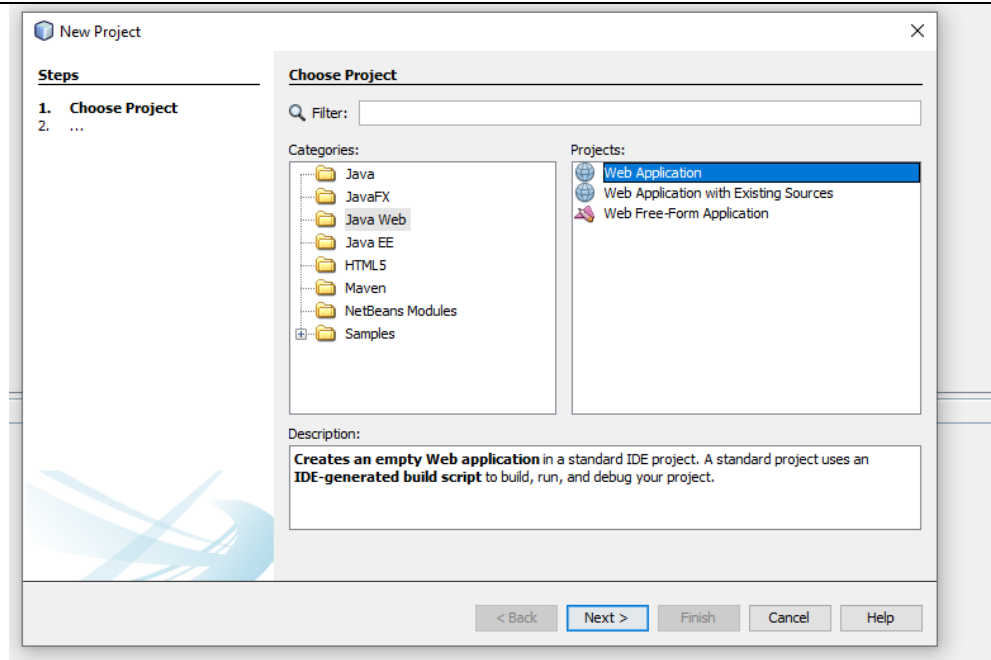


Figure 2, Create a Web Application Project in NetBeans (2)

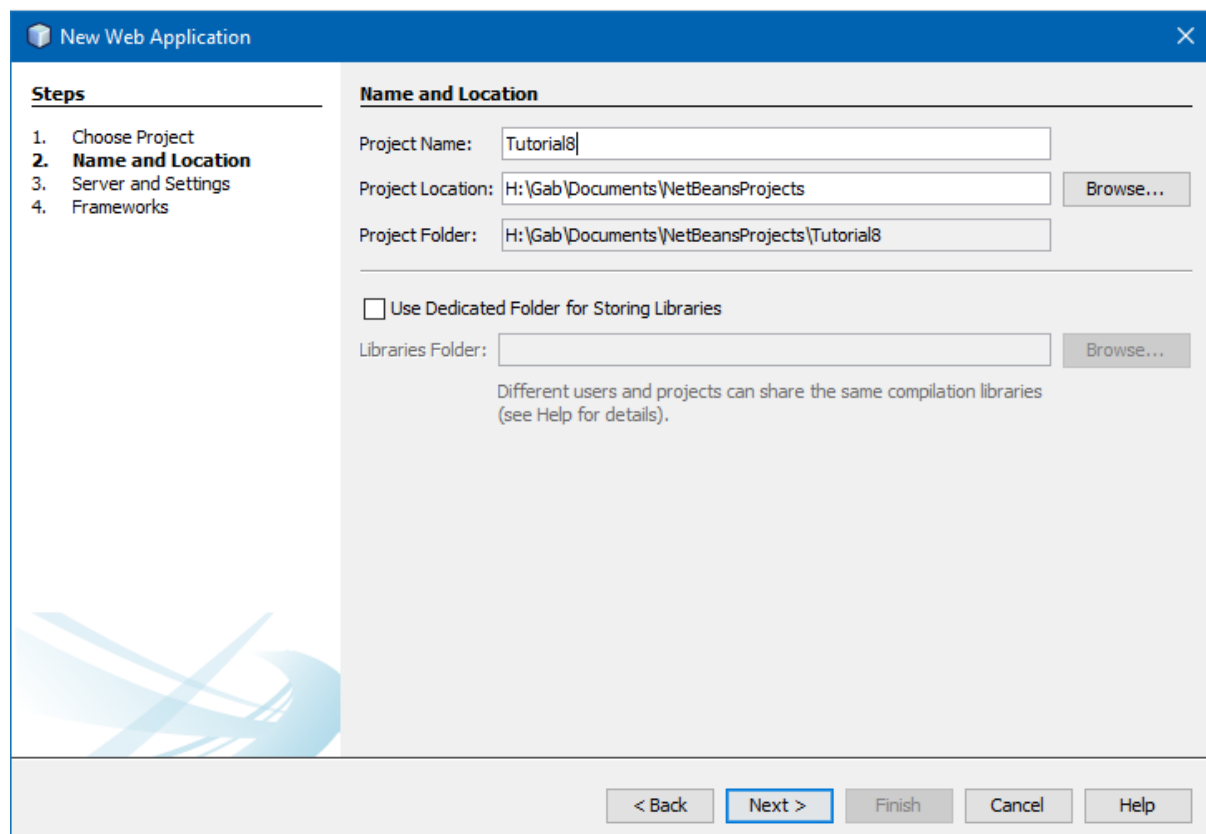


Figure 3, Create a Web Application Project in NetBeans (3)

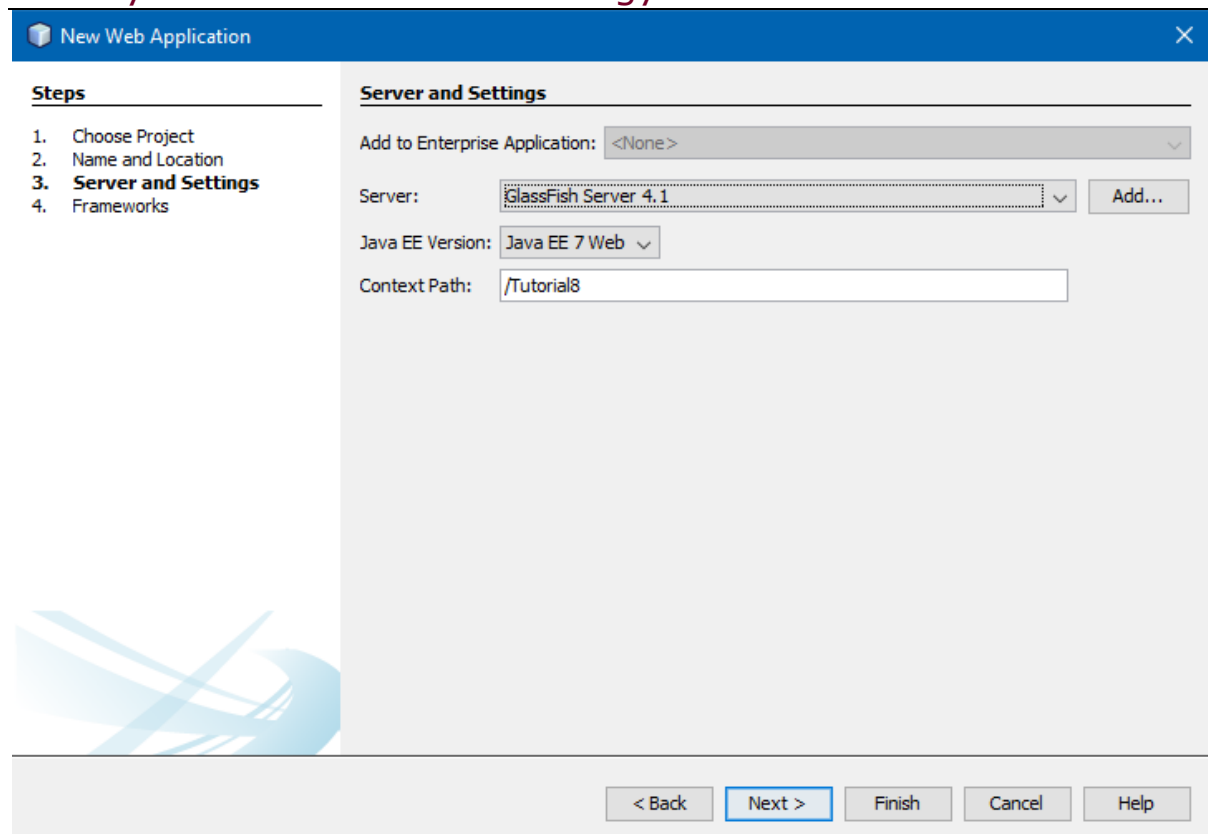


Figure 4, Create a Web Application Project in NetBeans (4),

Select GlassFish Server

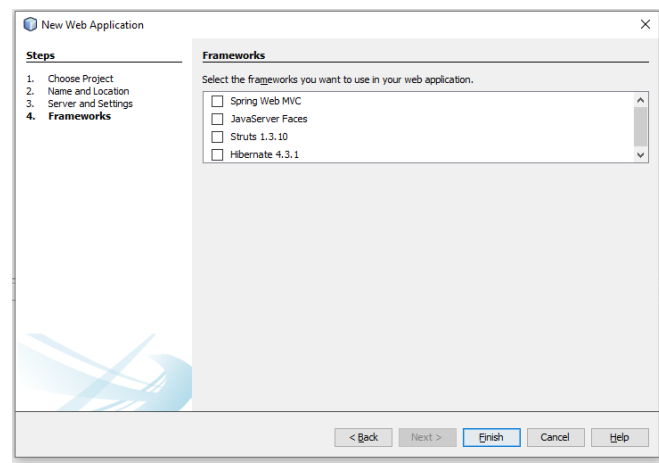
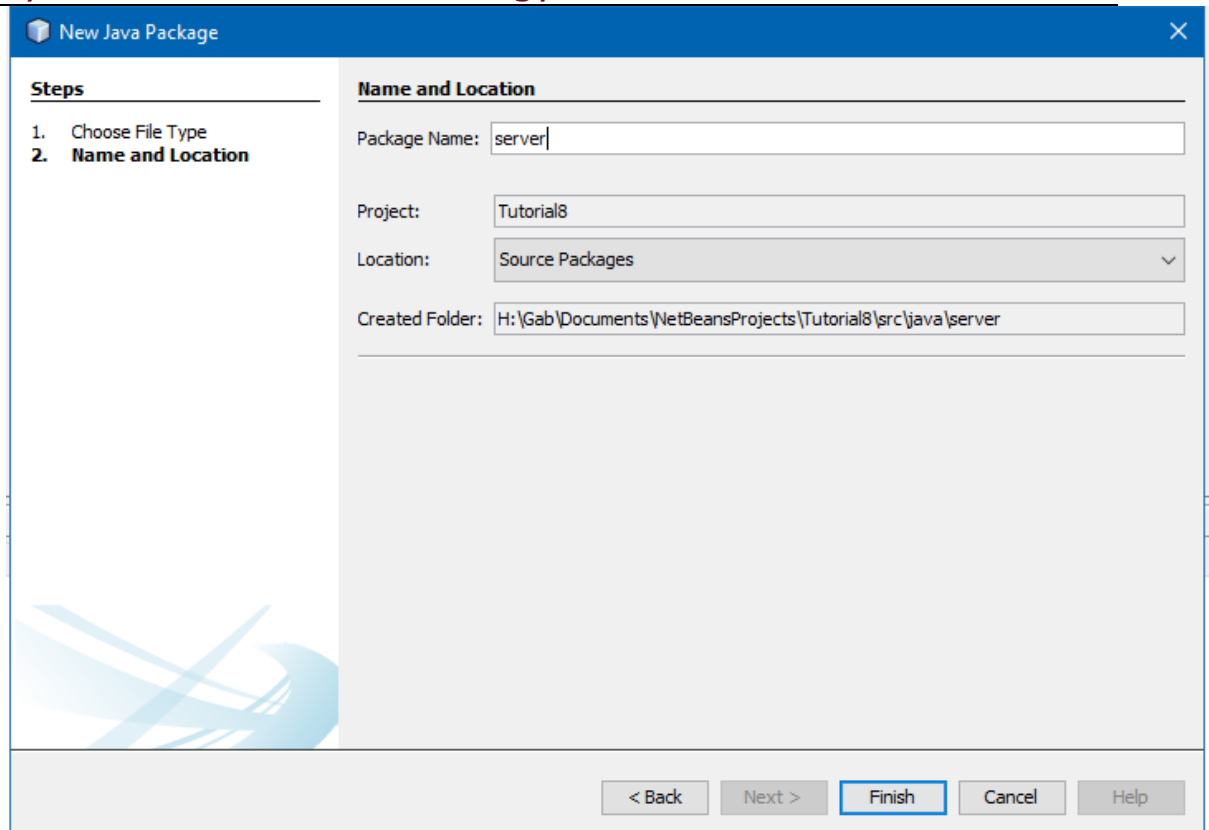


Figure 5, Create a Web Application Project in NetBeans (5) - Do not select any framework

- 3) Create a new Java package in the Web Application where we will put our server, called it server (Figure 6, Creating a server Java Package).



*Figure 6, Creating a server Java Package*

- 4) Now, we can create a proper Web Service, we will call it Tutorial8WebService (Figure 7, Create the Web Service)

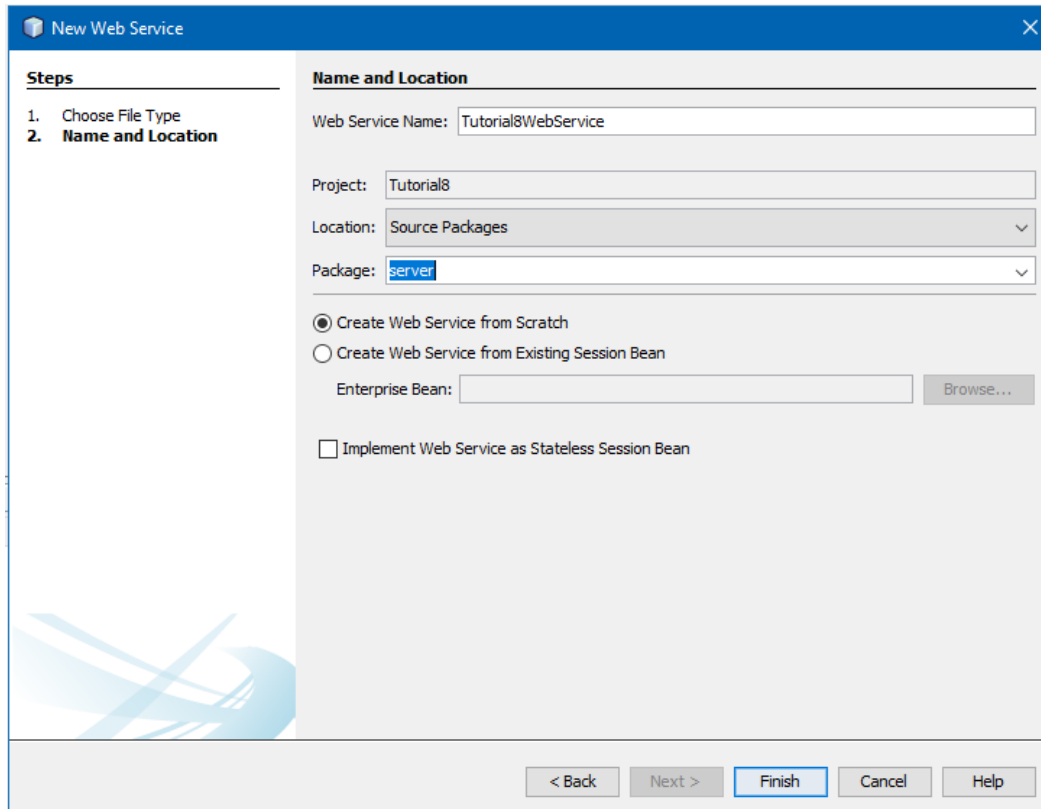


Figure 7, Create the Web Service

- 5) You can observe that NetBeans creates a standard method called Hello which returns (as a String) the message sent (passed as a String).
- 6) We can use the design view of the Web Service to add and remove methods, you can see the hello method, its parameters and the return type (Figure 8, Design view of the Web Service and Figure 9, Details of the functionalities of the Web Services Interface). This method is added automatically by the Web Services creator. We are not going to use the hello method so you can remove it if you want.

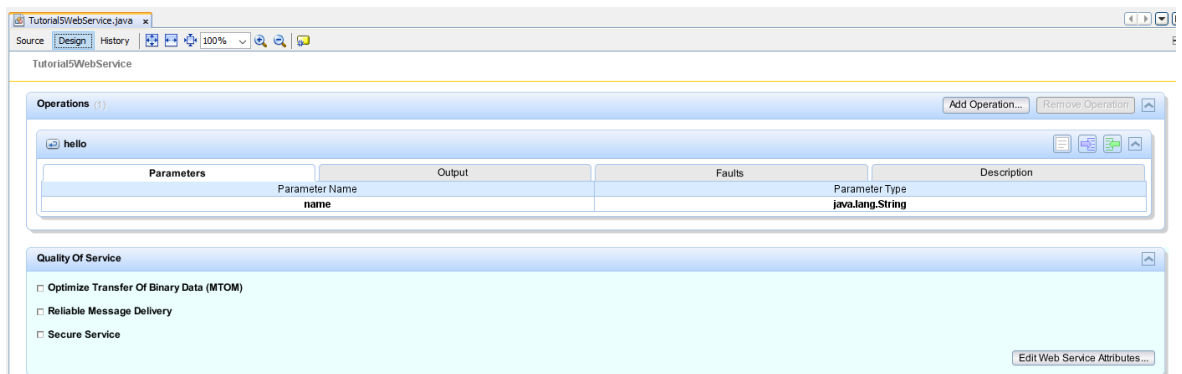


Figure 8, Design view of the Web Service

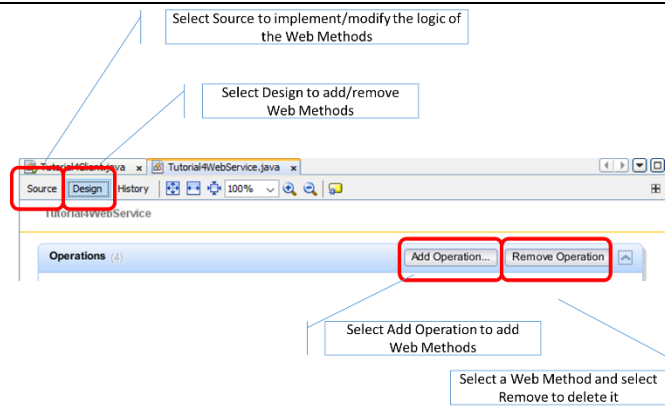


Figure 9, Details of the functionalities of the Web Services Interface

- 7) We use the AddOperation Button to add our usual isConnected method that returns true if the server is connected (Figure 10, Adding the isConnected method to the Serve). Netbeans will create an empty method which you have to complete in the next step.

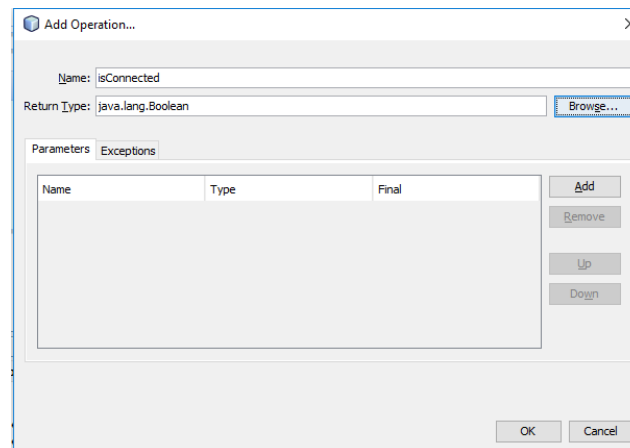


Figure 10, Adding the isConnected method to the Server

- 8) Add the usual simple logic of the usual isConnected method in the logic of the Web Service (Figure 11, Logic on the Web Service Method isConnected.)

```
/**
 * Web service operation
 */
@WebMethod(operationName = "isConnected")
public Boolean isConnected() {
    //TODO write your implementation code here:
    System.out.println("[SERVER] - Testing Connection...");
    return true;
}
```

Figure 11, Logic on the Web Service Method isConnected.

- 9) Now we deploy the server on the Glassfish server engine. **ALWAYS REMEMBER TO REDEPLOY THE SERVER EVERY TIME YOU CHANGE ITS CODE.**
- 10) NetBeans offers us a testing tool without even to have to write a client (NetBeans will write a simple client in a browser)(Figure 12, Testing the Web Service). Test the isConnected method, it returns true, that is correct! You can also check the server log to see if it is correct.

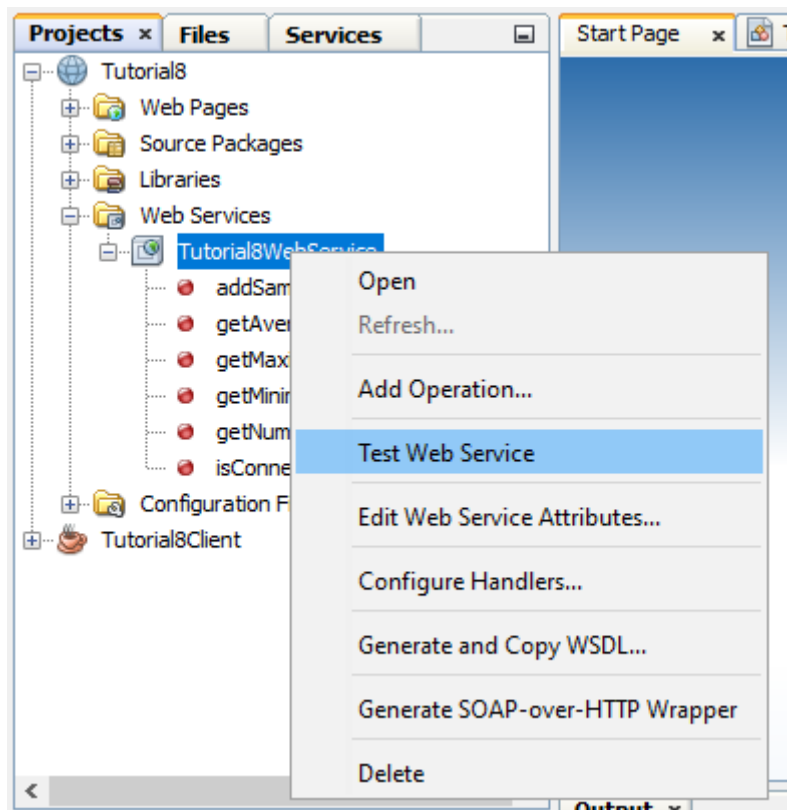


Figure 12, Testing the Web Service

- 11) Now we create a client, this is going to be a separate Java Application Project (**NOT A WEB APPLICATION, A SIMPLE JAVA APPLICATION**) (Figure 13, Creating the Client (Step 1) and Figure 14, Creating the Client (Step 2))



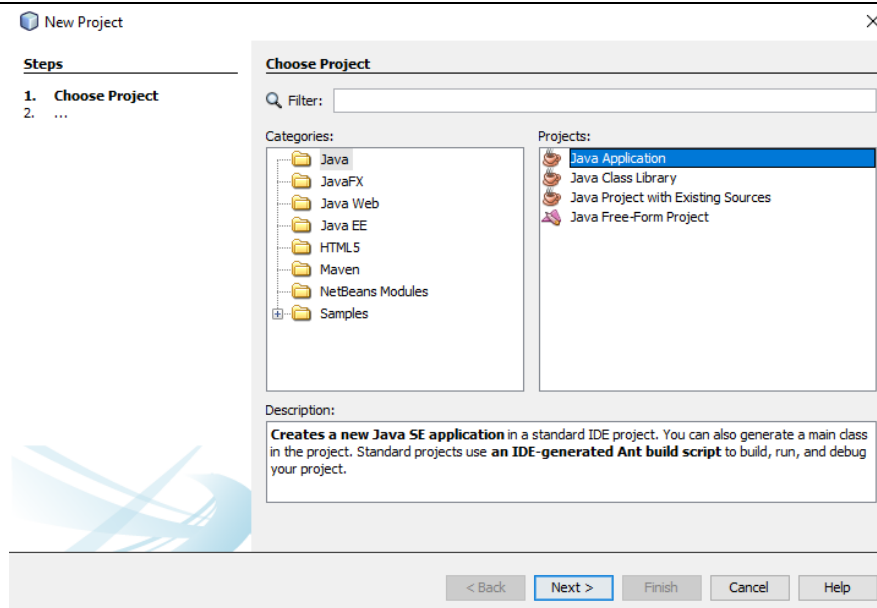


Figure 13, Creating the Client (Step 1)

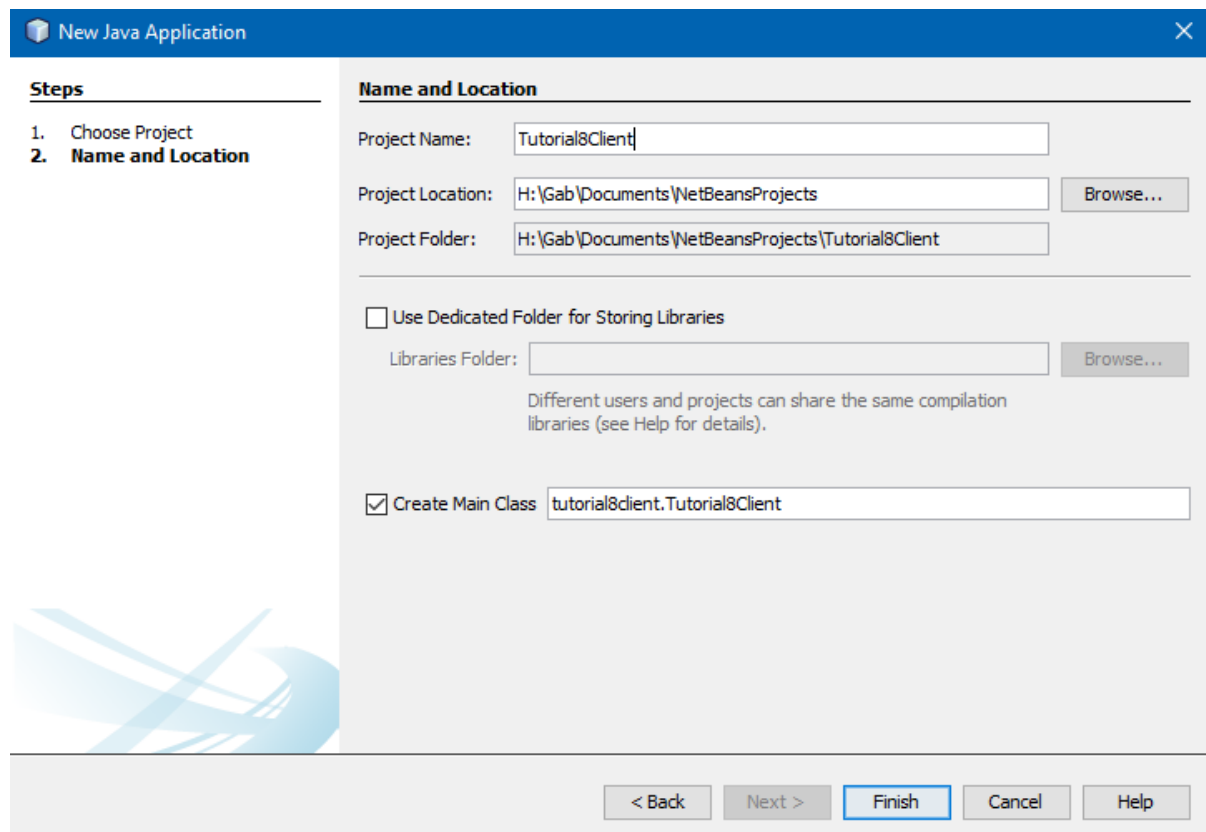


Figure 14, Creating the Client (Step 2)

- 12) Now we build a communication system between the client and the server. In order to do that, we have to create what is called a client stub (or Web Service Client) on the client which will be able to communicate with the Server. NetBeans will create behind the scenes all the code that handles the communication. Select the Web Service under Project and

leave the package empty (Figure 15, Creating the Web Service Client (Step 1)), and, Figure 16, Creating the Web Service Client (Step 2)).

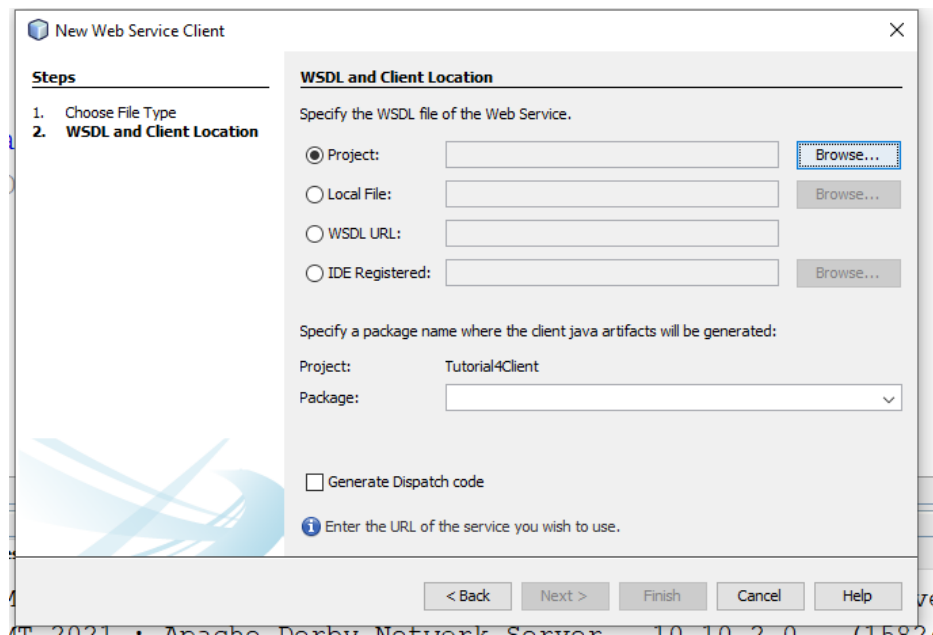


Figure 15, Creating the Web Service Client (Step 1)

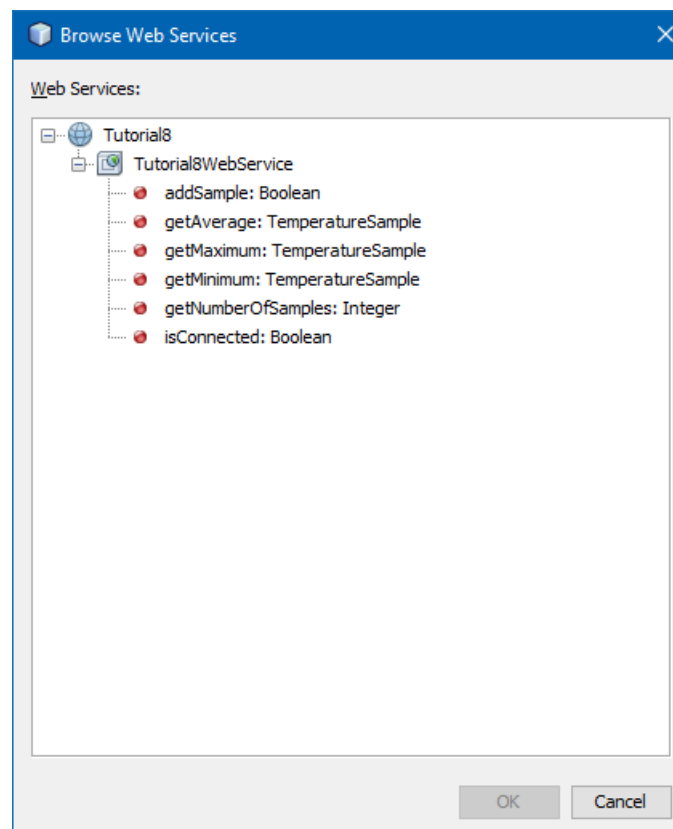


Figure 16, Creating the Web Service Client (Step 2)

- 13) You can notice that now on the client side, we have a representation of the server web methods. These are called stubs and they are an interface to the network that will handle the communication with the server (Figure 17, Client Stubs).

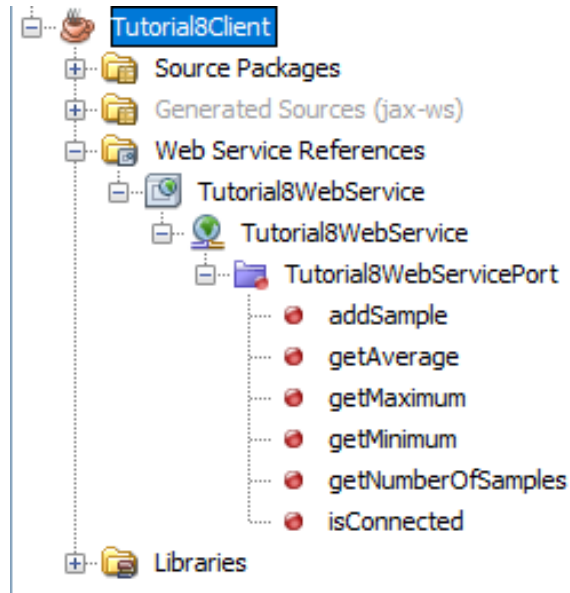


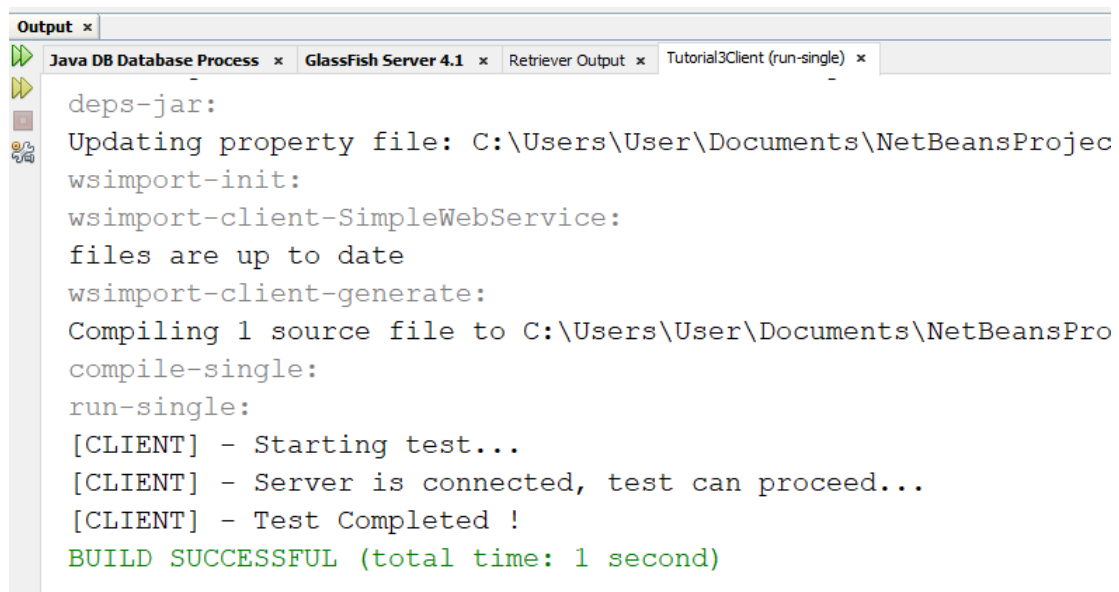
Figure 17, Client Stubs

- 14) If you want to use one of the remote methods on the server, you just have to drag and drop the icon of the method in the client code where you want to use it, this will create a client stub: a method on the client capable of connecting to the internet to connect to a server.
- 15) Now you can invoke your client stub (which will in turn connect to the network and call the server) from your code. Develop the Client-side code by creating an instance of Tutorial8Client and invoking a execute method. Develop the execute method to connect to the remote server (Figure 18, Client code)

```
private void execute()
{
    System.out.println("[CLIENT] - Starting Test...");
    if(isConnected())
    {
        System.out.println("[CLIENT] - Server is connected, continuing test...");
    }
    else
    {
        System.out.println("[CLIENT] - Server is NOT connected, test failed ! ");
    }
    System.out.println("[CLIENT] - Test Completed");
}
```

Figure 18, Client code, execute method

- 16) **ONLY If the client code does not compile because it cannot find the classes in the server package of the client stubs**, you can find the code under Generated Sources (jax-ws). Simply copy and paste the entire java package server into your client code.
- 17) Now we are ready to run the client and see if it is really capable of connecting to a server running in a separate project.
- 18) Observe how the logging calls (system.out.println(...)) from the Server and the Client appear in different stdout streams (one from the GlassFish Server and one from the Client).



```
Output x
Java DB Database Process x GlassFish Server 4.1 x Retriever Output x Tutorial3Client (run-single) x
deps-jar:
Updating property file: C:\Users\User\Documents\NetBeansProjec
wsimport-init:
wsimport-client-SimpleWebService:
files are up to date
wsimport-client-generate:
Compiling 1 source file to C:\Users\User\Documents\NetBeansPro
compile-single:
run-single:
[CLIENT] - Starting test...
[CLIENT] - Server is connected, test can proceed...
[CLIENT] - Test Completed !
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 19, Client Log

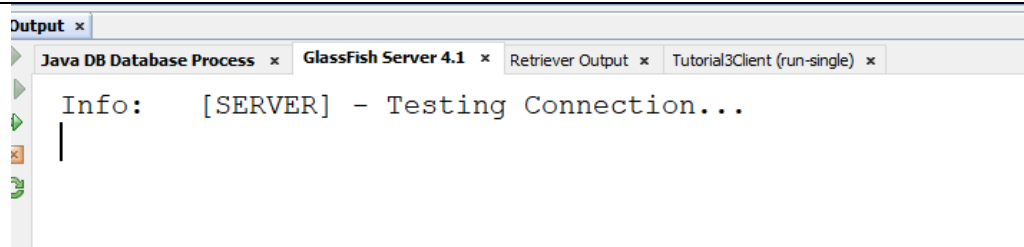


Figure 20, Server Log

- 19) Now, we develop the server further. We want to implement a simple temperature recording system. First we concentrate on how to implement a temperature. A Double type would be a simple choice but it would require to re-write the code if we wanted to add other information (such as time, place and unit of measure). To make our code extendable and maintainable, we create a user-defined type called TemperatureSample implemented as Class inside the server package which contains a private field Double, getters and setters (Figure 21, TemperatureSample class). You can create automatically getters and setters by right-clicking on the code, insert code, getters and setters. We also add the toString() method for logging reasons.

```
/**
 *
 * @author Gab
 */
public class TemperatureSample {
    Double value = null;

    public Double getValue() {
        return value;
    }

    public void setValue(Double value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "TemperatureSample{" + "value=" + value + '}';
    }
}
```

Figure 21, TemperatureSample class

- 20) Now we add to the server an addSample Web Method that will allow us to add TemperatureSamples to the server (Figure 22, create WebMethod skeleton of addSample step 1, Figure 23, create

WebMethod skeleton of addSample step 2, **Error! Reference source not found.**).

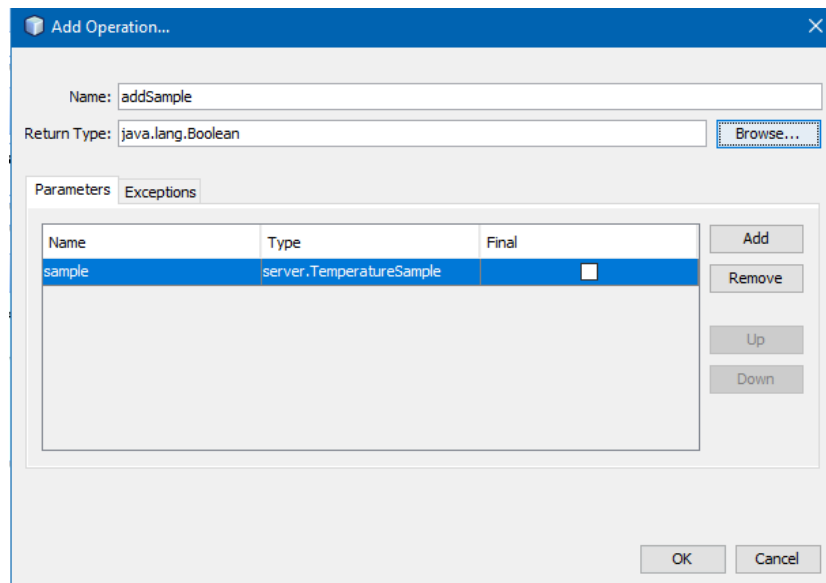


Figure 22, create WebMethod skeleton of addSample step 1

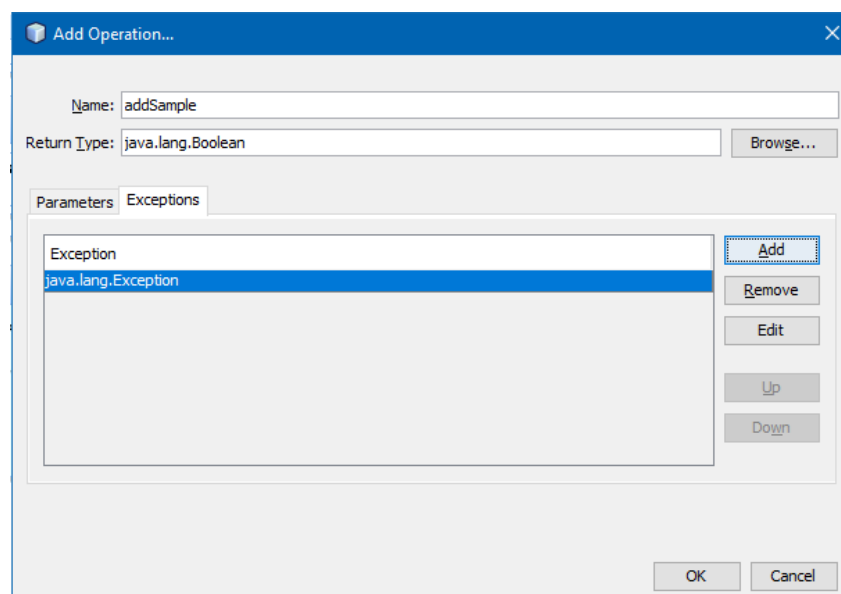


Figure 23, create WebMethod skeleton of addSample step 2

- 21) To store the Samples on the server, we declare an ArrayList of Temperature Samples (Figure 24, Declaring an ArrayList on the Server) which we use in the addSample Web Method.

```
/**
 *
 * @author Gab
 */
@WebService(serviceName = "Tutorial5WebService")
public class Tutorial5WebService
{
    ArrayList<TemperatureSample> samples = new ArrayList();
}
```

Figure 24, Declaring an ArrayList on the Server

- 22) Implement the logic of the addSample method to add the samples to the server (Figure 25, implement logic of the WebMethod addSample)

```
/**
 * Web service operation
 * @param sample
 * @return
 * @throws java.lang.Exception
 */
@WebMethod(operationName = "addSample")
public Boolean addSample(@WebParam(name = "sample") server.TemperatureSample sample) throws Exception {
    if(sample == null)
        throw new Exception();

    System.out.println("[SERVER] - addSample(" + sample + ") to " + samples);
    samples.add(sample);
    System.out.println("[SERVER] - Now samples are " + samples);
    return true;
}
```

Figure 25, implement logic of the WebMethod addSample

- 23) Add another Web Method: getNumberOfSamples which returns the size of the samples ArrayList, we will use it to test the Server. (Figure 26, create and implement the getNumberOfSamples Web Method.)

```
/**
 * Web service operation
 * @return
 */
@WebMethod(operationName = "getNumberOfSamples")
public Integer getNumberOfSamples() {
    System.out.println("[SERVER] - getNumberOfSamples()");
    return samples.size();
}
```

Figure 26, create and implement the getNumberOfSamples Web Method.

- 24) Finally, create and implement the `getMaximum` Web Method which returns the maximum of all samples. (Figure 27, create and implement the `getMaximum` Web Method.)

```
/**
 * Web service operation
 * @return
 * @throws java.lang.Exception
 */
@WebMethod(operationName = "getMaximum")
public TemperatureSample getMaximum() throws Exception {
    if(samples.isEmpty())
        throw new Exception();

    TemperatureSample max = new TemperatureSample();
    max = samples.get(0);
    for(int i = 0; i < samples.size(); i++)
        if(samples.get(i).getValue() > max.getValue())
            max = samples.get(i);

    return max;
}
```

Figure 27, create and implement the `getMaximum` Web Method.

- 25) Now we want to test the new Web Server functionalities from the Client. First we must deploy the server, then refresh the Client Stubs (Figure 28, Refreshing the Client Stubs). You can observe that the Web Service Reference has now been updated.

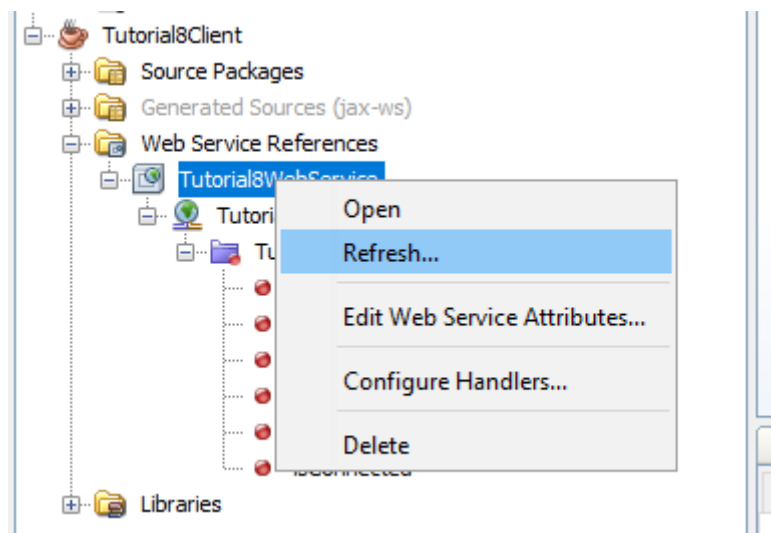


Figure 28, Refreshing the Client Stubs



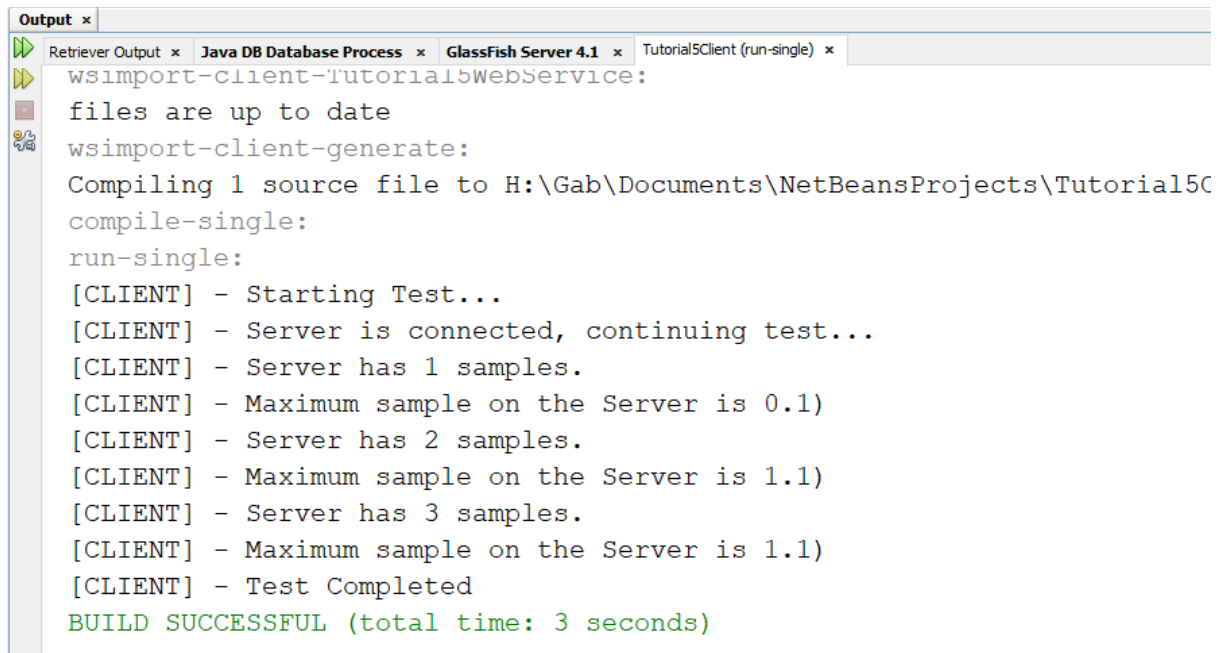
- 26) Copy the client stubs into your client code and invoke them. (Figure 29, Test the new Web Methods from the client.) **ONLY If the client code does not compile because it cannot find the classes in the server package of the client stubs**, you can find the code under Generated Sources (jaws). Simply delete the server package in your source code and re-paste the entire java package server into from generated sources into your client code.

```
if (isConnected()) {
    System.out.println("[CLIENT] - Server is connected, continuing test...");
    TemperatureSample s = new TemperatureSample();
    s.setValue(0.1);
    try {
        addSample(s);
        System.out.println("[CLIENT] - Server has " + getNumberOfSamples() + " samples");
        System.out.println("[CLIENT] - Maximum sample on the Server is " + getMaximumSample());
    } catch (Exception_Exception ex) {
        Logger.getLogger(Tutorial5Client.class.getName()).log(Level.SEVERE, null, ex);
    }
    s.setValue(1.1);
    try {
        addSample(s);
        System.out.println("[CLIENT] - Server has " + getNumberOfSamples() + " samples");
        System.out.println("[CLIENT] - Maximum sample on the Server is " + getMaximumSample());
    } catch (Exception_Exception ex) {
        Logger.getLogger(Tutorial5Client.class.getName()).log(Level.SEVERE, null, ex);
    }

    s.setValue(-1.1);
    try {
        addSample(s);
        System.out.println("[CLIENT] - Server has " + getNumberOfSamples() + " samples");
        System.out.println("[CLIENT] - Maximum sample on the Server is " + getMaximumSample());
    } catch (Exception_Exception ex) {}
    Logger.getLogger(Tutorial5Client.class.getName()).log(Level.SEVERE, null, ex);
}
```

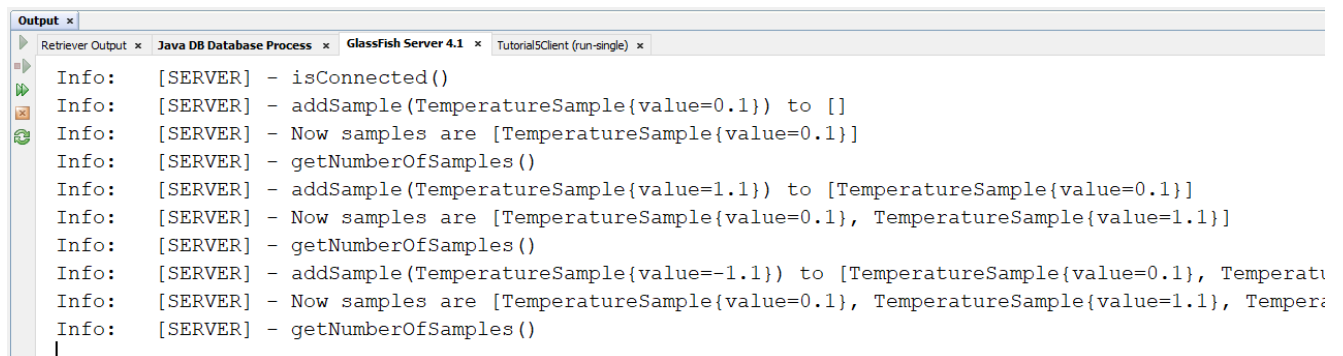
Figure 29, Test the new Web Methods from the client.

- 27) Test the client and check the logs on both sides: client and server.



```
Retriever Output x Java DB Database Process x GlassFish Server 4.1 x Tutorial5Client (run-single) x
wsimport-client-Tutorial5WebService:
files are up to date
wsimport-client-generate:
Compiling 1 source file to H:\Gab\Documents\NetBeansProjects\Tutorial5C
compile-single:
run-single:
[CLIENT] - Starting Test...
[CLIENT] - Server is connected, continuing test...
[CLIENT] - Server has 1 samples.
[CLIENT] - Maximum sample on the Server is 0.1)
[CLIENT] - Server has 2 samples.
[CLIENT] - Maximum sample on the Server is 1.1)
[CLIENT] - Server has 3 samples.
[CLIENT] - Maximum sample on the Server is 1.1)
[CLIENT] - Test Completed
BUILD SUCCESSFUL (total time: 3 seconds)
```

Figure 30, Client Log



```
Retriever Output x Java DB Database Process x GlassFish Server 4.1 x Tutorial5Client (run-single) x
Info: [SERVER] - isConnected()
Info: [SERVER] - addSample(TemperatureSample{value=0.1}) to []
Info: [SERVER] - Now samples are [TemperatureSample{value=0.1}]
Info: [SERVER] - getNumberOfSamples()
Info: [SERVER] - addSample(TemperatureSample{value=1.1}) to [TemperatureSample{value=0.1}]
Info: [SERVER] - Now samples are [TemperatureSample{value=0.1}, TemperatureSample{value=1.1}]
Info: [SERVER] - getNumberOfSamples()
Info: [SERVER] - addSample(TemperatureSample{value=-1.1}) to [TemperatureSample{value=0.1}, Temperat
Info: [SERVER] - Now samples are [TemperatureSample{value=0.1}, TemperatureSample{value=1.1}, Temper
Info: [SERVER] - getNumberOfSamples()
```

Figure 31, Server Log

- 28) Note that if you re-execute the client, the number of sample of the server grows (the server is now stateful). To reset the server, redeploy it.
- 29) Now, we want to implement data persistency on the server to overcome this. Temperature sample must be kept even if the service is redeployed or the service container is rebooted. In order to do so, we will use a simple technology, that of Serialization (<https://www.javatpoint.com/serialization-in-java>).
- 30) To serialize and deserialize objects (the ArrayList of TemperatureSample) to files, we need to modify the server.
- 31) First we add a variable on the Tutorial8WebService class to define what will be the name of the file we will use for Serialization and

Deserialization (Figure 32, Declaring a variable with the Serialization/Deserialization file.).

```
@WebService(serviceName = "Tutorial8WebService")
public class Tutorial8WebService {

    ArrayList<TemperatureSample> samples = new ArrayList();
    String persistencyFileName = "samples.txt";
```

Figure 32, Declaring a variable with the Serialization/Deserialization file.

- 32) Then we add and invoke a method called `saveStatus` which serializes the `ArrayList` everytime we add a sample (Figure 33, modifying the logic of the `addSample` method on the Server.)

```
@WebMethod(operationName = "addSample")
public Boolean addSample(@WebParam(name = "sample") server.TemperatureSample sample) throws Exception {

    if (sample == null) {
        throw new Exception();
    }

    System.out.println("[SERVER] - addSample(" + sample + ") to " + samples);
    samples.add(sample);
    System.out.println("[SERVER] - Now samples are " + samples);

    System.out.println("[SERVER] - serializing " + sample + " to " + persistencyFileName);
    saveStatus();
}
```

Figure 33, modifying the logic of the `addSample` method on the Server.

- 33) Then we implement the method `saveStatus` (Figure 34, implementing the `saveStatus` method on the Server.)

```
private void saveStatus() throws FileNotFoundException, IOException {
    System.out.println("[SERVER] - saveStatus()");

    FileOutputStream fout = new FileOutputStream(persistencyFileName);
    System.out.println(fout.toString());

    try (ObjectOutputStream out = new ObjectOutputStream(fout)) {
        out.writeObject(samples);
        out.flush();
        //closing the stream
    }
    System.out.println("[SERVER] - saveStatus(), success");
}
```

Figure 34, implementing the `saveStatus` method on the Server.

- 34) So far we have written the logic to save the status (all the Temperature Samples) but we never use that information. In order to do so, we have to implement several steps.
- 35) First we override the constructor of Tutorial8WebService to invoke a method which loads the data when the class is constructed.

```
ArrayList<TemperatureSample> samples  
String persistencyFileName = "sample
```

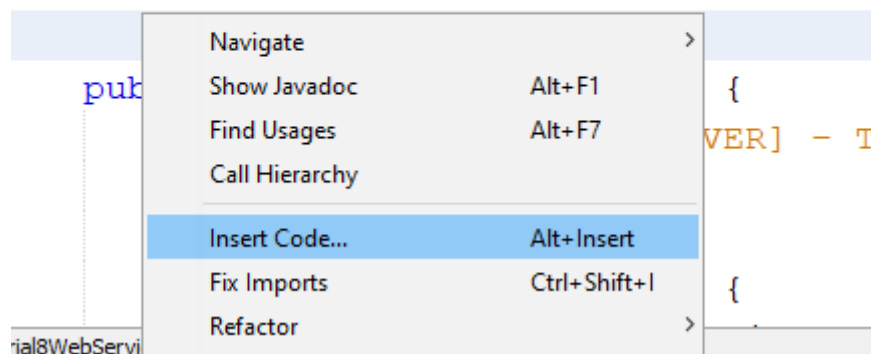


Figure 35, Overriding the default constructor 1

```
ArrayList<TemperatureSa  
String persistencyFileN
```

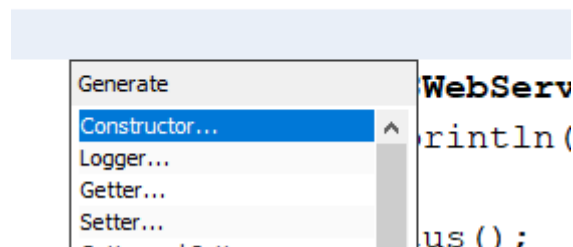


Figure 36, Overriding the default constructor 2

- 36) Then we invoke the loadStatus method from the constructor (Figure 37, Invoking the loadStatus method from the Constructor)

```
public Tutorial8WebService() {  
    System.out.println("[SERVER] - Tutorial8WebService()");  
    try {  
        loadStatus();  
    } catch (IOException ex) {  
        Logger.getLogger(Tutorial8WebService.class.getName()).log(Level.SEVERE, null, ex);  
    } catch (ClassNotFoundException ex) {  
        Logger.getLogger(Tutorial8WebService.class.getName()).log(Level.SEVERE, null, ex);  
    }  
}
```

Figure 37, Invoking the loadStatus method from the Constructor

37) Finally, we implement the loadStatus method (Figure 38, implementing the loadStatus method.)

```
private void loadStatus() throws FileNotFoundException, IOException, ClassNotFoundException {  
  
    System.out.println("[SERVER] - loadStatus()");  
  
    File f = new File(persistencyFileName);  
    System.out.println("----" + f.getAbsolutePath());  
  
    FileInputStream file;  
    file = new FileInputStream(persistencyFileName);  
    ObjectInputStream in = new ObjectInputStream(file);  
  
    // Method for deserialization of object  
    samples = (ArrayList<TemperatureSample>) in.readObject();  
  
    in.close();  
    file.close();  
  
    System.out.println("Object has been deserialized ");  
}
```

Figure 38, implementing the loadStatus method.

38) Observe that, when you try to execute the modified server, an exception is thrown. That is because, TemperatureSample is not a Serializable object (Figure 39, Serialization Exception).

```
Info: [SERVER] - serializing TemperatureSample{value=0.1, time=Mon Mar 08 11:19:06 CET 2021, pla  
Info: [SERVER] - saveStatus()  
Severe: server.TemperatureSample  
java.io.NotSerializableException: server.TemperatureSample  
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)  
at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)  
at java.util.ArrayList.writeObject(ArrayList.java:766)  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
at java.lang.reflect.Method.invoke(Method.java:498)  
at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:1128)
```

Figure 39, Serialization Exception

- 39) To solve this, we just have to declare that TemperatureSample implements the Serializable interface.

```
/**
 *
 * @author Gab
 */
public class TemperatureSample implements Serializable{
    Double value = null;
    Date time = null;
    String place = null;
```

*Figure 40, Implementing the Serializable Interface*

- 40) Try to re-deploy the server. You will observe that the past samples are loaded into the server.
- 41) Finally, export both the projects (client and server) as zip file on Netbeans.