



Avaya Open Interfaces

Open Queue Tutorial

Open Queue Tutorial

This tutorial describes how to utilise the features provided by the Open Interfaces Open Queue Web Services. It will show how to generate the necessary proxies to create a simple application that will authenticate itself with the service and create a new Open Queue contact.

Once the application is launched it will login into the Open Queue web service and create Open Queue contact for each row in the *intrinsic.csv* file.

The main procedures in this tutorial are:

- Action 1 : Create Project
- Action 2 : Create Web Service Proxy classes
- Action 3 : Create the Application

Tools Used

This tutorial was created using

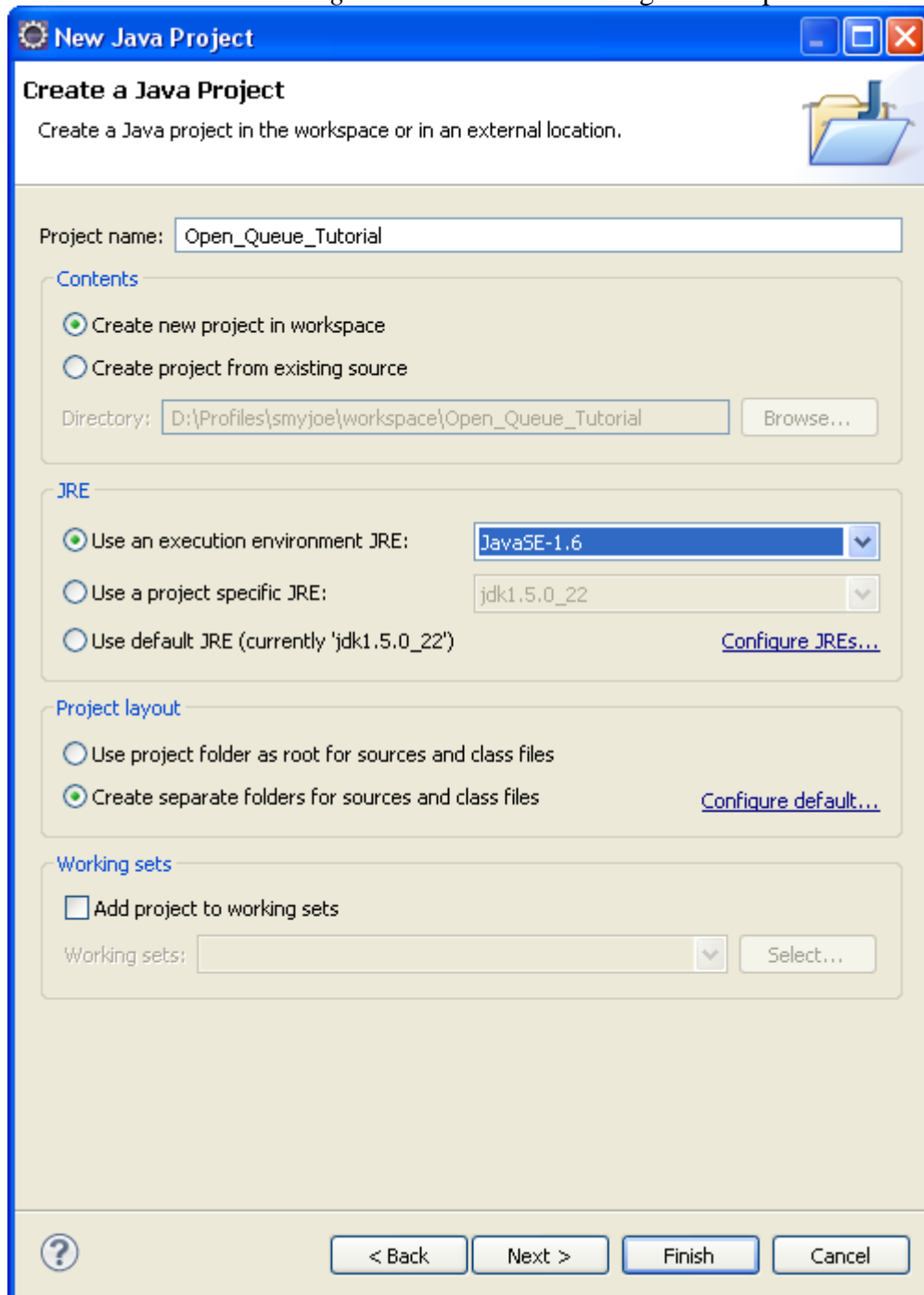
- Avaya Aura Contact Center 7.0
- Apache CXF 2.2.5
- Eclipse Galileo

Creating the Application

Action 1 – Create Project

Launch Eclipse and create a new Java Project using the default settings.

1. Open Eclipse IDE
2. Select File > New > Java Project
3. Select Next
4. Enter Project Name *Open_Queue_Tutorial* and select a desired location
5. Select Next, Next and Finished
6. Add a new src folder : Right click on the project tab, Select New > Src Folder. Call this new folder *autogen.src*. This is where our generated proxies will be stored.

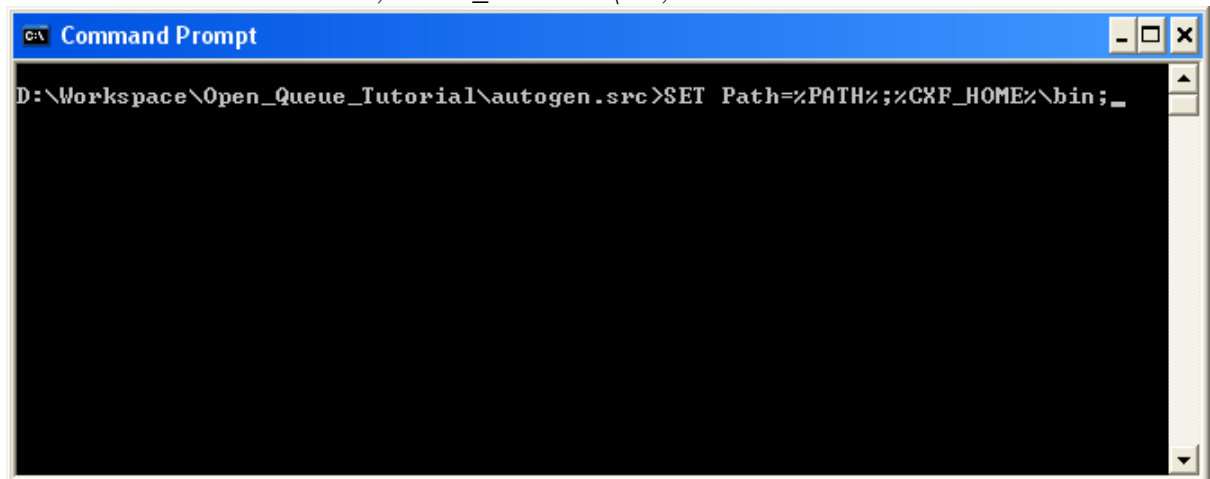


Action 2 – Create Web Service Proxy classes

Applications can communicate with web service directly using SOAP messaging, for simplicity we are going to generate proxy classes to abstract us away from the underlying messaging. There are many different tools available to assist a developer in generating the client proxies but in this example we are going to use CXF.

For this step we require that CXF is installed on your system and the CXF_HOME is configured. Please refer to the CXF documentation for more details.

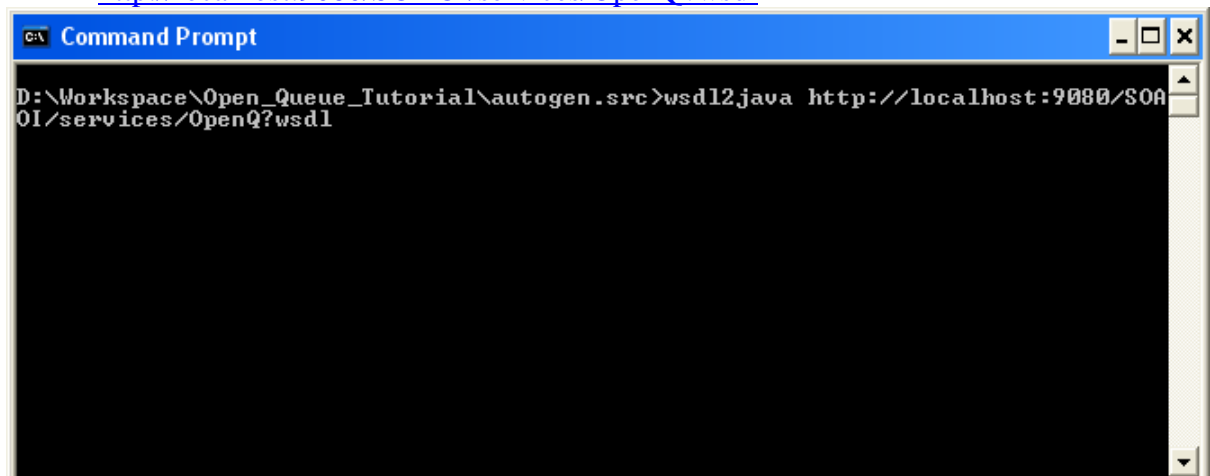
- Open a command window in the newly created *autogen.src* directory.
- Set the system PATH to the CXF bin directory
 - a. SET Path=%PATH%;%CXF_HOME%\bin;



```
C:\ Command Prompt
D:\Workspace\Open_Queue_Tutorial\autogen.src>SET Path=%PATH%;%CXF_HOME%\bin;
```

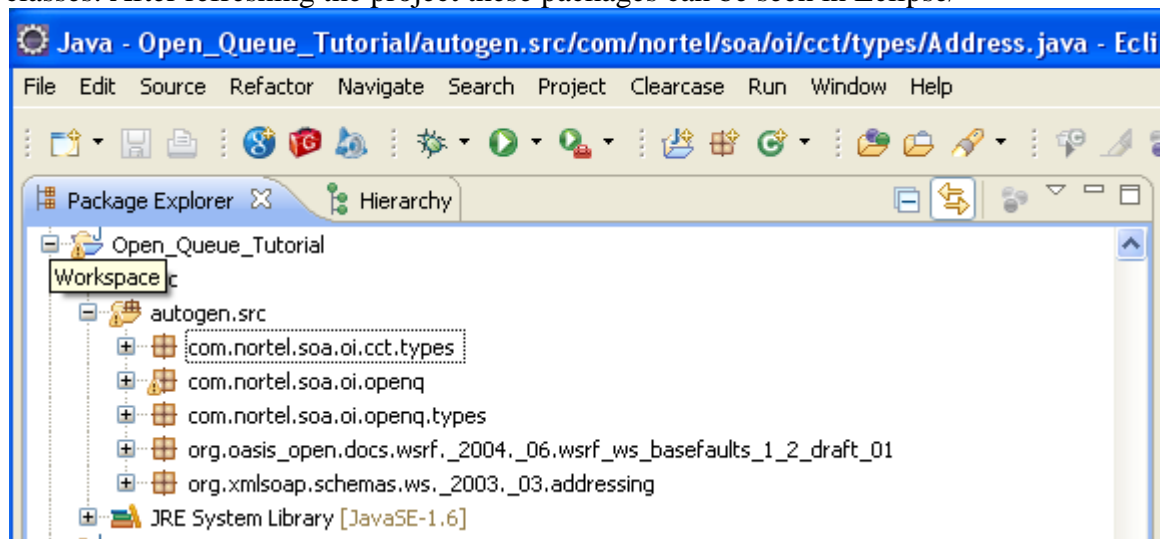
- To generate the Proxies we use a tool called **wsdl2java**. This tool will generate the proxies from the downloaded *WSDL* or by supplying the service URL. For this application we will supply the url

<http://localhost:9080/SOAOI/services/OpenQ?wsdl>



```
C:\ Command Prompt
D:\Workspace\Open_Queue_Tutorial\autogen.src>wsdl2java http://localhost:9080/SOAOI/services/OpenQ?wsdl
```

- When the tool has completed you will see two new packages with a series of java classes. After refreshing the project these packages can be seen in Eclipse/



Action 3 – Create the Application.

In this Step we are going to create a main class that will process a comma-separated values file to create a series of contacts using the Open Queue web service. To call the Open Queue web service you will typically need to perform the following steps.

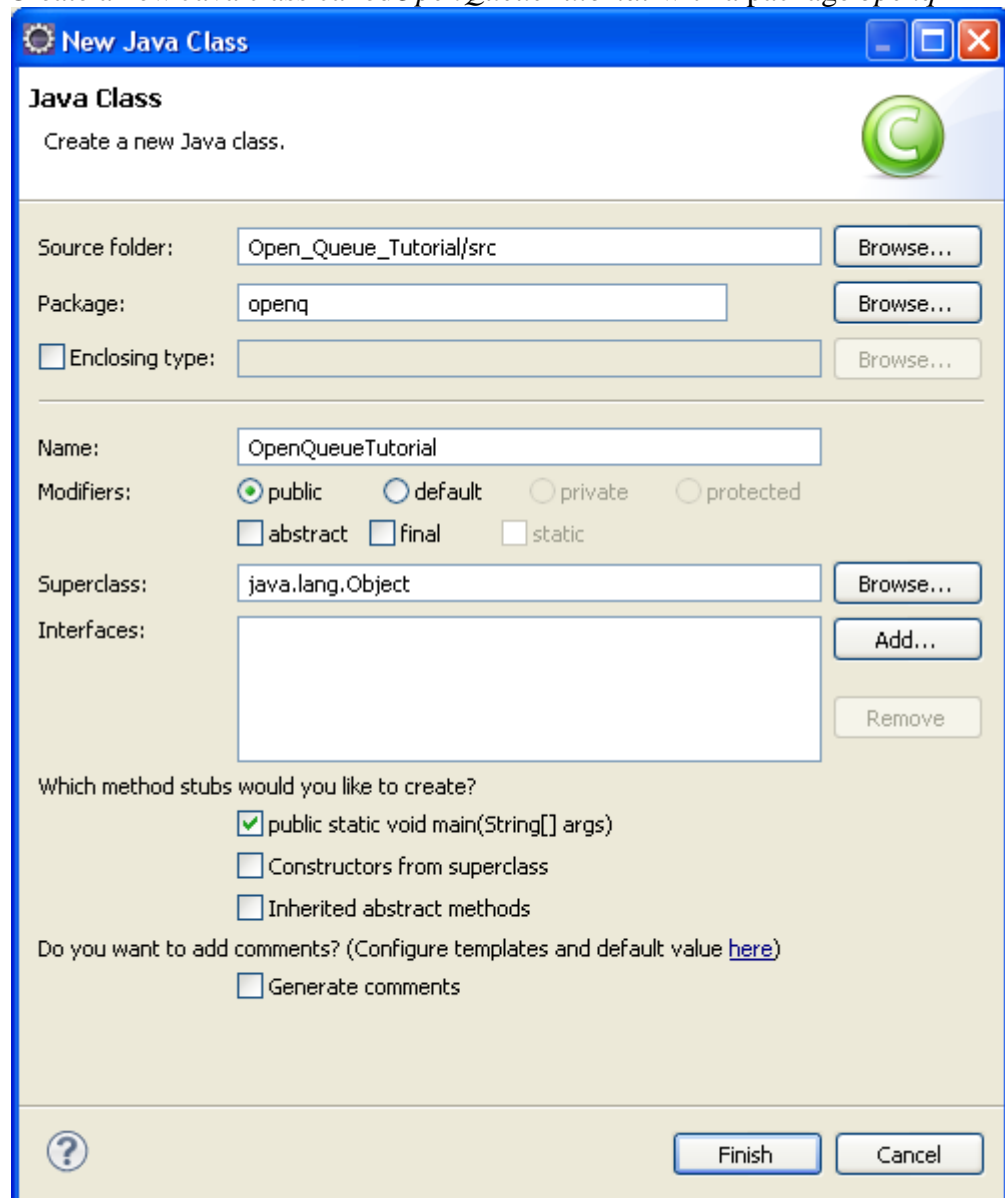
- Create the Service – configure the proxies to point to the CCMS server hosting the service.
- Login – Authenticate the application with the Web Service to retrieve an *sso (single sign on) token*.
- Add the Contact – call the method to add a new contact
- Logout – logout of the service to conserve resources.

In this project we have added an additional step to process the *csv* file.

Step 1 - Create the Class

In this step we are going to create a new main class and authenticate ourselves with the user to receive and *sso* token

1. Create a new Java class called *OpenQueueTutorial* with a package *openq*



Step 2 - Create the Service

Before we can call the methods on the service we must first point our application at the WSDL location. To do this we must supply the hostname of the CCMS server and the port on which the service is located, default is *9080*.

1. Create a global variable with the CCMS host name, web service port and a variable for the service.

```
static String host = "localhost:9080";  
static OpenQ service = null;
```

2. Create a new method called *createService*.

```
public static void createService() throws  
MalformedURLException, GetVersionFault{  
  
    String serviceUrl = "http://" + host +  
"/SOAUI/services/OpenQ?wsdl";  
    URL url = new URL(serviceUrl);  
    service = new SOAUIOpenQ(url).getOpenQ();  
  
    System.out.println("createService(): service[" + (service  
== null ? "null": service.getVersion(new GetVersionRequest())) + "]);  
}
```

3. When this method is called successfully it will create a new service point to the configured URL

Step 3 - Authenticate User

In this step we must authenticate our application with the CCMS service to receive an *sso* token. This token is used in all subsequent method calls to keep track of the application session.

1. Create the following global variables. These variables contain the username and password for the application. The username is always **OpenWsUser** but the password can be changed through the CCMS Server Configuration application

```
static String username = "OpenWsUser";
static String password = "Password123";
static String domain = "open_queue";
static SsoToken sso = null;
```

2. Create a new method called *login*

```
public static void login() {

    try{
        AuthenticationLevel details = new
AuthenticationLevel();
        details.setUsername(username);
        details.setPassword(password);
        details.setDomain(domain);

        sso = service.login(details);

    } catch (LogInFailedFault liex) {
        System.out.println("A login error has
occured. This may indicate another application is currently
logged in.");
        System.out.println("Please try again in a few
seconds.");
        System.out.println("If the problem persists
add the value logout to forcefully logout the existing session
i.e. SalesforceOpenQ '" + strFile + "' '" + host + "'logout");
        System.out.println("error[" +liex + "]");

        System.exit(0);
    }
}
```

3. When this method is successfully called it supplies the username and password to the service and receives an *sso* token in return.

Step 4 – Import the file

In this step we are going to open a text file called *intrinsic.csv*. This text file contains an external Id for the contact that will be created and a series of intrinsic values to be associated with this contact.

An intrinsic is a key/value pair. These intrinsic values offer application developers a way to associate business information with a contact. This information can be used in routing decisions or viewed by the agent when the contact is answered. Each key must have its own unique name and can have any text value associated with it.

The following sample file will create two contacts with external Ids 1 and 2. Each of these contacts will contain two intrinsic values called *intrinsic_name1* and *intrinsic_name2*

```
id,intrinsic_name1, intrinsic_name2
1,intrinsic_value1, intrinsic_value2
2,intrinsic_value1, intrinsic_value2
```

1. In the main method add code to open the file and process each of the rows in the *intrinsic.csv* file.

```
        BufferedReader br = new BufferedReader( new
FileReader("intrinsic.csv"));

        String strLine = "";

        StringTokenizer st = null;
        IntrinsicArray intrinsicArray = new IntrinsicArray();
        List<Intrinsic> intrinsics = intrinsicArray.getItem();

        int lineNumber = 0, tokenNumber = 0;
        String contactId = null;
        Contact contact = null;

        //read comma separated file line by line

        while( (strLine = br.readLine()) != null) {

            lineNumber++;

            //break comma separated line using ","

            st = new StringTokenizer(strLine, ",");

            while(st.hasMoreTokens()){

                tokenNumber++;
                intrinsics.clear();

                // Process Header
                if(lineNumber == 1){
                    headerMap.put(tokenNumber,
st.nextToken());
                }else{
                    //display csv values
```

```
        if(tokenNumber == 1){
            contactId = st.nextToken();
        }else{
            Intrinsic intrinsic = new
Intrinsic();
intrinsic.setKey((String)headerMap.get(tokenNumber));
intrinsic.setValue(st.nextToken());
                                intrinsic.setImmutable(true);
                                intrinsics.add(intrinsic);
                                }
                                }
    } // end while

    //reset token number
    tokenNumber = 0;

}
```

Step 5 – Add the Contact

In this step we will add a new Open Queue contact for each *external Id* present in the *intrinsic.csv* file. If a contact with a matching *external Id* exists in the Contact Center then an exception will be thrown, otherwise a new contact will be created.

1. Create a new method called *addContact*. The parameter *OutOfProviderAddressName* indicates the source of where this contact came from, similar to a calling address, this value is set to "SourceName" here.

```
public static Contact addContact(String id, IntrinsicArray
intrinsic) throws CreateOQContactFailedFault{

    return service.createOQContact(id, "SourceName",
intrinsic, sso);

}
```

2. When this method is successfully called it will return the newly created contact.

Step 6 – Logout

Only one active session is allowed to utilize the Open Queue web service so it is important to logout when the application is complete to free up resources. Applications can logout by either supplying the *sso* token or by supplying the login in credentials. In this step we are going to logout using login credentials to allow us to forcibly remove any older sessions.

1. Create a new method *logout*.

```
public static void logout() {  
  
    try {  
        AuthenticationLevel details = new  
AuthenticationLevel();  
        details.setUsername(username);  
        details.setPassword(password);  
        details.setDomain(domain);  
  
        com.nortel.soa.oi.openq.types.LogOffSessionRequestType request  
= new com.nortel.soa.oi.openq.types.LogOffSessionRequestType();  
        request.setAuthenticationLevel(details);  
        // log off session  
  
        service.logOffSession(request);  
    } catch (LogOffSessionFailedFault e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
  
}
```

2. When this method is successfully called the *sso* token will be invalidated. If the application fails to logout the session will timeout after a specified period, the default is 2 hours.

Step 7 – Putting it all together.

So now we have created all the methods we need to put it all together.

1. The following is the complete listing for the main method

```
public static void main(String[] args) {  
    try{  
  
        if(args == null || args.length ==0){  
            System.out.println("default args[1] - host["  
+ host +"]");  
            System.out.println("OpenQueue Web Service  
url[http://" + host + "]/SOA/Services/OpenQ?wsdl");  
        }else{  
            if(args.length >=1){  
                host = args[1];  
            }  
  
        }  
  
        // STEP 2 - Create the Service  
        createService();  
    }  
}
```

```

// STEP 3 - Authenticate User
login();

// STEP 4 - Import the file
HashMap headerMap = new HashMap();

//create BufferedReader to read csv file
BufferedReader br = new BufferedReader( new
FileReader("intrinsic.csv"));

String strLine = "";
StringTokenizer st = null;
IntrinsicArray intrinsicArray = new
IntrinsicArray();
List<Intrinsic> intrinsicList = intrinsicArray.getItem();

int lineNumber = 0, tokenNumber = 0;
String contactId = null;
Contact contact = null;

//read comma separated file line by line
while( (strLine = br.readLine()) != null) {

    lineNumber++;

    //break comma separated line using ","
    st = new StringTokenizer(strLine, ",");

    while(st.hasMoreTokens()){

        tokenNumber++;
        intrinsicList.clear();

        // Process Header
        if(lineNumber == 1){
            headerMap.put(tokenNumber,
st.nextToken());
        }else{
            //display csv values

            if(tokenNumber == 1){
                contactId =

            }else{
                Intrinsic intrinsic =
new Intrinsic();

                intrinsic.setKey((String)headerMap.get(tokenNumber));

                intrinsic.setValue(st.nextToken());

                intrinsic.setImmutable(true);

                intrinsicList.add(intrinsic);

```

```

    }

    }

    } // end while

    // STEP 5 - Add the Contact
    try{
        // ignore the header
        if(lineNumber > 1){
            contact =
addContact(contactId, intrinsicArray);
        }

        } catch (Exception ex) {
            System.out.println("Error: Unable
to Create contact with externalId[" + contactId + "], error[" +
ex.getMessage() + ""]);
        }

        if(contact != null){
            System.out.println("created contact
id[" + contact.getContactId()+ "], externalId[" +
contact.getExternalContactId() + "].");
        }
        //reset token number
        tokenNumber = 0;

    }

}

}

catch (Exception ex) {
    ex.printStackTrace();
} finally{

    // STEP 6 - Logout
    logout();
}

}

```

2. On successful completion of the method the following output should be seen on the console

```

default args[1] - host[localhost:9080]
OpenQueue Web Service url[http://localhost:9080]/SOA0I/services/OpenQ?wsdl
createService(): service[com.nortel.soa.oi.cct.types.GetVersionResponse@17b4703]
created contact id[4bf8f650-75ef-4fd1-891a-c9f29c2ed0fa], externalId[1].
created contact id[fe71bd4a-ad94-4691-a85d-f9ae3f0a0420], externalId[2].

```

3. Once the contact is created the contact will process it like any other contact and ultimately should forward it on to any agent with the OpenQ contact type.