Getting Started with the Bullhorn SOAP API and Java

Introduction

This article is targeted at developers who want to do custom development using the Bullhorn SOAP API and Java. You will create a sample application that creates a session to the Bullhorn system, retrieves candidates using a couple of methods, and displays the results on the console.

You will learn how to:

- set up your environment for development.
- access the Bullhorn SOAP API.
- create and publish a web application in Eclipse that retrieves candidate data.

You can follow the step-by-step instructions and/or download and run the provided code sample files.

Prerequisites

This article assumes knowledge of Java, Eclipse and object-oriented programming. You will also need the Eclipse AXIS plug-in.

Code files

There are starter and solution files provided for this tutorial as a download. The starter zip file needs to be extracted within your Java project before you start writing the code. The starter zip has all the libs and the build files needed for the project.

The solution zip file can be downloaded and run if you want to skip the steps and just see the completed application. Remember to add the authentication information and your API key to the solution file before running it.

GettingStartedWithWebServices_Java_Starter.zip

GettingStartedWithWebServices_Java_Solution.zip

1. Select **File->Import->General->Existing Projects into Workspace**.
2. Unzip the GettingStartedWithWebServices_Java_Solution.zip under the **Projects** directory in your Eclipse workspace.
3. Build and run the completed application.

Getting started

In order to develop against the Bullhorn web services API, you will need a **username**, **password** and **API key**. If you don't want to use your own, Bullhorn support can provide you with an API user account.

Note: Developers working directly for Bullhorn customers can get API access by having the customer contact Bullhorn Support. Once the APIs are enabled for a Bullhorn client, a client administrator can generate a customer-specific API key by going to **Tools > BH Connect > Web Services API**. If a key does not already exist, you can create a new one by clicking **Add Account**.

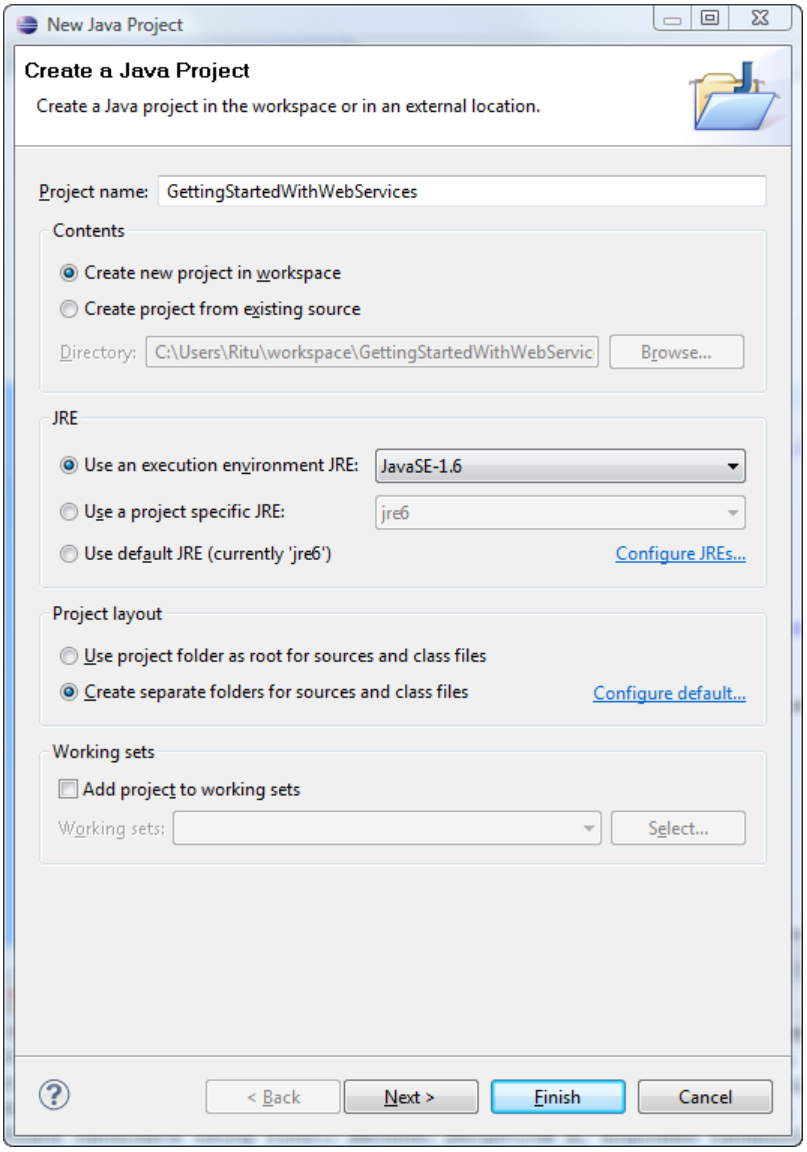
Setting up Eclipse

The following steps explain how to create and set up a project in Eclipse that connects to the Bullhorn WSDL.

Creating a project

In this section, you will create a Java project in Eclipse.

4. Launch Eclipse and select **File > New > Java Project** to create a new project.
5. Enter **GettingStartedWithWebServices** under **Project name**.



6. Select **Finish** to create the project.

Adding a web reference to the Bullhorn WSDL

Before using the API, you must first generate Java objects and classes that serve as proxies for their server-side counterparts from the Bullhorn's WSDL file. You can use the WSDL2Java utility to create these client stubs.

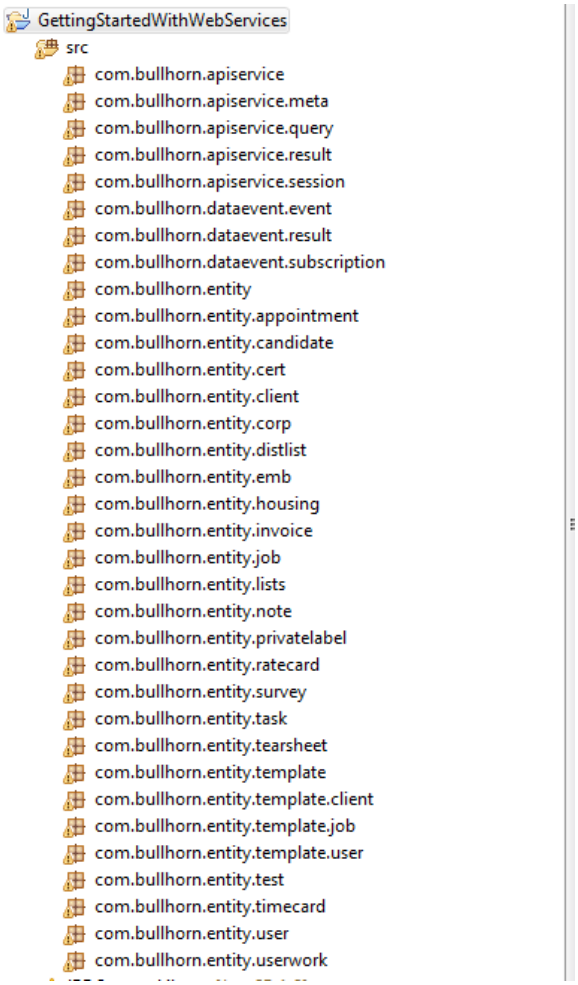
Note: You will find the Axis WSDL-to-Java tool in "org.apache.axis.wsdl.WSDL2Java".

Student files are provided with the tutorial that has the ANT build files created for you already. Download the GettingStartedWithWebServices_Java_Starter.zip file and save it to your local file system.

Using the WSDL2Java utility

You will run the WSDL2Java tool to create Java proxy classes for the Bullhorn SOAP API. The client application that you are building will use these classes to interface with the Bullhorn system.

7. In Eclipse, select the new project you just created.
8. Select **File > Import > General > Archive File**.
9. Click **Next**.
10. Browse to the student file **GettingStartedWithWebServices_Java_Starter.zip** on your local file system.
11. Make sure the **Into folder** points to the **GettingStartedWithWebServices** project folder.
12. Click **Finish**.
    You will see the following new files and folders created under your project.
    - lib – This folder contains the jar files for web services
    - build.xml - This file has the build definition in it
    - build.properties – This file defines the properties for the build.xml file
13. Open a command prompt window and browse to the **GettingStartedWithWebServices** project folder you created within Eclipse.
14. Run **ant wsdl2java** in the folder.
    You will see all the Java proxy classes that interface between your application and the web service generated under the **src** folder.
    You can now explore all the classes and look at methods within them.



Now that the web reference is associated with the project, you can start making calls to the API within your Java code.

Understanding how the Bullhorn SOAP API works

This section walks through a sample Java client application. The purpose of this sample application is to show you the required steps for creating a session between the client application and the Bullhorn system, and to demonstrate the invocation and subsequent handling of some API calls.

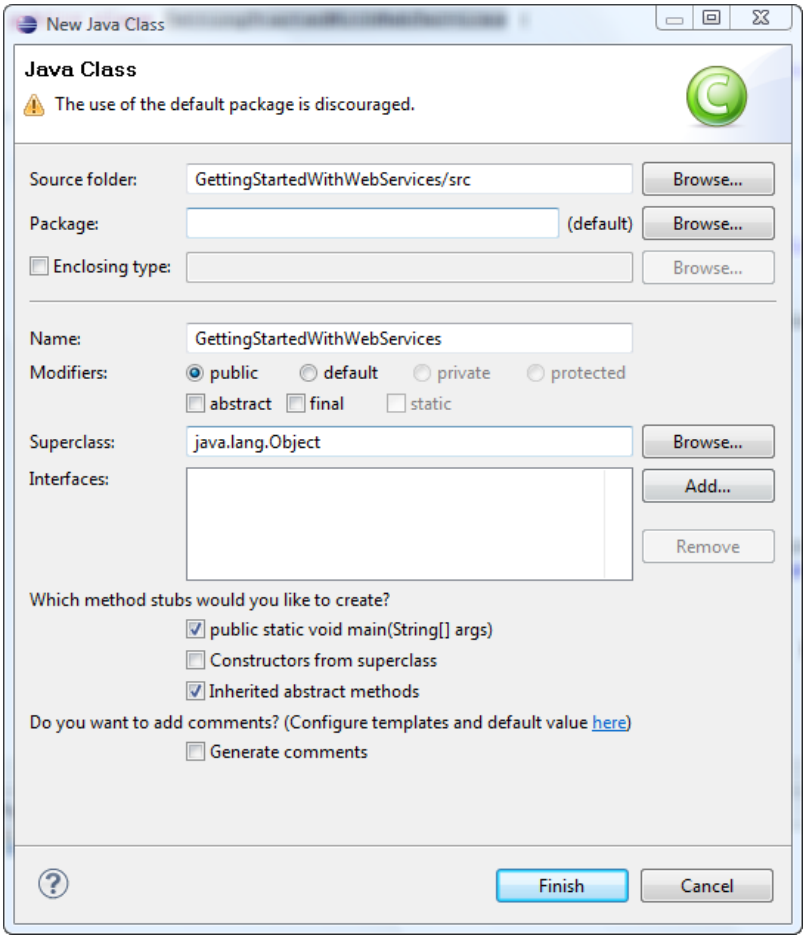
You will program the sample application to perform the following tasks:

- Create a session to the Bullhorn web service using your credentials.
- Build a basic query using dtoQuery.
- Retrieve a single instance of the Candidate DTO.
- Retrieve multiple instances of Candidate DTO.
- Publish the results on the console.

Creating a Java class

In this section, you will create the basic skeleton of the Java class using a wizard. This class is the only Java class that you will need to do all the coding for this application.

15. In Eclipse, select the project folder and choose **File > New > Class**.
16. Name the class **GettingStartedWithWebServices**.
17. Check the method stub for **public static void main(String[] args)**.
18. Click **Finish**.



Creating a session

All calls to the Bullhorn SOAP API require you to pass in a valid session. In order to create a session to the Bullhorn system within the web application, you need a username, password and API key. In this section, you will define the authentication mechanism for access and create a session to the Bullhorn service.

19. In GettingStartedWithWebServices.java file, define three string variables in the main method. The values for these variables should be given to you by support.

```
String username =  "yourusername";
  String password =  "yourpassword";

  String apiKey =  "yourapikey";
```

22. Define one more string variable for the WSDL URL as shown below.

```
String wsdlUrl =  "https://api.bullhornstaffing.com/webservices-1.1/?wsdl";
```

Note: At the time of publication, the latest version of the Bullhorn SOAP API is version 1.1.

23. Define a QName variable outside the main method as shown below.

```
private static final QName SERVICE_NAME = new

QName("http://apiservice.bullhorn.com/", "ApiService");
```

25. Instantiate the ApiService class by adding the following code.

```
URL serviceUrl = new
  URL(ApiService_Service.class.getResource("."), wsdlUrl);

  ApiService service =  new

  ApiService_Service(serviceUrl, SERVICE_NAME).getApiServicePort();
```

30. Start an API session by invoking the `startSession()` method in the `ApiService` class. The method startSession() accepts `username`, `password` and `apiKey` as arguments. The method returns a session object in the response. Assign the session object to the `currentSession` variable you created.

```
ApiSession currentSession = service.startSession(username, password,  apiKey);
```

Note: Sessions have a short expiration and need to be refreshed with each call to the Bullhorn SOAP API. Each call to the Bullhorn SOAP API requires a valid session object, and each response returns a new session object. You should store that new token in the current session variable so you always have a recently refreshed value.

The session object returned as part of the response should be used in the next call that is made to the API, as previous session objects will expire in five (5) minutes.

Building a basic query

Tip: Using the Bullhorn SOAP API Query Operation

One of the most powerful operations in the Bullhorn web services APIs is the query operation. This operation allows you to construct SQL-like queries for a particular type of entity.

The Bullhorn query operation is built on top of **Hibernate**, an object-relational mapping tool that exposes relational data as a series of objects. The Hibernate Query Language (HQL) is based on standard SQL but adapts its concepts to an object-oriented language (Java). The query operation exposes a subset of the operations supported by HQL.

To execute a query, you must construct a Java `query` DTO and then pass it in as an argument when you call the `query`operation. The two most important fields in the query DTO are the `entityName` property, where you specify the name of the entity you are querying for, and the where property, which contains the where clause for your

query. Within the `where` property, you can specify a single parameter or create a more complex query using `AND`, `OR`, or `NOT`.

In this section, you will create a query that returns the Candidate list with at most 10 candidates that are not deleted.

24. Create an instance of `dtoQuery` class. This class is within the package `com.bullhorn.apiservice.query`.

```
DtoQuery candidateQuery = new DtoQuery();
```

25. Next you will specify the `entityName` to be `Candidate`. The `setEntityName()` is a method within the `DtoQuery` class that accepts a string variable for the entity name.

```
candidateQuery.setEntityName("Candidate");
```

26. Define the `maxResults` property to be equal to 10. The `setMaxResults()` is a method within the `DtoQuery` class that accepts an integer for the maximum number of results to be returned in the response.

```
candidateQuery.setMaxResults(10);
```

27. Now we need to construct the `where` clause. The where clause is an SQL-like string that will be executed by the Bullhorn server. For our purposes, the query only checks to ensure that the Candidate has not been deleted, but you can query on any of the properties exposed on the DTO. For a full list of Candidate properties, see the reference documentation.

```
candidateQuery.setWhere("isDeleted  = 0");
```

28. Now that the query has been created, use the `query` method in the `service` class and pass it the `currentSession` and the `candidateQuery` variables as arguments. The method returns a SOAP response that is stored in the `queryResult`variable. The `ApiQueryResult` class is within the `package com.bullhorn.apiservice.result` and it used to store query results.

```
ApiQueryResult queryResult = service.query(currentSession,  candidateQuery);
```

29. Refresh the session by putting the new session in the global variable `currentSession`.

```
currentSession  = queryResult.getSession();
```

Run the debugger to see the `queryResult` variable with the returned data. Notice that the result contains a list of IDs. Each of these IDs corresponds to an instance of a Candidate in Bullhorn. There are several operations in Bullhorn that will return a list of ids, including `getAssociationIds()` and `getEntityNotes()`. When using these operations, you most often will follow up by retrieving the specific instance data using `find()` method or `findMultiple()` method, as below.

Retrieve an instance of a DTO

Once the response from the Bullhorn system is available in the `queryResult` variable, you can extract the ID nodes of candidates and use each ID to retrieve an instance of the Candidate DTO. In this section, you will use the `find()` operation and the `findMultiple()` operation to retrieve a single instance of a DTO or a list of DTOs.

Retrieving a single DTO using the find() method

You can use the `find()` method to retrieve an instance of the Candidate DTO.

30. Check using the `if` condition that the `queryResult` variable returned some id nodes

```
if (!queryResult.getIds().isEmpty())
  {

  }
```

33. Within the `if` loop, create a loop to access individual ID nodes in the `queryResult` variable.

```
for (int i =0; i< queryResult.getIds().size() ; i++)
  {

  }
```

36. Within the loop, you can use the `find()` method in the service class to retrieve an instance of the DTO. The `find()` method takes the `session`, entity name and the ID of the Candidate as arguments and returns the result as a DTO object in the ApiFindResult class.

```
ApiFindResult  candidate = service.find(currentSession, "Candidate",
  queryResult.getIds().get(i));
```

38. Refresh the session by putting the new session in the global variable `currentSession`.

```
currentSession =  candidate.getSession();
```

39. Type cast the `dto` object returned in the result as a `CandidateDto` object.

```
CandidateDto  thisCandidate = (CandidateDto)candidate.getDto();
```

Publishing the results of the find() operation on console

In order to view the results of the data retrieved from a Bullhorn system, you can print the returned data to the console. The candidate object has several methods defined to access the properties of the object.

For the purposes of this tutorial, you can use the `getName()` method to retrieve the candidate's name, the `getOccupation()` method to retrieve the occupation and the `getDateAvailable()` method to retrieve the date the candidate is available to start the new job.

```
System.out.println("Found candidate using find() method: "  +thisCandidate.getName() + ",
  " + thisCandidate.getOccupation() +  ", available on "
+thisCandidate.getDateAvailable());
```

Retrieving a list of DTOs using findMultiple() method

You can also use the `findMultiple()` method to retrieve several instances of the DTO together. If you know you will need to fetch more than 2-3 DTOs, it is more efficient to use `findMultiple()` as it will reduce the number of round trips required to get the data. However, it will also increase the size of the responses you receive.

35. The `findMultiple()` method takes the session, entity name and an array of up to 20 ID nodes as arguments and returns the result as an array of `dto` objects in the `ApiFindMultipleResult` class.

```
ApiFindMultipleResult  candidatesMultiple = service.findMultiple(currentSession,
  "Candidate", queryResult.getIds());
```

37. Refresh the session by putting the new session in the global variable `currentSession`.

```
currentSession =  candidatesMultiple.getSession();
```

38. To access individual DTOs, loop over the result array using the length of the array as a condition.

```
for (int i = 0; i < candidatesMultiple.getDtos().size(); i++ )
  {

  }
```

41. Within the `for` loop, access the individual dto's by type casting each object in the result array to `candidateDto`.

```
CandidateDto  thisCandidate = (CandidateDto)candidatesMultiple.getDtos().get(i);
```

Publishing the results of findMultiple() operation on console

39. Add the following code to print the results of the findMultiple() operation.

```
System.out.println("Found candidate using findMultiple() method:" +
  thisCandidate.getName() +  ", " + thisCandidate.getOccupation() +

  ", available on "  +thisCandidate.getDateAvailable());
```

Tips for using Java with Bullhorn web services

The following tips are useful when working with Java and the Bullhorn WSDL.

Using JDK 1.5 and above

We recommend using JDK 1.5 and above for this tutorial and working with Bullhorn SOAP API.

Using a local copy of the WSDL and building from it

The Bullhorn WSDL might be updated over time. To have complete control over your implementation, we recommend you download a copy of the Bullhown WSDL and use the local version to build your application by following the steps below:

40. Download the Bullhorn WSDL from **https://api.bullhornstaffing.com/webservices-1.1/?wsdl**.
41. Save the file as bullhorn.wsdl under the GettingStartedWithWebServices project workspace.
42. Edit the build.properties file to point the `wsdl.url` to the local file bullhorn.wsdl.
43. Run `ant WSDL2Java` again to generate Java classes from the local version of the WSDL.