White Paper - How to call Polarion from DOT-NET

Table of Contents

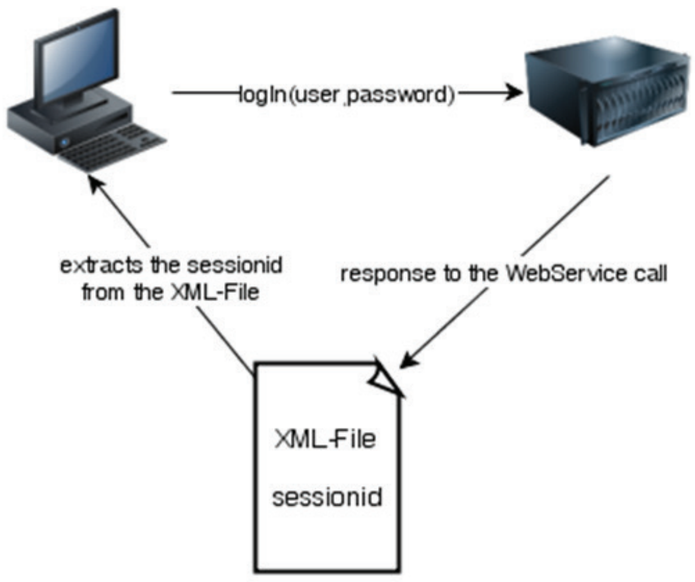
Contents

1 Introduction

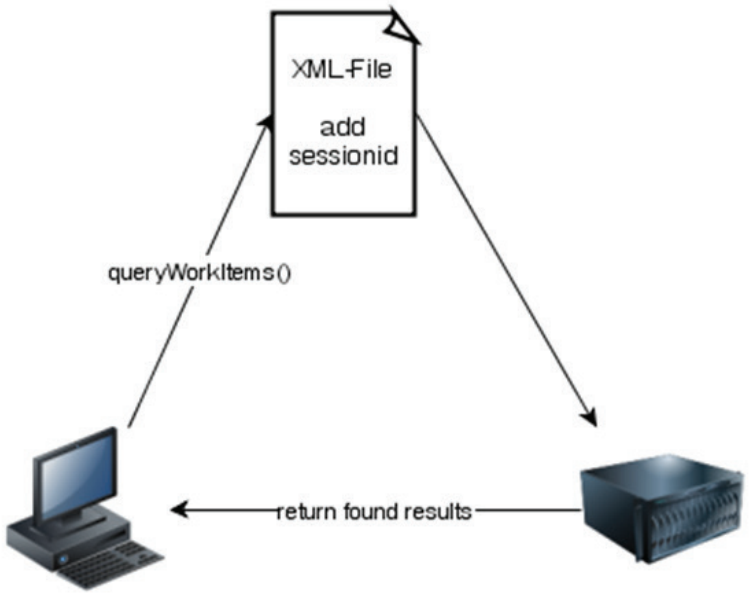
This white paper describes how to use a .NET application to connect to the Polarion WebService API as a WebService client. Examples are implemented in C#, but any .NET programming language may be used.

2 Polarion WebService API Methodology

The WebService Client connects to the Polarion WebServer by authenticating itself with a login() method call. When authentication is successful the WebServer returns the value, True, and establishes a session between the client and the Polarion server. The Polarion server will also provide a session identifier, which is not part of the return value, but is contained in the returned XML-file.


logIn(user,password)
extracts the sessionid from the XML-File
response to the WebService call
XML-File
sessionid

The sessionid will be extracted from the XML-File and stored in the connection to the Polarion server. Every time a WebService method is called the sessionid has to be included in the transmitted XML-File. Since this is done via method calls that Visual Studio is providing from wsdl files, an additional programming extension is needed to transparently retrieve the sessionid from the XML response file.



If the sessionid is not send with the XML-File, NotAuthenticatedException is returned. The sessionid has a lifetime of 10 minutes so if the client doesn't call any method within this time frame the session is closed.

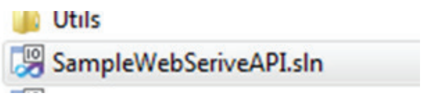
3 Visual Studio Integration

The .NET approach below may be used with MS Visual Studio 2010 or higher. The programming language is C#.

3.1 Before you start

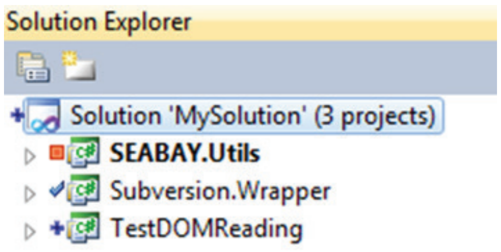
There two options to adapt the .NET integration software:

1. **Double click on the solution file to open an MSVS instance and read the solution.**

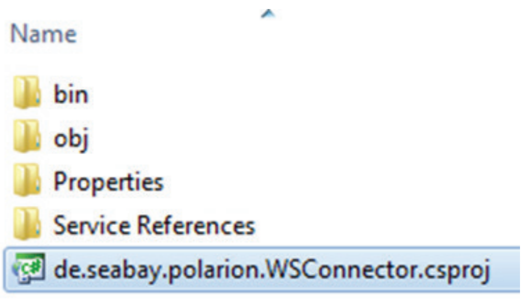


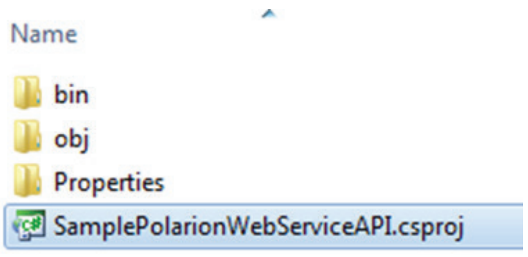
Note: If you run an MSVS version higher than 2010, you may be asked to convert the solution and the projects to your current version.

2. **Import the project into an existing solution**
If you already have a solution (e.g. MySolution.sln), right click on your solution (Add -> Existing Project ...),

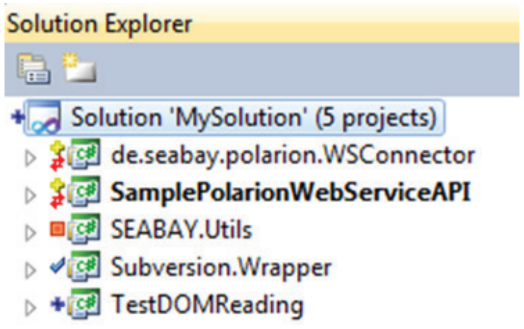


... and import the projects contained in the sub directories into the solution.





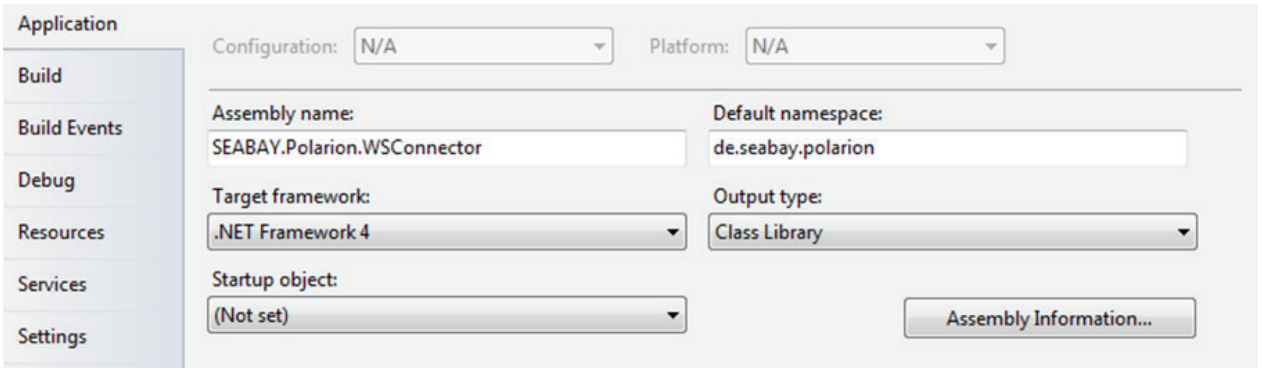
Once you have imported the projects into your solution and set the SampleWebServiceAPI as the startup project, your Solution Explorer should look like this:


Solution Explorer
Solution 'MySolution' (5 projects)
de.seabay.polarion.WSConnector
SamplePolarionWebServiceAPI
SEABAY.Utils
Subversion.Wrapper
TestDOMReading

3.2 The .csproj Project files

This section describes in detail the content of two files:

1. **de.seabay.polarion.WSConnector**
   This is the Connector to the Polarion WebService API and may also be used as a referenced assembly after adjusting it to your environment. The output type is Class Library and it is a .NET Framework 4 compilation.



Configuration: N/A
Platform: N/A
Assembly name: SEABAY.Polarion.WSConnector
Default namespace: de.seabay.polarion
Target framework: .NET Framework 4
Output type: Class Library
Startup object: (Not set)

2. <u>SampleWebServiceAPI</u>
   This Console Application is a sample of a Polarion WebService Application. The following section details how the Console Application:
   - Retrieves WorkItems
   - Changes data and stores the data changes
   - Retrieves information from a given TestRun

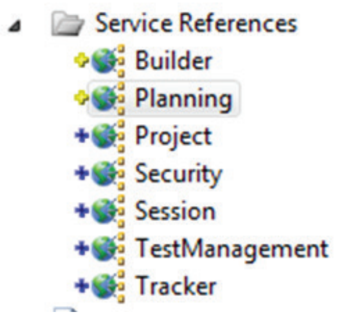
Note: The sample application uses the de.seabay.polarion.WSConnector. Without the usage of this Connector/Assembly a connection cannot be established.

- BuilderWebService
- PlanningWebService
- ProjectWebService
- SecurtityWebService
- SessionWebService
- TestManagementWebService
- TrackerWebService

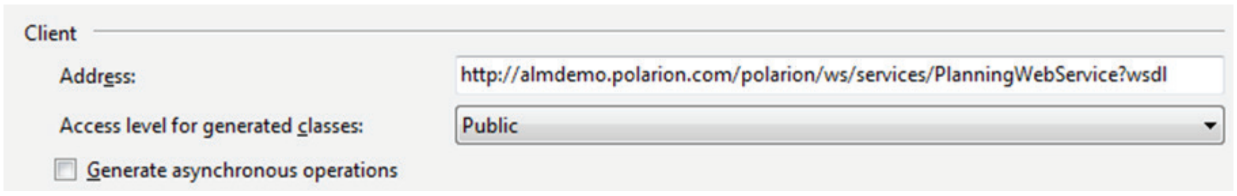
Specifics of each service are documented in the Polarion JavDoc API at http://almdemo.polarion.com/polarion/sdk/doc/javadoc/index.html.

3.2.1.1 Adjusting the Connector to your environment

Update the WebService API to connect to your Polarion Server and version of Polarion in the ServiceReferences menu of the connector project:



Right-click on each service reference and adjust the server settings to map to your environment. In our example we reference alm-demo server, but you will, of course, substitute the server names in your environment.


Client

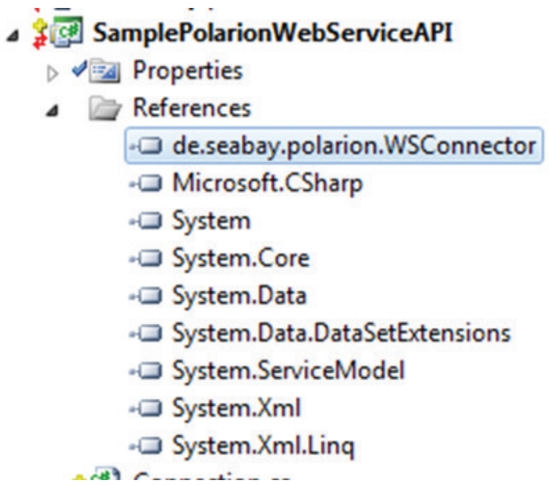
Address: http://almdemo.polarion.com/polarion/ws/services/PlanningWebService?wsdl

Access level for generated classes: Public

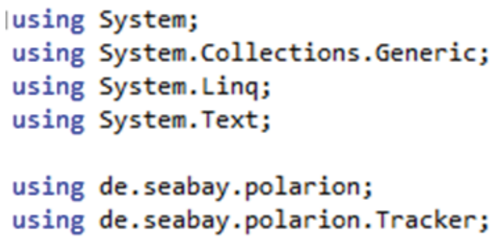
☐ Generate asynchronous operations

3.2.1.2 How to use it in your environment

To use the Connector in your environment, add it as a referenced Assembly. Right-click on the References sub-menu, and add it either as a project reference or a .dll file.

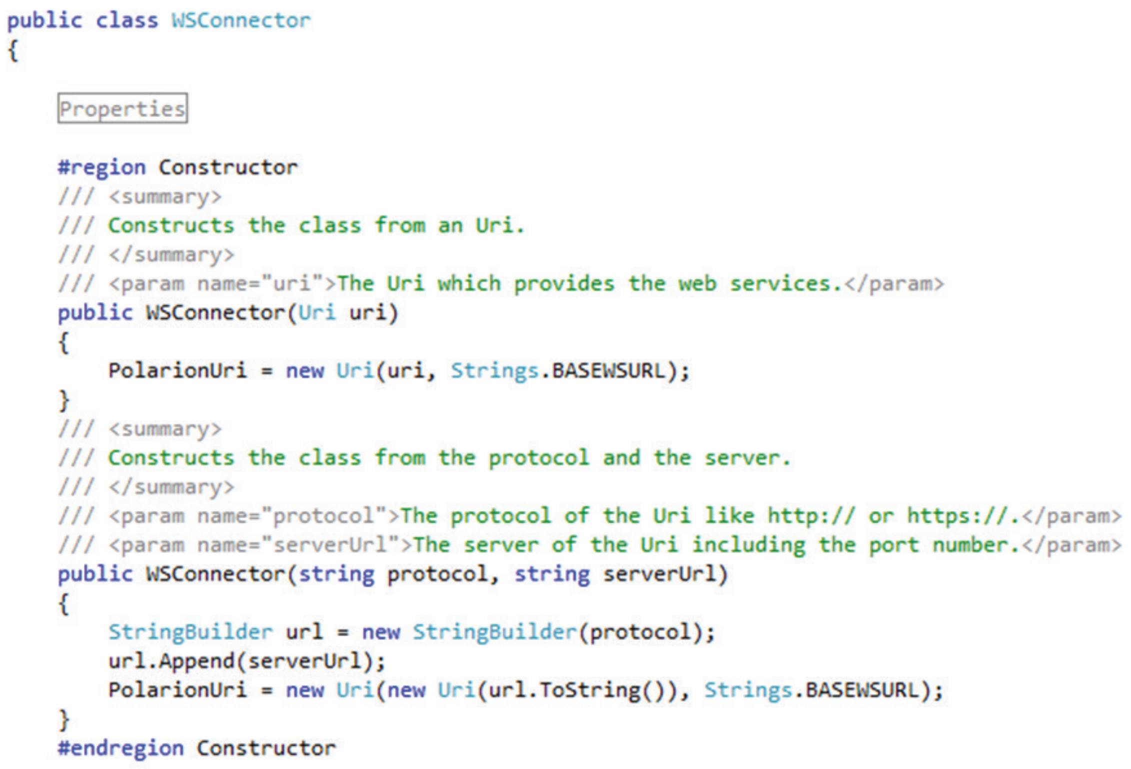

```
SamplePolarionWebServiceAPI
  Properties
  References
    de.seabay.polarion.WSConnector
    Microsoft.CSharp
    System
    System.Core
    System.Data
    System.Data.DataSetExtensions
    System.ServiceModel
    System.Xml
    System.Xml.Linq
```

Add namespace de.seabay.polarion for each WebService you would like to use:


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using de.seabay.polarion;
using de.seabay.polarion.Tracker;
```

To use the capabilities of the Connector you need to instantiate a new WSConnector class the assembly will provide to Constructors. The first one accepts a Uri and the second one accepts a protocol and a serverUrl.


```
public class WSConnector
{

    Properties

    #region Constructor
    /// <summary>
    /// Constructs the class from an Uri.
    /// </summary>
    /// <param name="uri">The Uri which provides the web services.</param>
    public WSConnector(Uri uri)
    {
        PolarionUri = new Uri(uri, Strings.BASEWSURL);
    }
    /// <summary>
    /// Constructs the class from the protocol and the server.
    /// </summary>
    /// <param name="protocol">The protocol of the Uri like http:// or https://.</param>
    /// <param name="serverUrl">The server of the Uri including the port number.</param>
    public WSConnector(string protocol, string serverUrl)
    {
        StringBuilder url = new StringBuilder(protocol);
        url.Append(serverUrl);
        PolarionUri = new Uri(new Uri(url.ToString()), Strings.BASEWSURL);
    }
    #endregion Constructor
```

Start a new instance with either Constructor, and call the connect method:

```
WSConnector Factory = new WSConnector("http://", "almdemo.polarion.com");
Factory.Connect();
```

If the WSConnect does not connect to all WebServices, use the provided classes. Otherwise an exception will be returned and the connection to the Polarion WebService will not be established.

Each WebService connection is stored in the WSConnector and is provided from the WSConnector as a list of WebServices. You can use the connection directly by obtaining the object reference from the list, assuming the variable Factory contains the class reference to the WSConnector (see above):

```
public TrackerWebServiceClient Tracker { get { return Factory.WebServices[WebServiseFactory.Tracker]; } }
```

If you want to call the queryWorkItems method from the TrackerService, take the Tracker variable, add the method call, and provie the parameter.

```
WorkItem[] wiList = Tracker.queryWorkItems("type:requirement", "id", new string[] { "id", "title" });
```

3.2.1.3 Behind the scenes

The .NET Framework provides classes that can be overwritten in order to manipulate the XML-Files received from the WebServer and sent to the WebServer.

The class SessionIdBehavior registers a SessinIdInspector. The class is implemented in the file SessionIdInspector.cs. This class does the work and overwrites the following methods:

- **AfterReceiveReply:** This method is called after the XML-File is retrieved from the WebService and before the method returns. It looks for an attribute sessionID in the XML-File and detects where the sessionid is stored as a private property within the class.
- **BeforeSendRequest:** This method works in the opposite direction when the client sends a request to the WebServer. Here, previously stored sessionid is added to the attributes of the XML-File after the client calls the method and before the XML-File is send to the WebServer.

4 Sample use cases

This section describes some WSConnector sample use cases. The Console Application contains all sample use cases. Each use case can be called with a different parameter call to the exe file:

- **-r query**
  Retrieving WorkItems
- **-c**
- **-t testRunidunid**
  Retrieving information from TestRuns

Each sample use case is implemented in an own method. Before each of the methods can be called a connection to the WebService has to be established, as previously described. SampleWebServiceAPI encapsulates methods in the connection class in C# file connection.cs, and takes care of all the house keeping and provides the WebServices as properties.

4.1 Connecting with the Connection class

You should connect with the Connection class by instantiating it, and providing the protocol and the server. Here we are using the Polarion almdemo server available at almdemo.polarion.com. Also for this documentation we created an account with user name and password of seabay.

```
static void Main(string[] args)
{
    Connection con = new Connection("http://", "almdemo.polarion.com");

    con.login("seabay", "seabay");

    if (con.IsLoggedIn == true)
    {
```

Once the connection is established a case can be called.

4.2 Retrieving WorkItems

This use case is implemented in the method GetWorkItems.
It prints all WorkItems of the provided query on the console and waits for the user to enter any key. It can easily change to also support sql queries.

```
static void GetWorkItems(string query)
{
    WorkItem[] wiList = con.Tracker.queryWorkItems(query, null, new string[] { "id", "title" });

    int index= 1;
    foreach (WorkItem wi in wiList)
    {
        Console.WriteLine("({0}) = {1} - {2}", index++, wi.id, wi.title);
    }

    Console.ReadKey();
}
```

4.3 Create a new WorkItem and change data

This use case is implemented in the method ChangeWorkItems.
It shows how WebServices can be used to create a new workitem and perform an action on that workitem. When a new workitem is created the projectid and the type of the workitem must be provided. The title should also be set.

This use case shows how a custom field with a multi selectable enumeration can be set. In this case, a new instance of the WorkItem class will be filled with the data before calling the createWorkItem of the TrackerWebService.

```
static void ChangeWorkItems(string workItemType)
{
    //  create a new WorkItem instance
    TrackerService.WorkItem newWorkItem = new TrackerService.WorkItem();

    //  set project
    newWorkItem.project = new TrackerService.Project();
    newWorkItem.project.uri = con.Project.getProject("drivepilot").uri;

    //  set the workitem type
    TrackerService.EnumOptionId enumId = new TrackerService.EnumOptionId();
    enumId.id = workItemType;
    newWorkItem.type = enumId;

    //  set the title
    newWorkItem.title = "Created By a WebService API-Call";
```

After the IWorkItem class is instantiated and filled – at minimum with the type and project – the new workitem can be created.

```
//  create the workitem
string newWorkItemUri= con.Tracker.createWorkItem(newWorkItem);
```

Following the call, the workitem can be seen in Polarion.

The next use case involves filling in the custom field targetVersion with the value Version 1.0. In this case, the API provides a class EnumOptionId that should be filled with the following data:

- **key:** The key property is the id of the custom field. In this case it is targetVersion.
- **parentItemURI:** This property should be filled in with the Uri of the workitem. It identifies the work item where the custom field will be stored.
- **value:** The value must be the same datatype as declared in the custom fields definition. The targetVersion a multi-selectable enumeration. In this case the value is an array of EnumOptionIds.

The first the step is to fill the EnumOptionId and assign as an array the value:

```
//  set the value, it is of type EnumOptionId[]
enumId = new TrackerService.EnumOptionId();
enumId.id= "Version_1_0";
cf.value= new TrackerService.EnumOptionId[] {enumId};
con.Tracker.setCustomField(cf);
```

After calling the setCustomField method from the TrackerWebService, the workitem will contain the value of the custom field.

The last step involves performing an action on the workitem.

```
//  change the workitem status
foreach (TrackerService.WorkflowAction workflowAction in con.Tracker.getAvailableActions(newWorkItemUri))
{
    if (workflowAction.nativeActionId == "reviewed")
    {
        con.Tracker.performWorkflowAction(newWorkItemUri, workflowAction.actionId);
        break;
    }
}
```

4.4 Retrieving information from TestRuns

There are two options for retrieving information from Testruns, search by the Testrun, or search by the id. The application will located the TestrunId from the parameter list and the ITestRun object with the following statement:

```
TestManagementService.TestRun testRun= con.TestManagement.getTestRunById("drivepilot", testRunId);
```

The code iterates over the Testrecords and checks if any test-case executions failed. In this case of failed executions, the Testcase and Defect Ids and Titles will be displayed on the Console:

```
foreach (TestManagementService.TestRecord testRecord in testRun.records)
{
    if (testRecord.result.id == "failed")
    {
        TrackerService.WorkItem testcase = con.Tracker.getWorkItemByUri(testRecord.testCaseURI);
        TrackerService.WorkItem defect = con.Tracker.getWorkItemByUri(testRecord.defectURI);
        Console.WriteLine("failed Testcase: {0} - {1}, defect: {2} - {3}",
                          testcase.id, testcase.title, defect.id, defect.title);
    }
}
```

5 Additional information

After reviewing this document, you understand how you can connect Polarion using DOT-NET protocols. For more information about Polarion or specific Polarion solutions, visit www.polarion.com, or contact us at info@polarion.com.