



---

**Q-Pulse Web Service APIs**  
**Developing an Outlook 2007 Plugin**

# Contents

---

<b>Introduction.....</b>	<b>3</b>
<b>Testing the Web Services Installation .....</b>	<b>4</b>
<b>Initial overview of the desired functionality.....</b>	<b>5</b>
<b>Getting Started.....</b>	<b>6</b>
<b>Customising the Outlook Mail Read Ribbon.....</b>	<b>7</b>
<b>Running the Add-In .....</b>	<b>10</b>
Generating the Service Proxies .....	10
Q-Pulse Web Service Operations.....	13
Implementing the rest of the Ribbon Control .....	18
Implementing 'GetOccurrenceImage', 'PopulateMenu' and 'GetEnabled' .....	18
<b>The Final Result .....</b>	<b>21</b>
<b>Limitations .....</b>	<b>22</b>
<b>Useful Links .....</b>	<b>22</b>

## Introduction

The purpose of this tutorial is to demonstrate the use of the Q-Pulse Occurrence Web Service API by implementing an Outlook 2007 Add-in which allows the user to raise an Occurrence in Q-Pulse from an email in Microsoft Outlook.

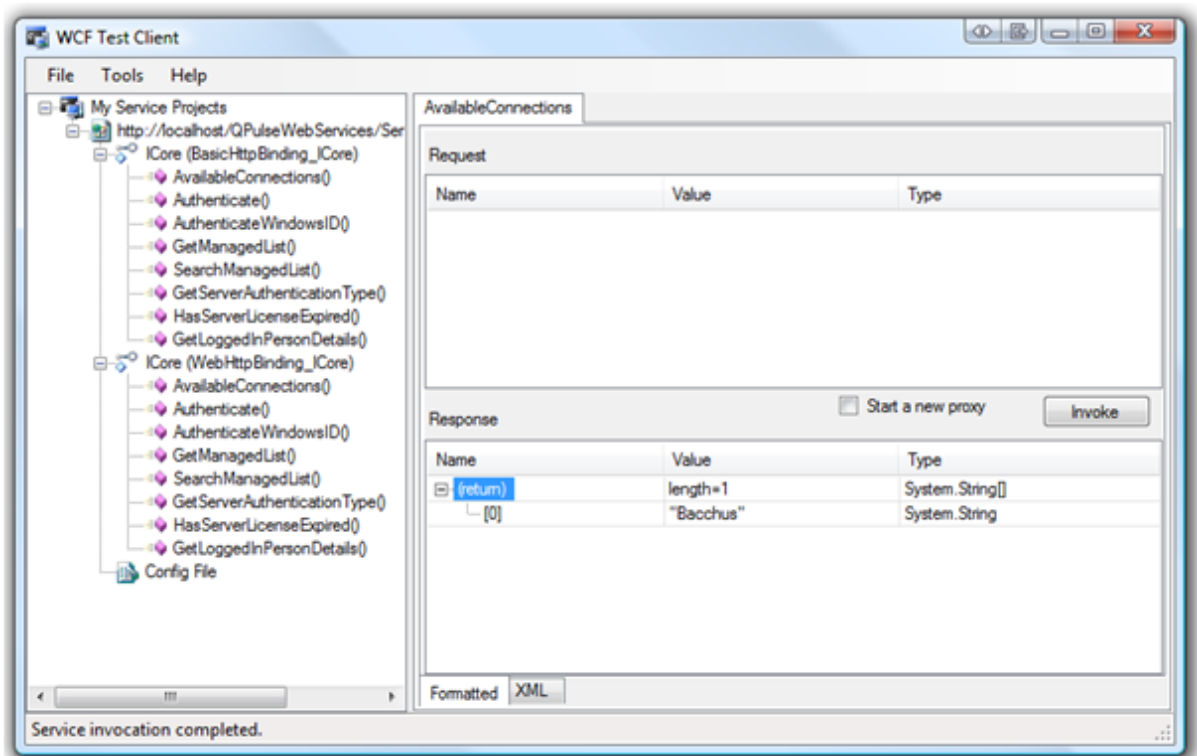
For this tutorial, you will need:

- Visual Studio 2008 Full Version
- Fully configured installation of Q-Pulse 5.7 with Web Services Extension Pack
- A valid Q-Pulse login with full access to the Occurrence module
- Occurrence Report Types defined in the Q-Pulse database
- Tutorial source-code files

In the example code, some of the comments, declaration of class level fields and exception handling have been removed to help improve readability. Please refer to the tutorial source-code files for the more complete version.

## Testing the Web Services Installation

In order to develop against the Web Service, we must first ensure it has been installed and configured correctly. Launch an instance of the Visual Studio 2008 Command Prompt and run the WcfTest utility by typing in 'wcfTestclient' and pressing return. Right-click on "My Service Projects", choose "Add Service" and type in the location of the Core Web Service. It should be accessible from the machine where Web Services was installed and the URL should be of the form: **http://machineName/qpulse5webservices/services/core.svc**. If the test utility is unable to load the schema from the service endpoint, contact your system administrator to help diagnose the problem. If the connection to the endpoint was successful, you should see the side panel populated with service methods that are available to invoke. Double-click on the "AvailableConnections()" method under the "ICore (BasicHttpBinding\_ICore)" section (a WebHttpBinding endpoint is also visible due to the Q-Pulse Web Services offering backwards compatibility with ASP.NET Web Services, however, this binding will not work in the WcfClient). Click on the Invoke button. In the Response window you should see a list of available databases returned from the Web Service. If not, there must be a problem with your installation or configuration – please contact your system administrator.



## Initial overview of the desired functionality

From the high-level user's perspective, we would like them to be able to open an email and have a custom control displayed on the Ribbon UI. Using the control, they should be able to select an Occurrence Report Type from a list and raise an Occurrence in Q-Pulse based on the content of the email.

From a technical perspective there are four main activities that need to be supported to achieve this:

1. Authentication against the database (for the purposes of this tutorial we will assume there is only one database configured).
2. Fetch the list of Occurrence Report Types.
3. Populate a Ribbon control displaying the list of Report Types.
4. Raise an Occurrence using the information from the body of an email.

With some more consideration, it would be reasonable to suggest that the Occurrence Report Type list be fetched before the Ribbon UI is built (preferably on Outlook start-up).

If the potentially time-consuming authentication check and Report Type list fetch was performed synchronously on Outlook start-up, the application responsiveness would be impaired. The solution is to invoke the Web Service operations in an asynchronous manner, so the process is non-blocking. This maintains application responsiveness; however, our code becomes slightly more complex, as we have to write event handlers and be aware of any new state the client could get into (loading the UI before data has been fetched).

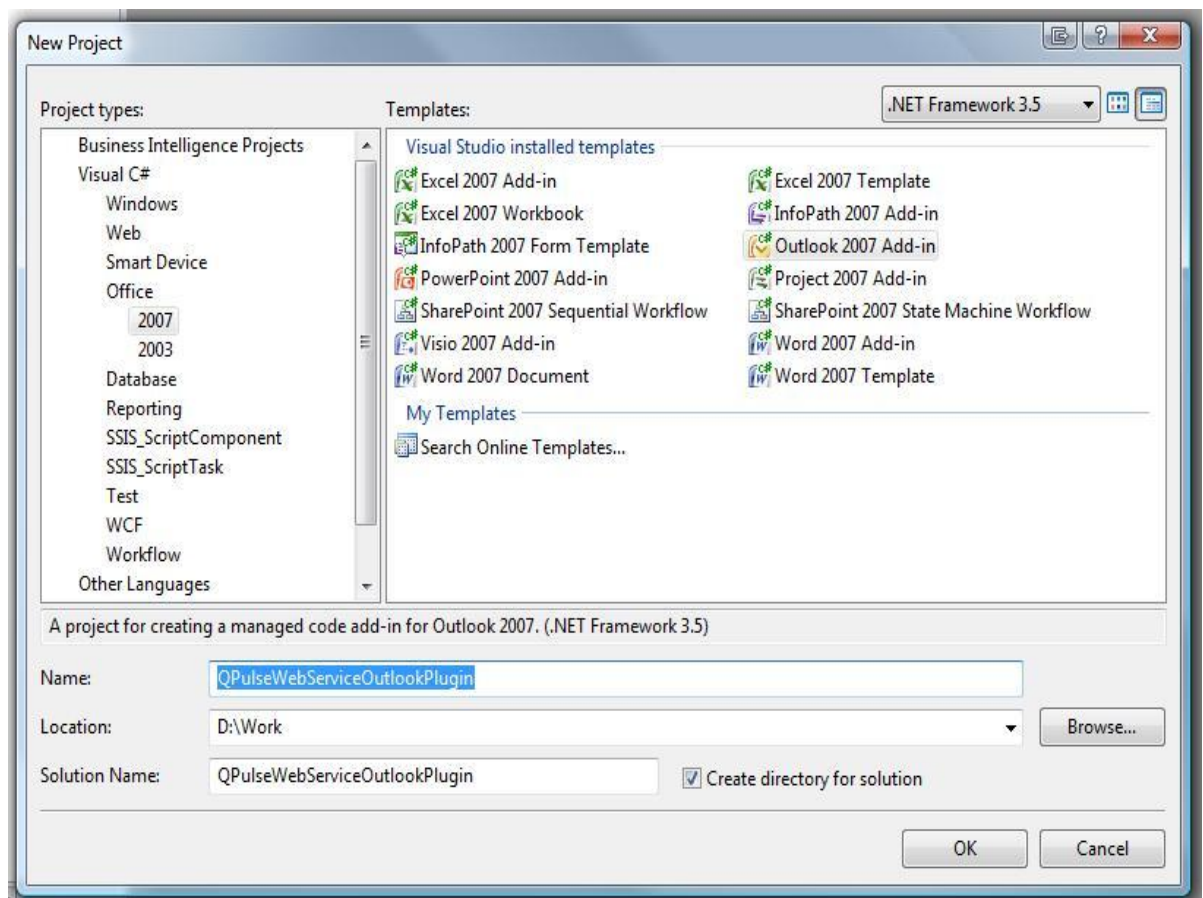
The refined technical perspective of the four activities now looks like:

1. On Outlook start-up, asynchronously authenticate the user against the database using Windows Authentication (the user must be configured for Windows Authentication).
2. Fetch the list of Occurrence Report Types asynchronously.
3. On first activation of the mail read window, build the Ribbon control (remembering that data might not yet be available).
4. Raise an Occurrence from the email to the Q-Pulse database (synchronously).

We will leave the description of the four activities at a high-level perspective for the time being, and go into more depth as we begin to implement the functionality.

## Getting Started

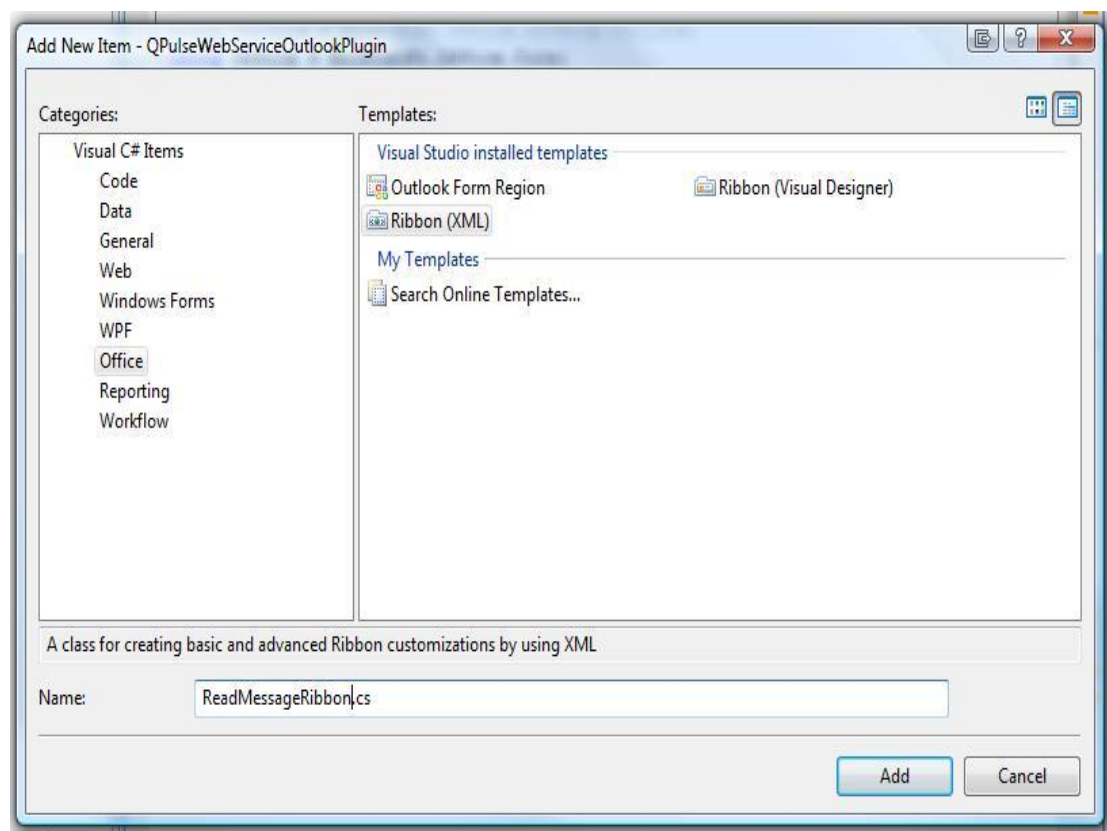
To create the correct type of project, go to File -> New -> Project. Under Visual C# -> Office 2007, choose the Outlook 2007 Add-in option and name it QPulseWebServiceOutlookPlugin.



This creates a new solution and project pre-populated with a file called ThisAddIn.cs. ThisAddIn has two methods ThisAddIn\_Startup and ThisAddIn\_Shutdown which Outlook invokes via reflection when loading the Add-In during application start-up and unloading the Add-In during application shutdown.

## Customising the Outlook Mail Read Ribbon

Right-click on the project in the Solution Explorer pane and choose Add -> New Item. Under Visual C# Items -> Office, choose Ribbon(XML) and name it ReadMessageRibbon



This creates two new files: ReadMessageRibbon.cs (where you put your logic) and ReadMessageRibbon.xml (where you define the Ribbon controls).

The next step is to override the CreateRibbonExtensibilityObject method in the ThisAddIn class which should return a new instance of ReadMessageRibbon:

```
protected override IRibbonExtensibility CreateRibbonExtensibilityObject()
{
    return new ReadMessageRibbon();
}
```

In ReadMessageRibbon.xml edit the XML to look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  onLoad="Ribbon_Load">
  <ribbon>
    <tabs>
      <tab idMso="TabReadMessage">
        <group id="Q-Pulse" label="Q-Pulse">
          <dynamicMenu
            id="occurrenceMenu"
            label="Raise Occurrence"
            size="large"
            screentip="Raises an Occurrence in Q-Pulse"
            getImage="GetOccurrenceImage"
            getContent="PopulateMenu"
            getEnabled="GetEnabled"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

The first notable element 'tab' specifies where the new control should be added. We use the 'idMso' attribute to specify that we want to add the control to an existing tab called 'TabReadMessage' which is the one displayed when an email is opened.

You can find all the possible Office 2007 Control ID values by downloading the [list](#). If the "id" attribute is used instead of 'idMso', a new tab will be created.

The next XML element 'group' creates a new group within the tab.

The 'dynamicMenu' element creates the actual control that we are interested in. The attributes 'getImage', 'getContent' and 'getEnabled' respectively specify the callback methods to get the control's image, the menu data to display and whether the control should be enabled or not. There are more attributes available, but these are the ones most applicable for this Add-In.

In the ReadMessageRibbon class we now need to declare those three methods:

```
public IPictureDisp GetOccurrenceImage(IRibbonControl control)
{
    return null;
}

public string PopulateMenu(IRibbonControl control)
{
    return string.Empty;
}

public bool GetEnabled(IRibbonControl control)
{
    return false;
}
```

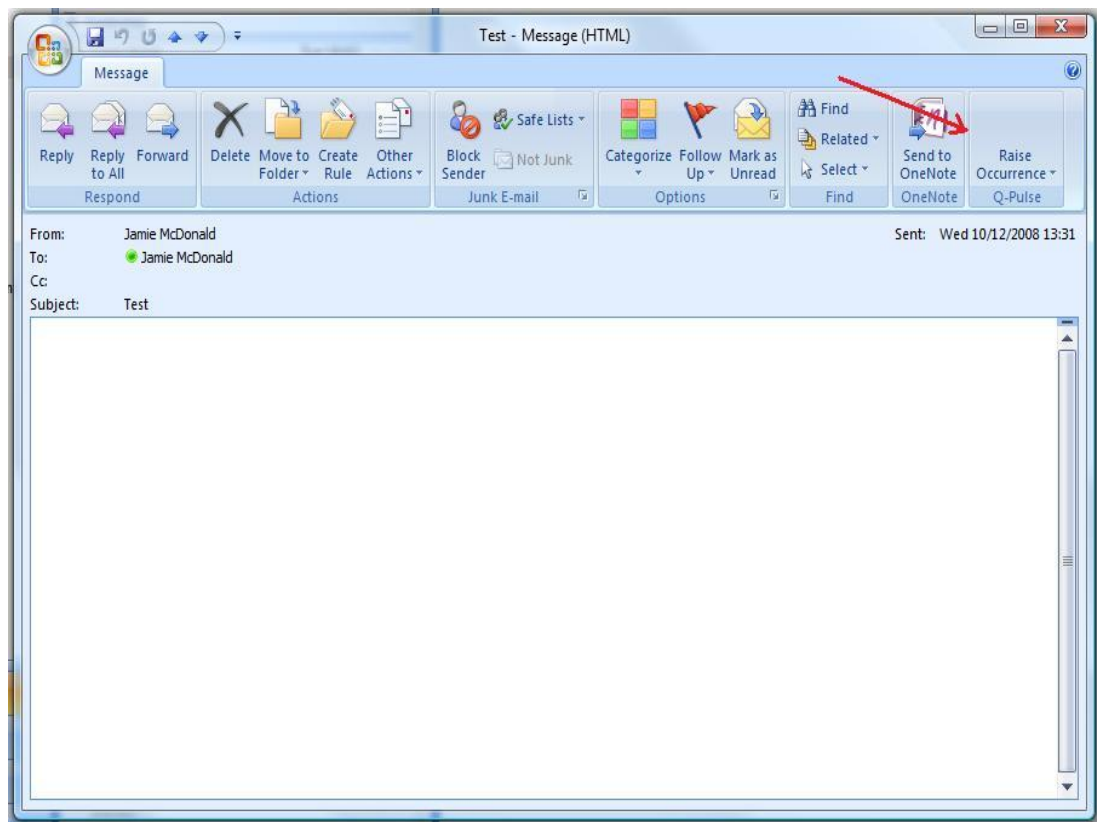
In the 'IRibbonExtensibility Members' region there should be a method called GetCustomUI. This method is called by Outlook and should return the XML of controls to be added (in our case the XML from ReadMessageRibbon.xml). The method is called for each ribbon Outlook builds whereas we are only interested in adding our control to the Ribbon being built for reading emails. The string passed into the method specifies which Ribbon is being built. (a list of possible Ribbon IDs is listed in this [article](#)). The one we are interested in is 'Microsoft.Outlook.Mail.Read'. When the RibbonID passed into the method is not this, we just want to return an empty string. It's also good practice to add the RibbonMessage.xml file to the project resources and refer to it from there (right-click on the Project, Properties and the Resources tab and drag the file on to the space). The method should now look like this:

```
public string GetCustomUI(string ribbonID)
{
    if (ribbonID == "Microsoft.Outlook.Mail.Read")
    {
        return Properties.Resources.ReadMessageRibbon;
    }

    return String.Empty;
}
```

## Running the Add-In

The project should now be in a state to run. Make sure there are no other instances of Outlook running and hit F5 in Visual Studio to Start Debugging. If all is well, an assembly signing key should appear in the Solution Explorer pane called QPulseWebServiceOutlookPlugin\_TemporaryKey.pfx and Outlook should start. When you open an email, the new control should now be visible.

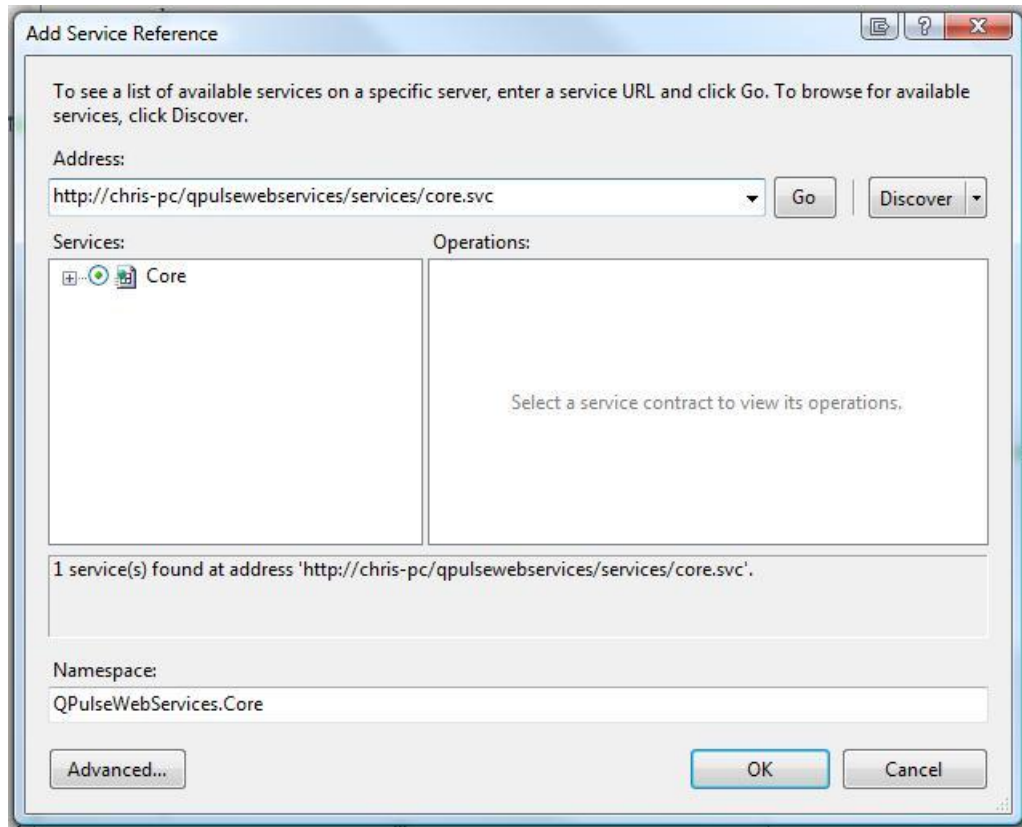


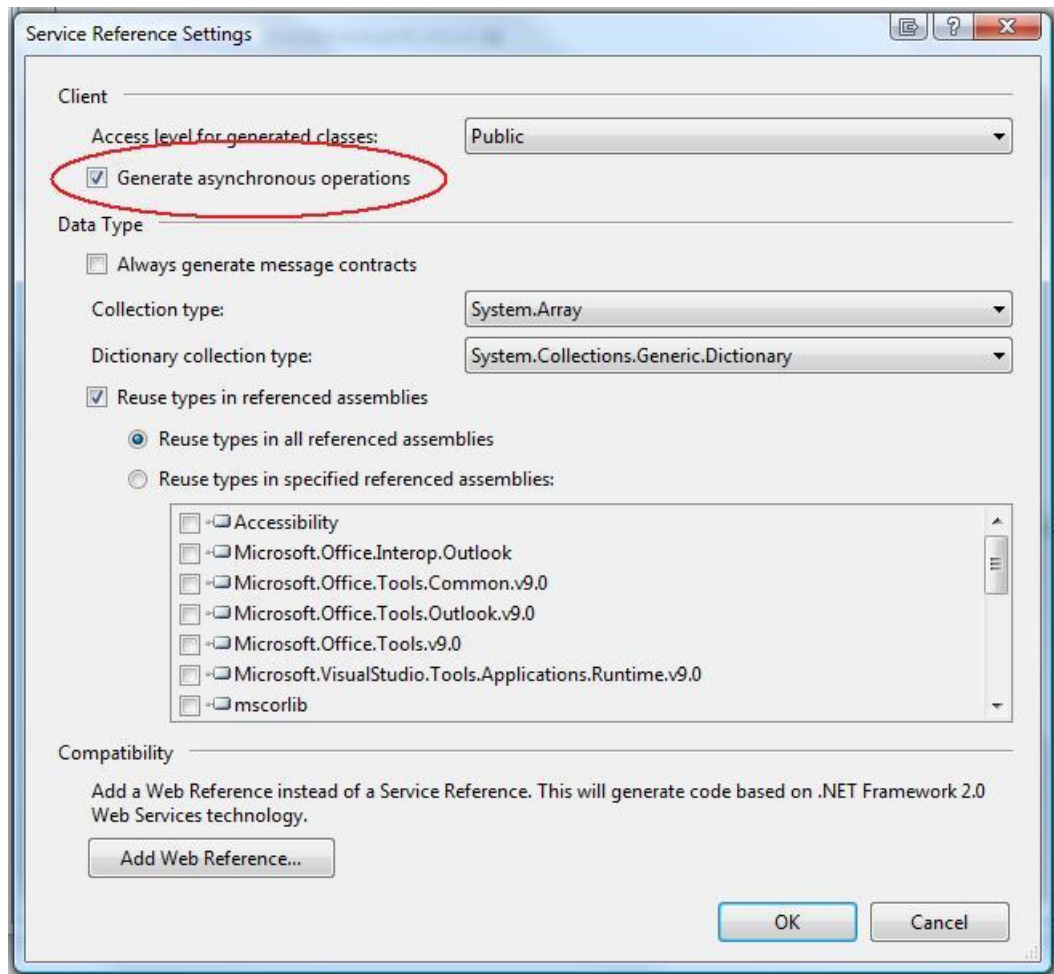
## Generating the Service Proxies

Now we've got the initial control in place, we need to consider connecting to the Q-Pulse Web Services. We need to connect to two service endpoints, Core and Occurrence: Core.svc will be used to attempt authentication. Occurrence.svc will be used to get the list of Report Types and raise an Occurrence in Q-Pulse.

Right-click on the References folder of the Project and choose 'Add New Service Reference'. In the dialog, enter the address to the Core service and click the 'Go' button. If properly installed, the Web Service should be available from the machine it was installed on and the URL should be of the form <http://machineName/gpulse5webservices/service/core.svc>, where **machineName** is the name of the machine where the Web Service is hosted. (If you are unable to locate it, please refer to the 'Testing the Web Services Installation' section of this document). Change the string in the 'Namespace' text box to 'QPulseWebServices.Core'

Before clicking the 'OK' button, click on the 'Advanced' button and tick the 'Generate asynchronous operations' box in order to generate the methods required to invoke the Web Service asynchronously. All other settings should remain at default.





Add another Service Reference in a similar fashion for Occurrence.svc and name it QPulseWebServices.Occurrence.

In the app.config file there will be duplicate service endpoints declared. Only one must be defined for each endpoint, so comment out or delete the 'customBinding' for both endpoints. (Failing to do so will result in a System.InvalidOperationException). The reason that Visual Studio puts in a WebHttpBinding endpoint configuration is due to the Q-Pulse Web Services offering backwards compatibility with ASP.NET Web Services.

The sections to comment out or delete should look like:

```
<endpoint binding="customBinding"
  bindingConfiguration="WebHttpBinding_ICore"
    contract="QPulseWebServices.Core.ICore"
  name="WebHttpBinding_ICore" />
```

and

```
<endpoint binding="customBinding"
  bindingConfiguration="WebHttpBinding_IOccurrence"
    contract="QPulseWebServices.Occurrence.IOccurrence"
  name="WebHttpBinding_IOccurrence" />
```

## Q-Pulse Web Service Operations

There are three asynchronous calls to the Web Service that are required:

1. Authenticate the user to obtain a token (later used for raising an Occurrence).
2. Fetch the list of Occurrence Report Types.
3. Raise an Occurrence using information from the current email being viewed.

The first two of these operations will be initiated at startup from the ReadMessageRibbon constructor like so:

```
public ReadMessageRibbon ()
{
    WebServiceAttemptAuthentication();
    WebServiceGetOccurrenceReportTypes();
}
```

These two operations can be performed in parallel, as they do not rely on each other, i.e. fetching of Report Types doesn't require an authentication token.

The `WebServiceAttemptAuthentication()` method looks like this:

```
private void WebServiceAttemptAuthentication()
{
    using (var client = new CoreClient())
    {
        // subscribe to the AuthenticateWindowsIDCompleted
        client.AuthenticateWindowsIDCompleted += (sender,
        eventArgs) =>
        {
            if (eventArgs.Error == null)
            {
                // get the token result, for later use
                mAuthenticationToken = eventArgs.Result;
            }
            else
            {
                ExceptionHandler.HandleException(eventArgs.Error);
            }
        };

        // make the call to the Web Service to start the
        asynchronous
        authentication

        client.AuthenticateWindowsIDAsync(GetCurrentWindowsID(),
        mDatabase);
    }
}
```

It subscribes an event handler to the `AuthenticateWindowsIDCompleted` event which gets the authentication token. The authentication token will be used later to determine whether to enable/disable the button on the Ribbon and will be used when raising an Occurrence. The database is a class-level string and the username is obtained from the current thread Principal and Identity which is encapsulated in the `GetCurrentWindowsID`:

```
private static string GetCurrentWindowsID()
{
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    return Thread.CurrentPrincipal.Identity.Name;
}
```

The `WebServiceGetOccurrenceReportTypes()` method looks like this:

```
private void WebServiceGetOccurrenceReportTypes()
{
    using(var client = new OccurrenceClient())
    {
        // subscribe to the ListAvailableReportTypesCompleted event
        client.ListAvailableReportTypesCompleted += (sender, eventArgs) =>
        {
            // get the names of the ReportTypes out of the array of
            ManagedListItems
            // these will now be available for the drop-down menu to display
            mOccurrenceReportTypes =
                from listItem in eventArgs.Result.Items
                select listItem.Name;

            RefreshRibbon();

        };

        // make the call to the Web Service
        client.ListAvailableReportTypesAsync(mDatabase);
    }
}
```

It subscribes to the `ListAvailableReportTypesCompleted` event, which then populates a class-level `IEnumerable` field with the names of the `OccurrenceReportTypes`, ready for the drop-down menu to display. The call to `RefreshRibbon` causes the Ribbon to be invalidated to update the Ribbon control in case it has already been built.

The RefreshRibbon() method looks like:

```
private void RefreshRibbon()
{
    // if the ribbon has been created, get it to refresh
    if (mRibbon != null)
    {
        // this will cause GetOccurrenceImage, GetEnabled and PopulateMenu
        methods to be called.
        mRibbon.InvalidateControl("occurrenceMenu");
    }
}
```

The third Web Service operation, looks like this:

```
private void WebServiceRaiseOccurrence(IRibbonControl control, string
reportType)
{
    var inspector = (Inspector)control.Context;
    var currentEmail = (MailItem)inspector.CurrentItem;

    // make the call to the Web Service to raise a new Occurrence
    using (var client = new OccurrenceClient())
    {
        // subscribe to the RaiseNewOccurrenceCompleted to handle the result
        client.RaiseNewOccurrenceCompleted += (sender, EventArgs) =>
        {
            Occurrence occ = EventArgs.Result;
            if (occ.ID > 0 && occ.BrokenRules.Length == 0)
            {
                MessageBox.Show(
                    string.Format(
                        "Occurrence was successfully raised in Q-Pulse with Record ID:
{0}",
                        occ.Number));
            }
            else
            {
                var sb = new StringBuilder();
                foreach (var rule in occReceived.BrokenRules)
                {
                    sb.AppendLine(rule.Description);
                }
                MessageBox.Show(string.Format("Raising of Occurrence failed.
The broken rules were: {0}", sb)
            );
        }
    };

    // create a new Occurrence with the data in preparation for sending to
Web Service
    var occurrence = new Occurrence
    {
        DataFields = new[] { new DataField {
                                Name = "Occurrence
Description",
                                Value = currentEmail.Body } },
        RecordedByPersonName = currentEmail.SenderName,
        ReportTypeName = reportType
    };

    // make the call to the Web Service
    client.RaiseNewOccurrenceAsync(mAuthenticationToken, occurrence);
}
```

```
    }
}
```

When creating the new Occurrence with the DataField, we are making the assumption that each of the current Report Types has a DataField called “Occurrence Description”.

## Implementing the rest of the Ribbon Control

### Implementing 'GetOccurrenceImage', 'PopulateMenu' and 'GetEnabled'

#### GetOccurrenceImage

The callback for the 'getImage' property of the Ribbon control expects an image of type `stdole.IPictureDisp`. This may seem like an odd type request, but it's due to the fact that Outlook was not developed on the .NET platform (we integrate with it via COM interop). One way of converting a .NET `System.Drawing.Image` to a `stdole.IPictureDisp` is to create a class which inherits from `System.Windows.Forms.AxHost` and use the protected `GetIPictureDispFromPicture` method to perform the conversion (please see the `IconHelper.cs` file in the root folder of the tutorial source code). The class we created to achieve this is called `IconHelper`, so the `GetOccurrenceImage` now looks like:

```
public IPictureDisp GetOccurrenceImage(IRibbonControl control)
{
    return
    IconHelper.ImageToPictureDisp(Properties.Resources.OccurrenceIcon);
}
```

(Additionally, we added a folder to the solution called `Images`, added a `.gif` to it and dragged the image into the Project Resources tab).

#### PopulateMenu

The `PopulateMenu` method should return a string of XML elements which describe the menu items to be displayed in the control. The implementation consists of looping through the Report Type list, building up a string of XML. The method content is trivial and verbose, so it's left to the user to view the source code for this method.

The only thing worth noting is that the menu buttons have a callback method (`MenuButtonClick`) specified for their 'onAction' attributes:

```
public void MenuButtonClick(IRibbonControl control)
{
    // find out which Report Type was clicked and raise the
    Occurrence
    string reportType = GetClickedReportType(control);
    WebServiceRaiseOccurrence(control, reportType);
}
```

This method simply makes a call to `'GetClickedReportType'` (see tutorial source code) to identify which button on the menu was clicked and then calls `'WebServiceRaiseOccurrence'` method which

raises an Occurrence via the Web Service. The method also displays a MessageBox to let the user know if the raising of the Occurrence succeeded or not.

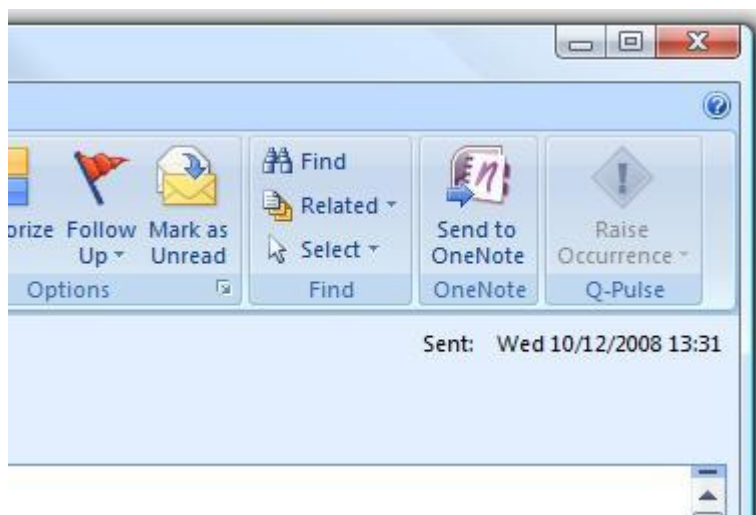
### GetEnabled

The GetEnabled method has to determine whether the control should be enabled based on whether all Report Type data has been successfully fetched from the Web Service.

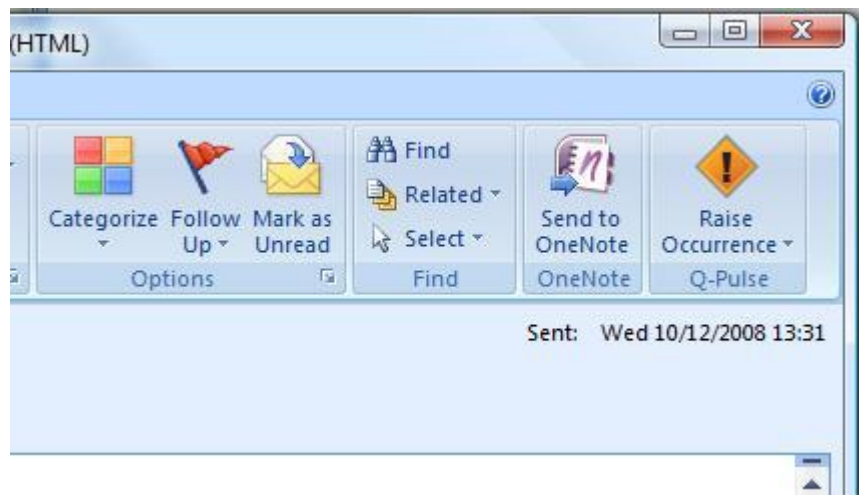
The method simply looks like:

```
public bool GetEnabled(IRibbonControl control)
{
    return mOccurrenceReportTypes.Count() > 0;
}
```

The control in disabled state look like this:

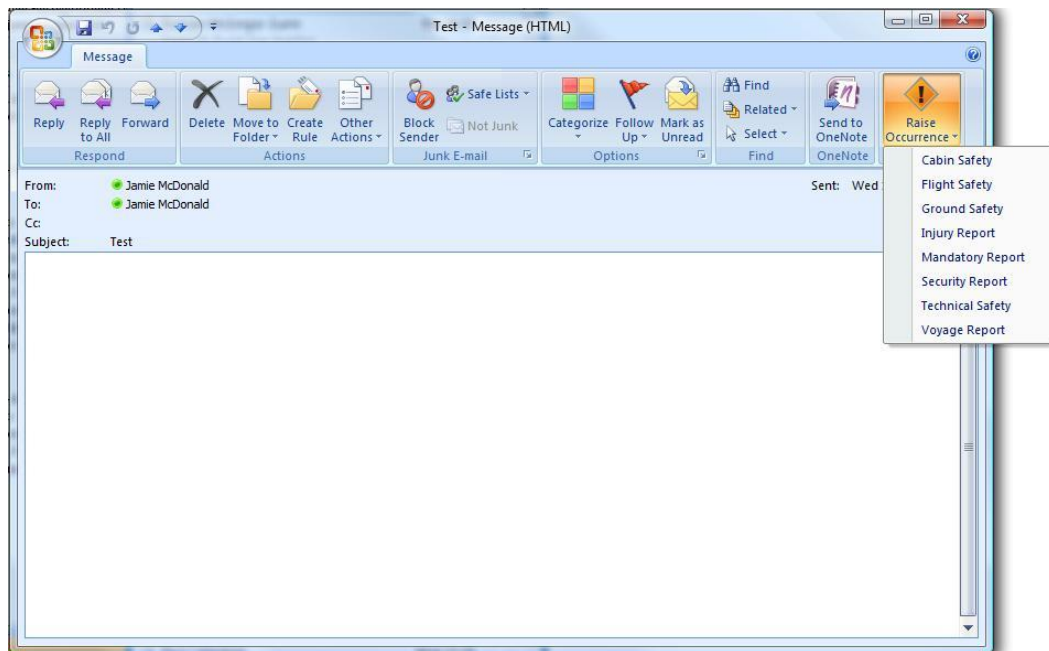


And when enabled, looks like this:

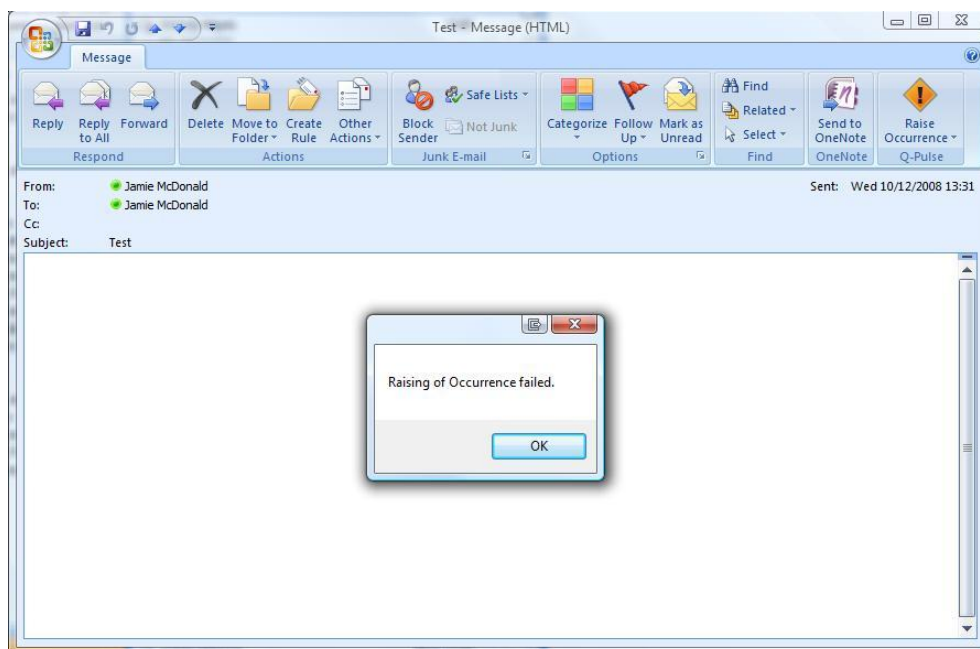


## The Final Result

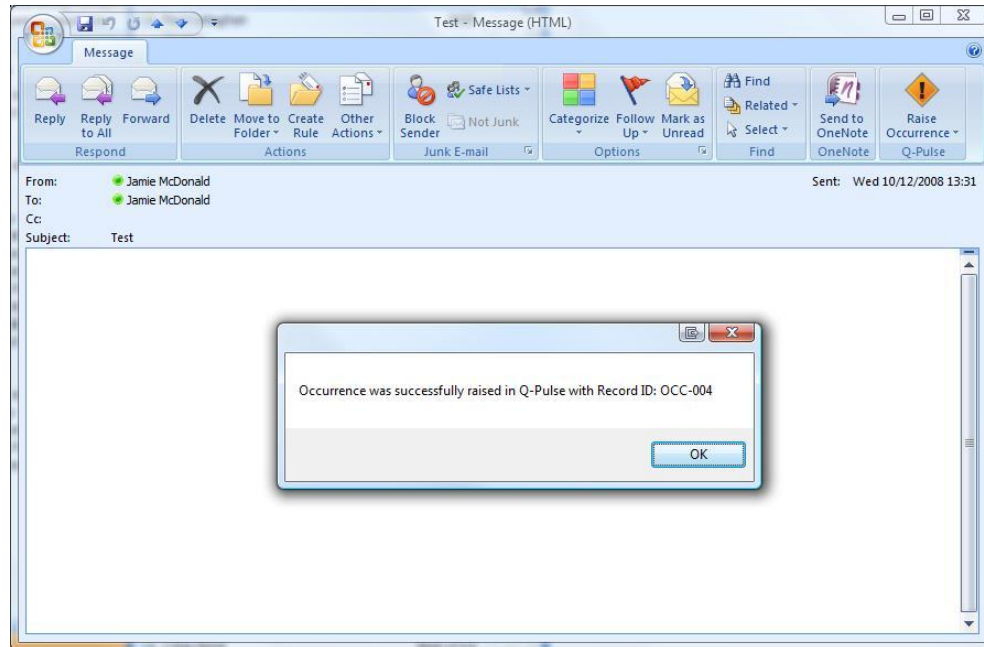
The menu being displayed when the data has been fetched:



The message box displayed when the raising of an Occurrence failed:



The message box displayed when the raising of an Occurrence succeeds:



## Limitations

There is one notable limitation which could be considered for enhancement in the next iteration of development.

- The Report Type list is loaded only on application start-up. It is possible that a ReportType is added/renamed/deleted in the database after this point, in which case the Outlook user does not have an accurate list to choose from and may result in failure to raise an Occurrence.

## Useful Links

- Office Development with Visual Studio Developer Portal - <http://msdn.microsoft.com/en-us/vsto/default.aspx>



**Gael Ltd.**

Orion House,  
S. E. Technology Park,  
East Kilbride,  
Scotland. G75 0RD

**t:** +44 1355 593400

**f:** +44 1355 579191

**e:** [info@gaelquality.com](mailto:info@gaelquality.com)

**w:** [www.gaelquality.com](http://www.gaelquality.com)

Q-Pulse is a registered trademark of Gael Products Ltd. All rights reserved worldwide.  
Copyright © 2010 Gael Products Ltd. Gael Quality is a trading division of Gael Ltd.