Creating and Debugging Office Add-ins

Lab Time: 60 minutes

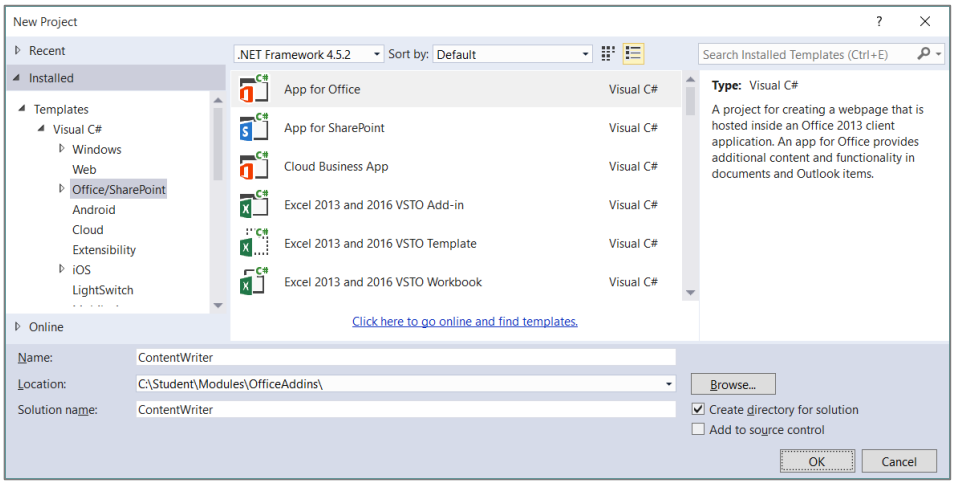
Lab Folder: C:\Student\Modules\OfficeAddins\Labs

Lab Overview: In this lab you will get hands-on experience developing an Office Add-ins for Word, Excel and Outlook.

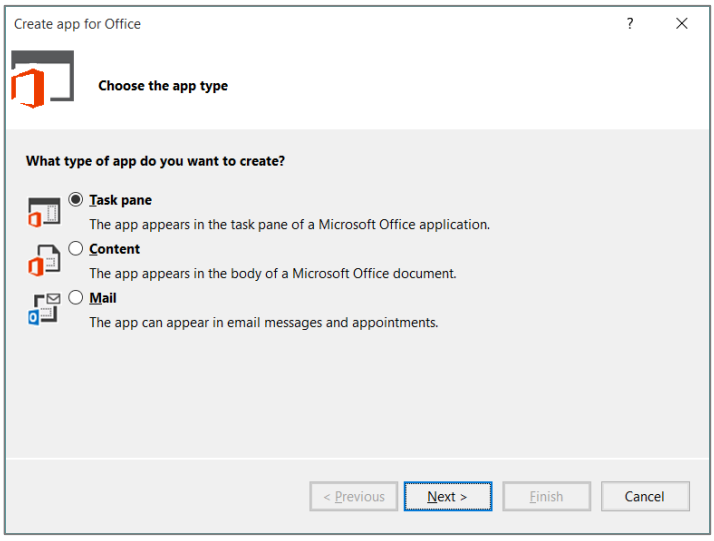
Exercise 1: Creating the ContentWriter Add-in Office Project

In this exercise you will create a new Office Add-in project in Visual Studio so that you can begin to write, test and debug an Office Word Add-in. The user interface of the Office Add-in you will create in this lab will not be very complicated as it will just contain HTML buttons and JavaScript command handlers.

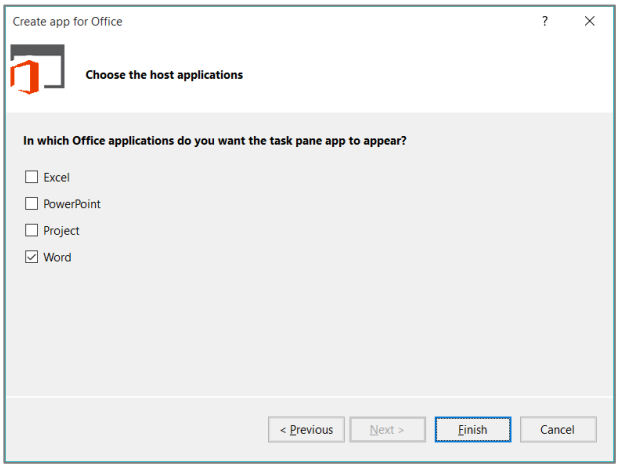
1. Launch Visual Studio 2015 as administrator.
2. From the **File** menu select the **New Project** command. When the **New Project** dialog appears, select the **App for Office** project template from the Office/SharePoint template folder as shown below. Name the new project **ContentWriter** and click OK to create the new project.



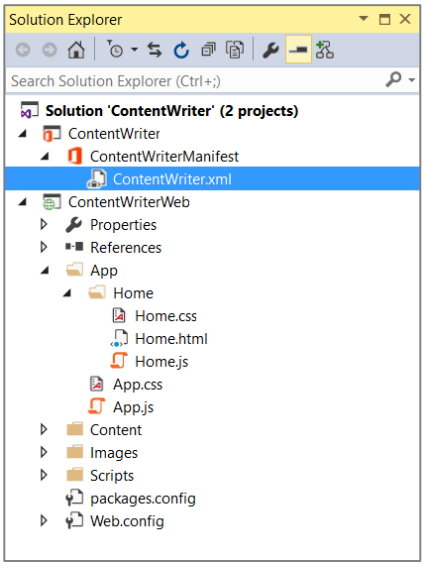
3. When you create a new **App for Office** project, Visual Studio prompts you with the **Choose the app** type page of the **Create app for Office** dialog. This is the point where you select the type of App for Office you want to create. Leave the default setting with the radio button titled **Task pane** and select **OK** to continue.



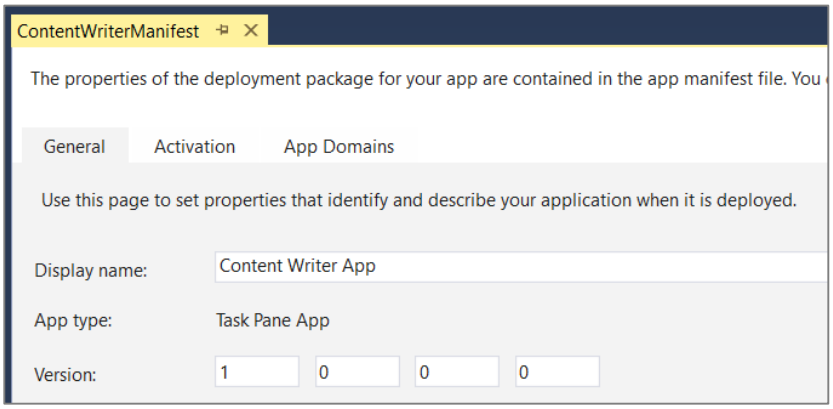
4. On the **Choose the host applications** page of the **Create app for Office** dialog, uncheck all the Office application except for **Word** and then click **Finish** to create the new Visual Studio solution.



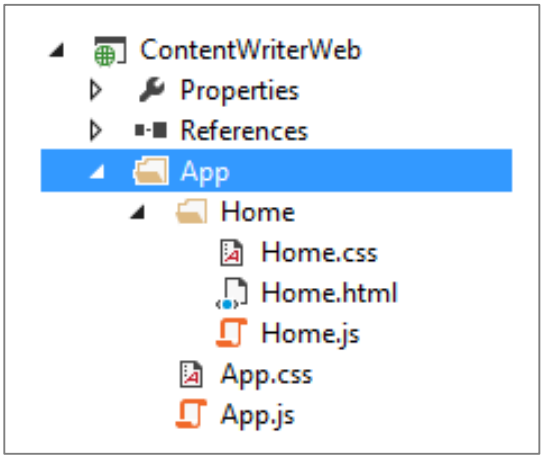
5. Take a look at the structure of the new Visual Studio solution once it has been created. At a high-level, the new solution has been created using two Visual Studio projects named **ContentWriter** and **ContentWriterWeb**. You should also observe that the top project contains a top-level manifest for the app named **ContentWriterManifest** which contains a single file named **ContentWriter.xml**.



6. In the Solution Explorer, double-click on the node named **ContentWriterManifest** to open the app manifest file in the Visual Studio designer. Update the Display Name settings in the app manifest from **ContentWriter** to **Content Writer App**.



7. Save and close **ContentWriterManifest**.
8. Over the next few steps you will walk through the default app implementation that Visual Studio generated for you when the app project was created. Begin by looking at the structure of the app folder which has two important files named **app.css** and **app.js** which contain CSS styles and JavaScript code which is to be used on an app-wide basis.



9. You can see that inside the app folder there is a child folder named **Home** which contains three files named **Home.html**, **Home.css** and **Home.js**. Note that the app project is currently configured to use **Home.html** as the app's start page and that **Home.html** is linked to both **Home.css** and **Home.js**.
10. Double-click on **app.js** to open it in a code editor window. You should be able to see that the code creates a global variable named **app** based on the JavaScript Closure pattern. The global app object defines a method named **initialize** but it does not execute this method.

```
var app = (function () {
  "use strict";

  var app = {};

  // Common initialization function (to be called from each page)
  app.initialize = function () {
    $('body').append(
      '<div id="notification-message">' +
      '<div class="padding">' +
      '<div id="notification-message-close"></div>' +
      '<div id="notification-message-header"></div>' +
      '<div id="notification-message-body"></div>' +
      '</div>' +
      '</div>');

      $('#notification-message-close').click(function () {
        $('#notification-message').hide();
      });
```

```
      // After initialization, expose a common notification function
      app.showNotification = function (header, text) {
        $('#notification-message-header').text(header);
        $('#notification-message-body').text(text);
        $('#notification-message').slideDown('fast');
      };
    };

      return app;
})();
```

11. Close **app.js** and be sure not to save any changes.
12. Next you will examine the JavaScript code in **home.js**. Double-click on **home.js** to open it in a code editor window. Note that **Home.html** links to **app.js** before it links to **home.js** which means that JavaScript code written in **Home.js** can access the global **app** object created in **app.js**.
13. Walk through the code in **Home.js** and see how it uses a self-executing function to register an event handler on the **Office.initialize** method which, in turn, registers a document-ready event handler using jQuery. This allows the app to call **app.initialize** and to register an event handler using the **getDataFromSelection** function.

```
(function () {
  "use strict";

  // The initialize function must be run each time a new page is loaded
  Office.initialize = function (reason) {
    $(document).ready(function () {
      app.initialize();
      $('#get-data-from-selection').click(getDataFromSelection);
    });
  };

  // Reads data from current document selection and displays a notification
  function getDataFromSelection() {
    Office.context.document.getSelectedDataAsync(Office.CoercionType.Text,
      function (result) {
        if (result.status === Office.AsyncResultStatus.Succeeded) {
          app.showNotification('The selected text is:', '"' + result.value + '"');
        } else {
          app.showNotification('Error:', result.error.message);
        }
    });
  }
})();
```

14. Delete the **getDataFromSelection** function from **Home.js** and also remove the line of code that binds the event handler to the button with the **id** of **get-data-from-selection** so your code matches the following code listing.

```
(function () {
  "use strict";

  // The initialize function must be run each time a new page is loaded
  Office.initialize = function (reason) {
    $(document).ready(function () {
      app.initialize();
      // your app initialization code goes here
    });
  };

})();
```

15. Save your changes to **Home.js**. You will return to this source file after you have added your HTML layout to **Home.html**.
16. Now it time to examine the HTML that has been added to the project to create the app's user interface. Double-click **Home.html** to open this file in a Visual Studio editor window. Examine the layout of HTML elements inside the body element.

```
<body>
    <div id="content-header">
        <div class="padding">
            <h1>Welcome</h1>
        </div>
    </div>
    <div id="content-main">
        <div class="padding">
```

```
            <p><strong>Add home screen content here.</strong></p>
            <p>For example:</p>
            <button id="get-data-from-selection">Get data from selection</button>

            <p style="margin-top: 50px;">
                <a target="_blank" href="https://go.microsoft.com/fwlink/?LinkId=276812">Find more...</a>
            </p>
        </div>
    </div>
</body>
```

17. Replace the text message of Welcome inside the **h1** element with a different message such as **Add Content to Document**. Also trim down the contents of the **div** element with the **id** of **content-main** to match the HTML code shown below.

```
<body>
    <div id="content-header">
        <div class="padding">
            <h1>Add Content to Document</h1>
        </div>
    </div>
    <div id="content-main">
        <div class="padding">
            <!-- your app UI layout goes here -->
        </div>
    </div>
</body>
```

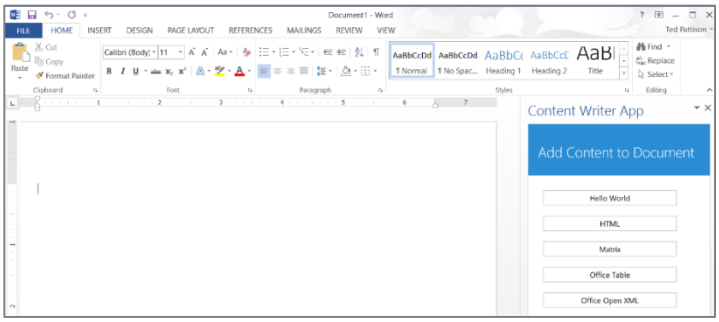
18. Update the **content-main** div to match the following HTML layout which adds a set of buttons to the app's layout.

```
<div id="content-main">
    <div class="padding">
        <div>
            <button id="addContentHellowWorld">Hello World</button>
        </div>
        <div>
            <button id="addContentHtml">HTML</button>
        </div>
        <div>
            <button id="addContentMatrix">Matrix</button>
        </div>
        <div>
            <button id="addContentOfficeTable">Office Table</button>
        </div>
        <div>
            <button id="addContentOfficeOpenXml">Office Open XML</button>
        </div>
    </div>
</div>
```

19. Save and close **Home.html**.
20. Open the CSS file named **Home.css** and add the following CSS rule to ensure all the app's command buttons and select element have a uniform width and spacing.

```
#content-main button, #content-main select{
  width: 210px;
  margin: 8px;
}
```

21. Save and close **Home.css**.
22. Now it's time to test the app using the Visual Studio debugger. Press the {F5} key to run the project in the Visual Studio debugger. The debugger should launch Microsoft Word 2013 and you should see your App for Office in the task pane on the right side of a new Word document as shown in the following screenshot.



23. Close Microsoft Word to terminate your debugging session and return to Visual Studio.
24. Return to the source file named **Home.js** or open it if it is not already open.
25. Add a new function named **testForSuccess** with the following implementation.

```
function testForSuccess(asyncResult) {
    if (asyncResult.status === Office.AsyncResultStatus.Failed) {
        app.showNotification('Error', asyncResult.error.message);
    }
}
```

26. Create a function named **onAddContentHellowWorld** and add the following call to **setSelectedDataAsync**.

```
function onAddContentHellowWorld() {
    Office.context.document.setSelectedDataAsync("Hello World!", testForSuccess);
}
```

27. Finally, add a line of jQuery code into the app initialization logic to bind the click event of the **addContentHellowWorld** button to the **onAddContentHellowWorld** function.

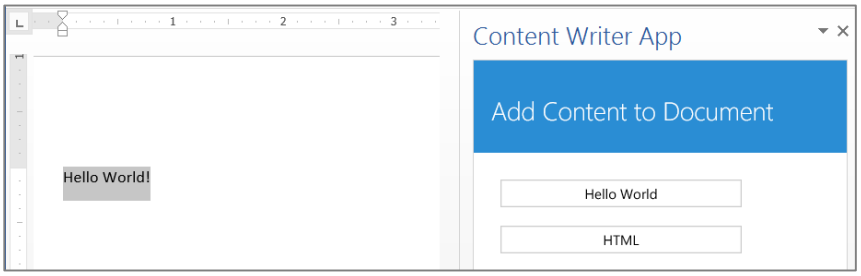
```
Office.initialize = function (reason) {
    $(document).ready(function () {
        app.initialize();
        // add this code to wire up event handler
        $("#addContentHellowWorld").click(onAddContentHellowWorld)
    });
};
```

28. When you are done, the **Home.js** file should match the following listing.

```
(function () {
    "use strict";

    // The initialize function must be run each time a new page is loaded
    Office.initialize = function (reason) {
        $(document).ready(function () {
            app.initialize();
            // wire up event handler
            $("#addContentHellowWorld").click(onAddContentHellowWorld)
        });
    };

    // write text data to current at document selection
    function onAddContentHellowWorld() {
        Office.context.document.setSelectedDataAsync("Hello World!", testForSuccess);
    }

    function testForSuccess(asyncResult) {
        if (asyncResult.status === Office.AsyncResultStatus.Failed) {
            app.showNotification('Error', asyncResult.error.message);
        }
    }

})();
```

29. Save your changes to **Home.js**.

30. Now test the functionality of the app. Press the **{F5}** key to begin a debugging session and click the **Hello World** button. You should see that "Hello World" has been added into the cursor position of the Word document.



You have now successfully run and tested the app and its JavaScript logic using the Visual Studio debugger. Close Microsoft Word to stop the debugging session and return to Visual Studio.

Exercise 2: Writing to a Word Document Using Coercion Types and Open XML

In this exercise you will continue working on the Visual Studio solution for the **ContentWriter** app you created in the previous exercise. You will add additional JavaScript code to insert content into the current Word document in a variety of formats.

1. In Visual Studio, make sure you have the **ContentWriter** project open that you created in the previous exercise.
2. In the Solution Explorer, double click on **Home.js** to open this JavaScript file in an editor window.
3. Just below the **onAddContentHelloWorld** function, add four new functions named **onAddContentHtml onAddContentMatrix**, **onAddContentOfficeTable** and **onAddContentOfficeOpenXml**.

```
function onAddContentHellowWorld() {
    Office.context.document.setSelectedDataAsync("Hello World!", testForSuccess);
}

function onAddContentHtml() {
}

function onAddContentMatrix() {
}

function onAddContentOfficeTable() {
}

function onAddContentOfficeOpenXml() {
}
```

4. Just below the call to **app.initialize**, add the jQuery code required to bind each of the four new functions to the click event of the associated buttons.

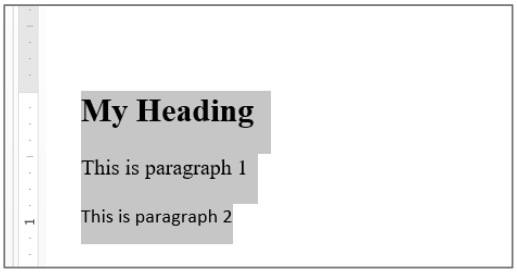
```
Office.initialize = function (reason) {
    $(document).ready(function () {
        app.initialize();
        // wire up event handler
        $("#addContentHellowWorld").click(onAddContentHellowWorld);
        $('#addContentHtml').click(onAddContentHtml);
        $('#addContentMatrix').click(onAddContentMatrix);
        $('#addContentOfficeTable').click(onAddContentOfficeTable);
        $('#addContentOfficeOpenXml').click(onAddContentOfficeOpenXml);
    });
};
```

5. Implement the **onAddContentHtml** function to create an HTML div element with several child elements using jQuery and then to write that HTML to the Word document using the HTML coercion type using the code in the following listing.

```
function onAddContentHtml() {
    // create HTML element
    var div = $("<div>")
            .append($("<h2>").text("My Heading"))
            .append($("<p>").text("This is paragraph 1"))
            .append($("<p>").text("This is paragraph 2"))
```

```
    // insert HTML into Word document
    Office.context.document.setSelectedDataAsync(div.html(), { coercionType: "html" }, testForSuccess);
}
```

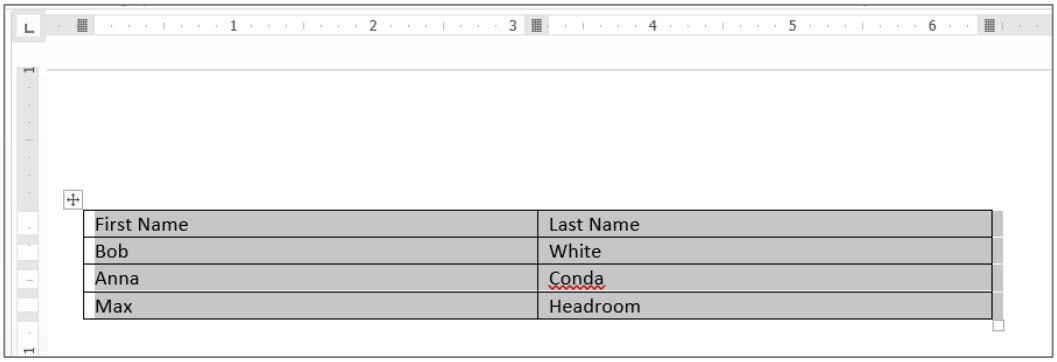
6. Test your work by starting a debug session and clicking the HTML button. When you click the button, you should see that the HTML content has been added to the Word document.



7. Implement **onAddContentMatrix** by creating an array of arrays and then by writing the matrix to the Word document using the matrix coercion type as shown in the following code listing.

```
function onAddContentMatrix() {
    // create matrix as an array of arrays
    var matrix = [["First Name", "Last Name"],
                  ["Bob", "White"],
                  ["Anna", "Conda"],
                  ["Max", "Headroom"]];

    // insert matrix into Word document
    Office.context.document.setSelectedDataAsync(matrix, { coercionType: "matrix" }, testForSuccess);
}
```

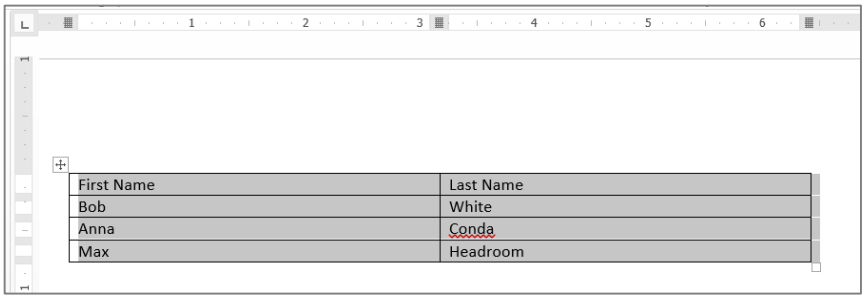
8. Test your work by starting a debug session and clicking the **Matrix** button. When you click the button, you should see that the content from the matrix has been added to the Word document as a table.



9. Implement **onAddContentOfficeTable** by creating a new **Office.TableData** object and then by writing it to the Word document using the table coercion type as shown in the following code listing.

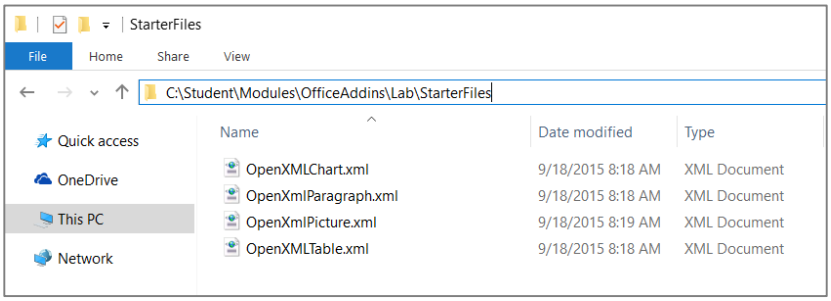
```
function onAddContentOfficeTable() {

    // create and populate an Office table
    var myTable = new Office.TableData();
    myTable.headers = [['First Name', 'Last Name']];
    myTable.rows = [['Bob', 'White'], ['Anna', 'Conda'], ['Max', 'Headroom']];

    // add table to Word document
    Office.context.document.setSelectedDataAsync(myTable, { coercionType: "table" }, testForSuccess)
}
```

10. Test your work by starting a debug session and clicking the **Office Table** button. When you click the button, you should see that the content from the Office Table object has been added to the Word document as a table.

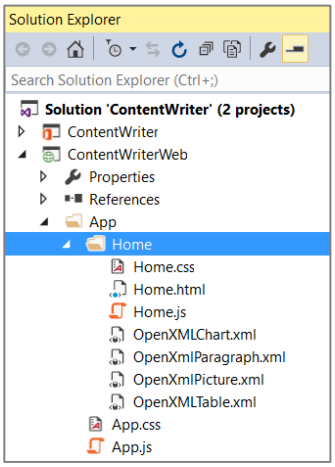


First Name	Last Name
Bob	White
Anna	Conda
Max	Headroom

11. Look inside the folder for this lab and locate the child folder named **StarterFiles**. You should see that this folder contains four XML files as shown in the following screenshot.



12. Add the four XML files into the Visual Studio project into the same folder as the HTML start page named **Home.html**.



13. Quickly open and review the XML content inside each of these four XML files. This will give you better idea of what Open Office XML looks like when targeting Microsoft Word.
14. Open **Home.html** and locate the button element with the id of **addContentOfficeOpenXml**. Directly under this button, add a new HTML select element as shown in the following code listing.

```
<div>
  <button id="addContentOfficeOpenXml">Office Open XML</button>
  <select id="listOpenXmlContent">
    <option value="OpenXmlParagraph.xml">Paragraph</option>
    <option value="OpenXmlPicture.xml">Picture</option>
    <option value="OpenXmlChart.xml">Chart</option>
    <option value="OpenXmlTable.xml">Table</option>
  </select>
</div>
```

15. Save and close **Home.html**.
16. Return to the code editor window with **Home.js**.

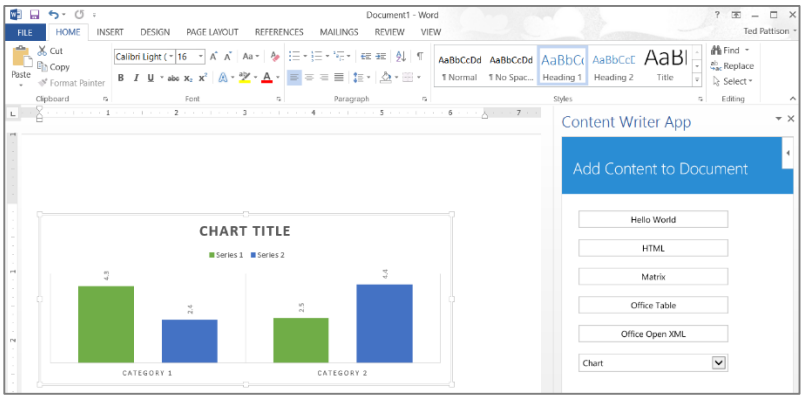
17. Implement the **onAddContentOfficeOpenXml** function to obtain the currently selected file name from the select element and then to execute an HTTP GET request using the jQuery $.ajax function to retrieve the associated XML file. In the success callback function, call **setSelectedDataAsync** to write the XML content to the current Word document using the **ooxml** coercion type as shown in the following code listing.

```
function onAddContentOfficeOpenXml() {
    var fileName = $("#listOpenXmlContent").val();

    $.ajax({
        url: fileName,
        type: "GET",
        dataType: "text",
        success: function (xml) {
            Office.context.document.setSelectedDataAsync(xml, { coercionType: "ooxml" }, testForSuccess)
        }
    });
}
```

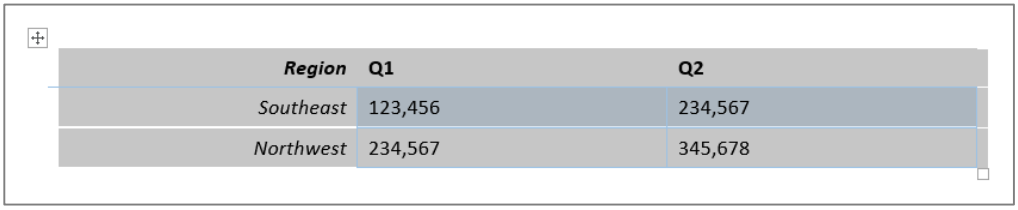
18. Test your work by starting a debug session and clicking the **Office Open XML** button when the select element has the default selected value of **Paragraph**. You should see that the Open Office XML content has been used to create a formatted paragraph.

19. Change the value of the select element to Picture and click the Office Open XML button. You should see that the Open Office XML content has been used to insert an image into the document.

20. Change the value of the select element to **Chart** and click the **Office Open XML** button. You should see that the Open Office XML content has been used to create a simple bar chart.



21. Change the value of the select element to Table and click the Office Open XML button. You should see that the Open Office XML content has been used to create a formatted table.



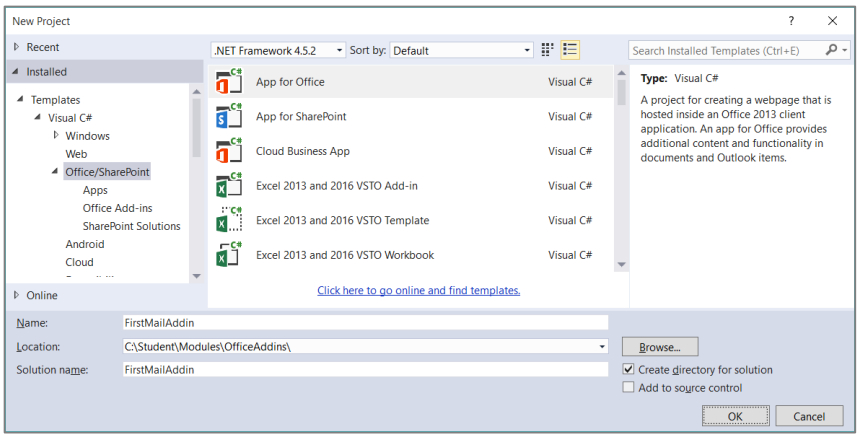
Region	Q1	Q2
Southeast	123,456	234,567
Northwest	234,567	345,678

Congratulations! In this exercise you extended the add-in's capabilities by adding JavaScript code to insert content into the active Word document using Open Office XML.

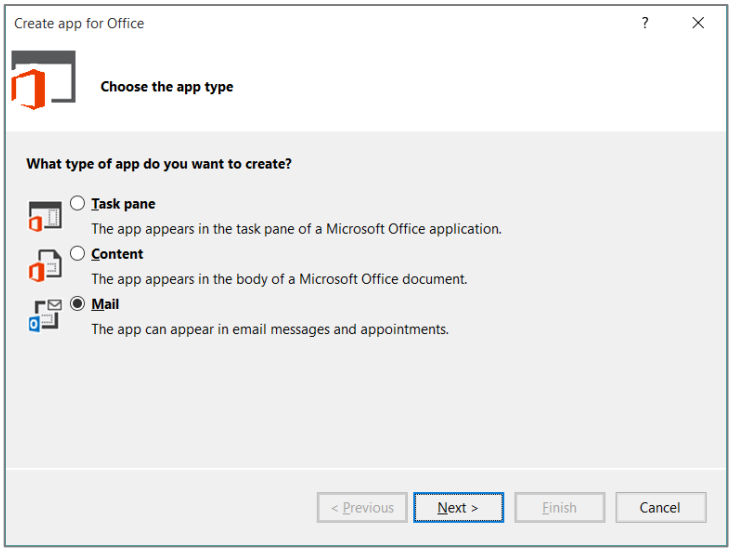
Exercise 3: Creating and Debugging a Mail App using Visual Studio

In this exercise, you will create a new Outlook Add-in for reading and creating mail that you will deploy to both the Outlook Web App in Office 365 and Outlook Desktop client on Windows.

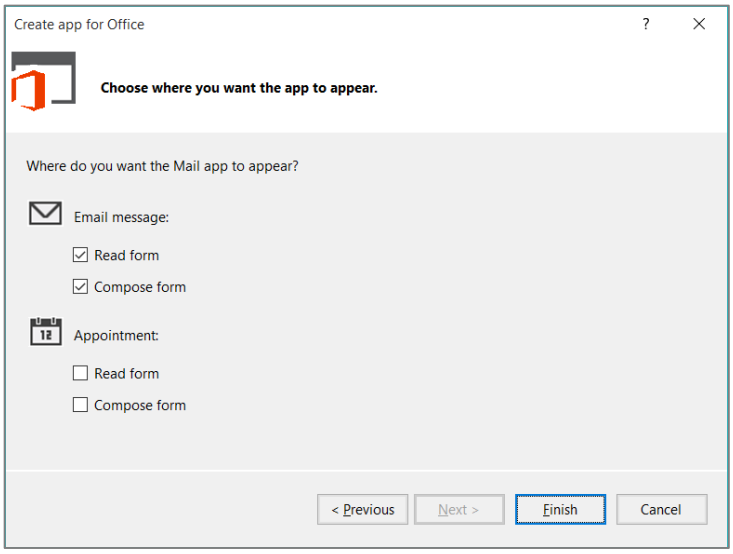
1. Launch **Visual Studio 2015** if it is not already running.
2. In Visual Studio select **File/New/Project**.
3. In the **New Project** dialog, select **Templates, Visual C#, Office/SharePoint** and click **App for Office**. Name the new project **FirstMailAddin** and then click **OK**.



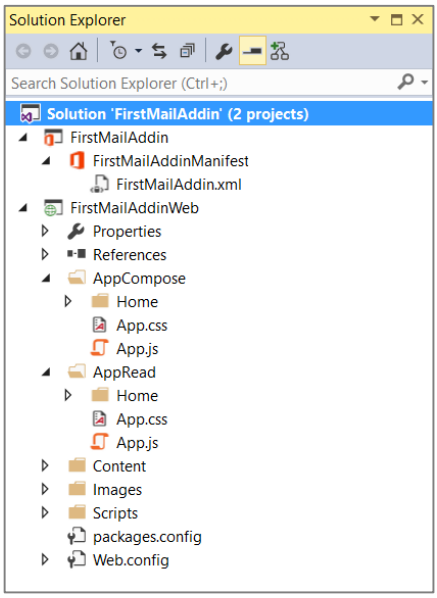
4. On the **Create app for Office / Choose the app type** dialog, select **Mail** and click **Next**.



5. On the **Create app for Office / Choose where you want the app to appear**, check only the two options for **Email message** and click **Finish**.



6. Take a moment to examine the project structure.



a) Inspect the Outlook Add-in manifest file which is located in the Office Add-in project at the top of the Solution Explorer window. This is the file that will tell the hosting Office client application, Outlook, about the Add-in and where the web application that implements the Add-in resides.

b) Next take a look at the **AppCompose** and **AppRead** folders. These contain the client-side applications that host the two different experiences for the Add-in you created.

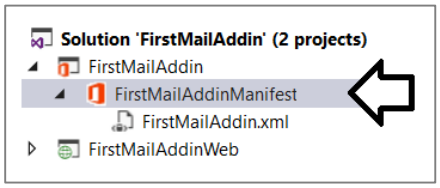
c) The first one, **AppComponse**, will host the web application for the Add-in experience when creating an email.

d) The second one, **AppRead**, will host the web application for the Add-in experience when reading an email.

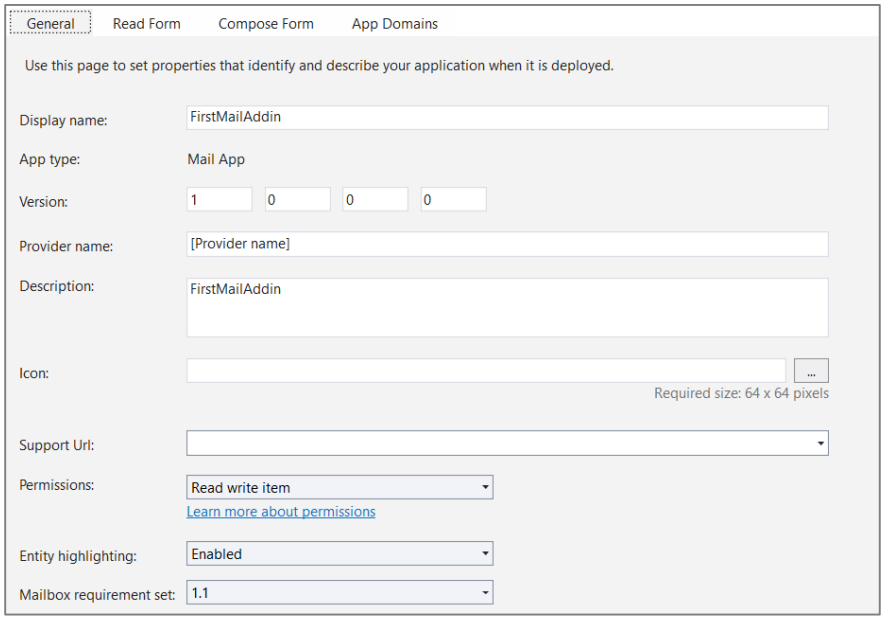
e) The last few folders, **Content**, **Images** and **Scripts** are all the typical supporting files in any web application.

7. Now it's time to open and inspect the Outlook Add-in manifest file.

a) Double click on **FirstMailAddinManifest** node within the Solution Explorer to open the add-in manifest designer.

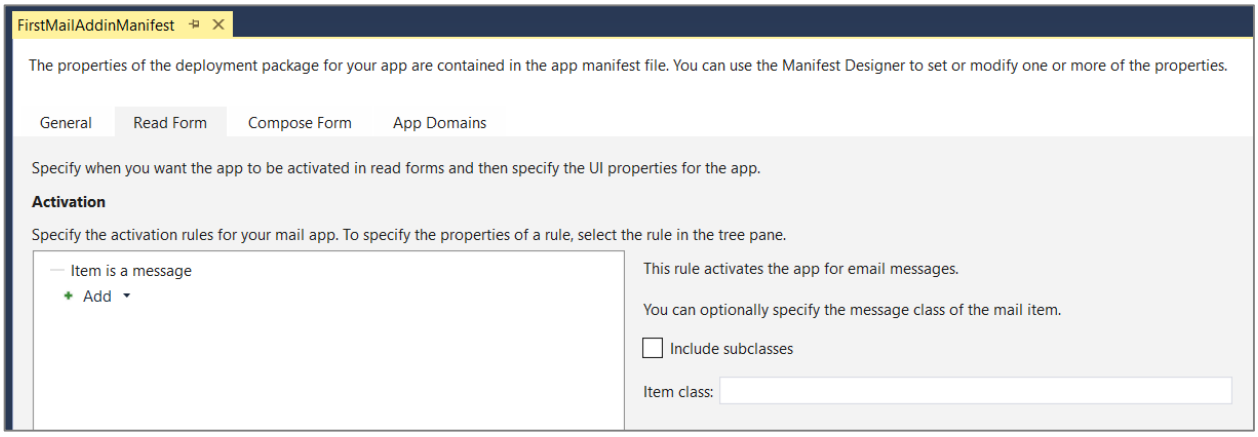


b) Look at the **General** tab. This is where you can change the name, version, description and provider of the Add-in. In addition you can also set the permissions the Add-in requires.

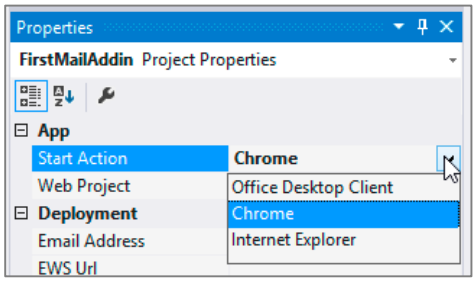


Notice the last option for the Mailbox requirement set. The SDK documentation for Office.js on MSDN will reference a specific requirement set version that a specific feature or capability was added to the API. This is how your application can be developed to support functionality in specific Outlook clients on different platforms as not all clients on all platforms may support the latest features immediately.

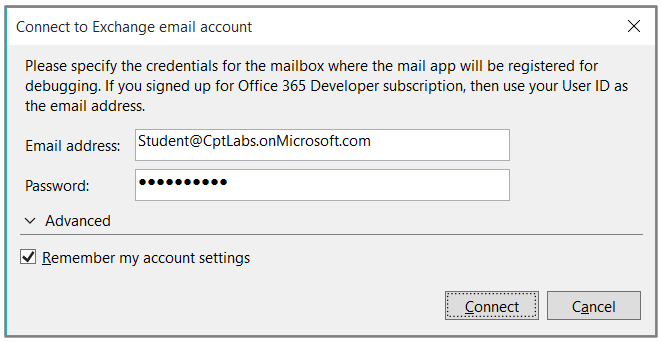
8. Now it's time to set the activation rules for this Mail add-in. Select the **Read Form** tab in the add-in manifest designer. This is where you can make customizations to the read form for the add-in. As you can see, the **Activation** section allows you to specify under which conditions the Add-in will be available. By default it already has the **Item is a message** rule entered.



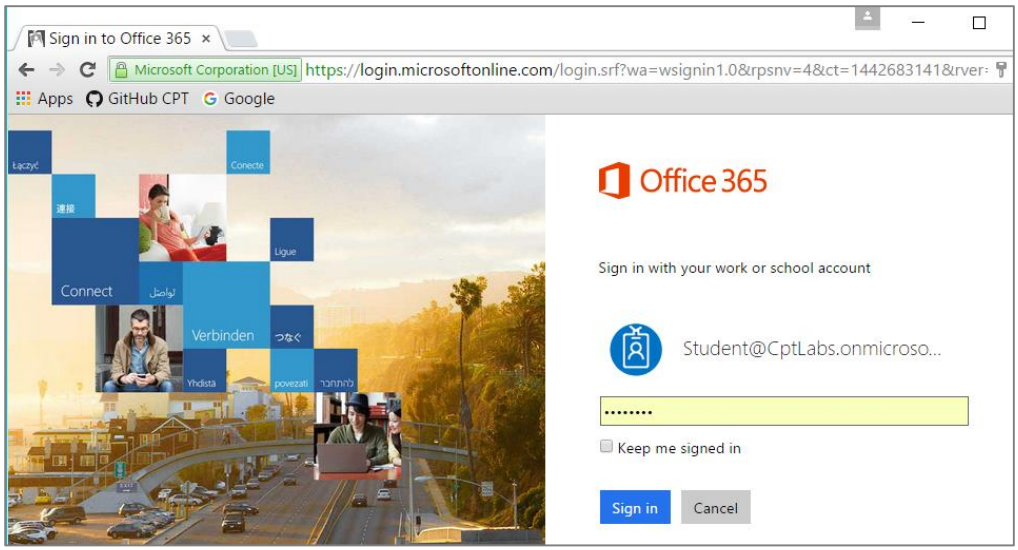
9. Now it's time to deploy the Outlook add-in to see how it works within the browser.
   a) Select the **FirstMailAddin** project within the Solution Explorer tool window and then locate the **Start Action** property in the project's property sheet. Select one of the browser options such as Chrome or Internet Explorer.



b) Press **{F5}** to start the project with the Visual Studio debugger. When the debugging session starts, Visual Studio will prompt you to login using your Office 365 credentials. Enter the credential for your Office 365 developer account and make sure to check the checkbox with the option **Remember my account settings**. Press the **Connect** button to log in and continue the debugging session.

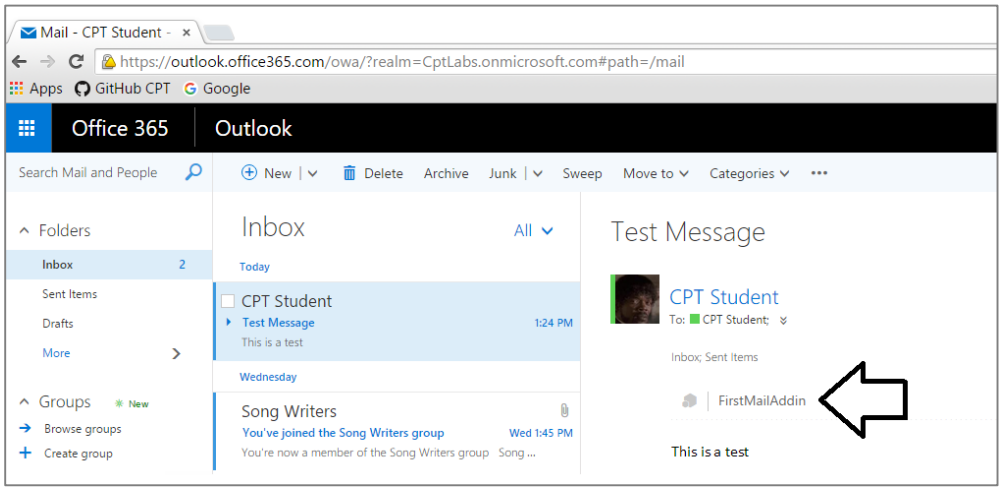


c) At this point you might be prompted a second time for credentials within the browser. If this is the case, go through the login process again entering the credentials for your Office 365 developer account.

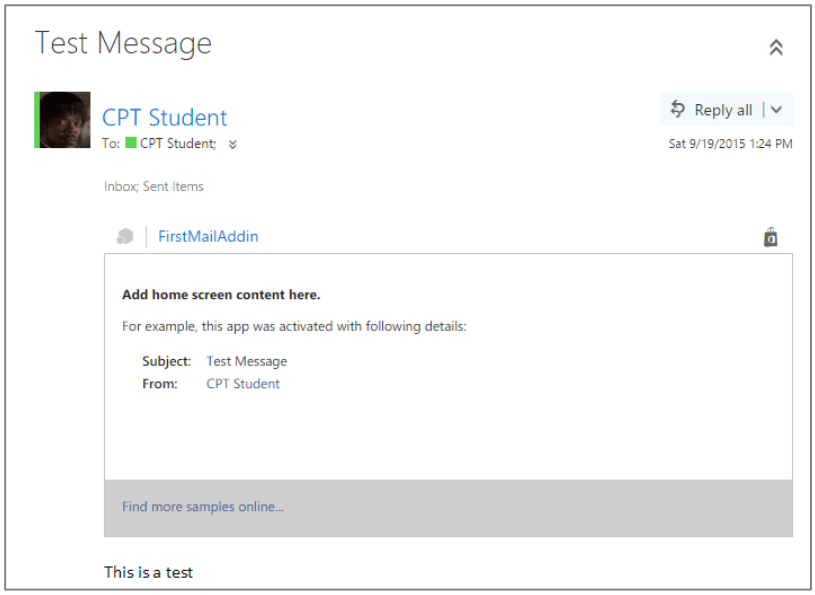


d) After logging in, you should be directed to the web-based Outlook inbox associated with your Office 365 developer account. In order to test and debug the **AppRead** experience for this add-in, you must select an email message from your inbox. If you don't have any messages, send one to yourself for testing purposes.

e) Once you select a message, the add-in will appear as a link button just below the header and above the body of the message.

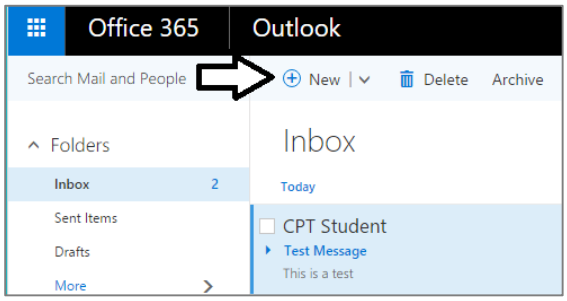


f) Click on the **FirstMailAddin** link button to expand the **AppRead** experience for this add-in.

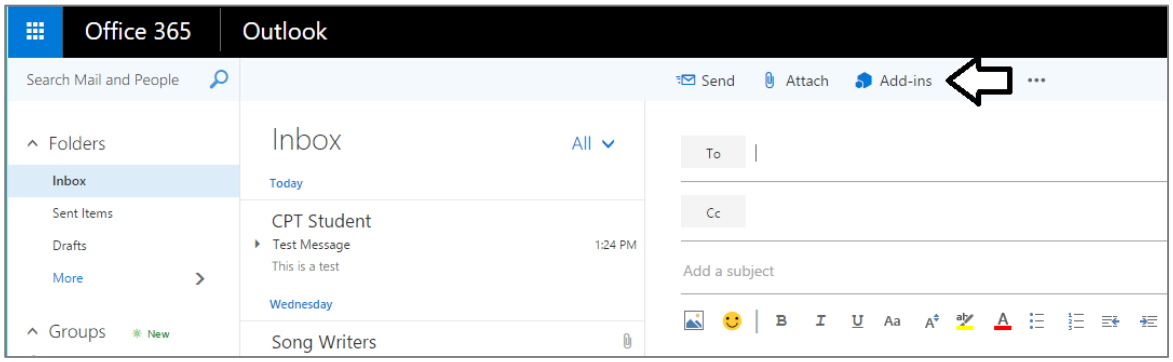


10. Now it's time to test out the **AppCompose** experience for this add-in by creating a new message.

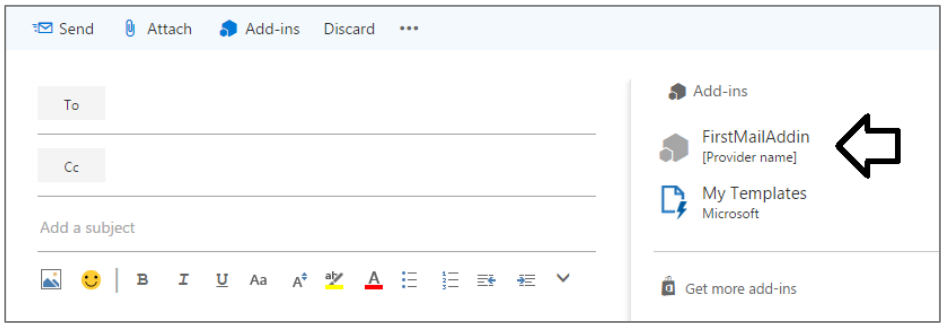
a) Create a new message by click the **New** button in the Outlook toolbar.



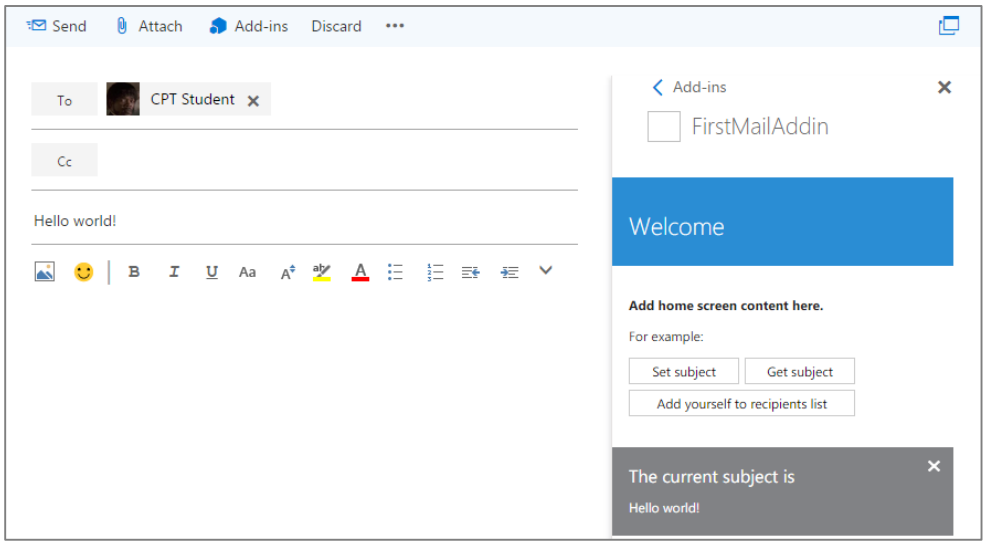
b) Add-ins which provide a compose form are shown in a task pane. However, you must click the **Add-ins** button to get the task pane to appear.



c) You should now see a task pane showing add-in and the **FirstMailAddin** should be on the list. Click the FirstMailAddin button to display the compose form for this add-in.



d) The compose form has three buttons on it. Click the **Set subject** button and notices how it sets the Subject for the current email message. Click the other two buttons to test their behavior as well.

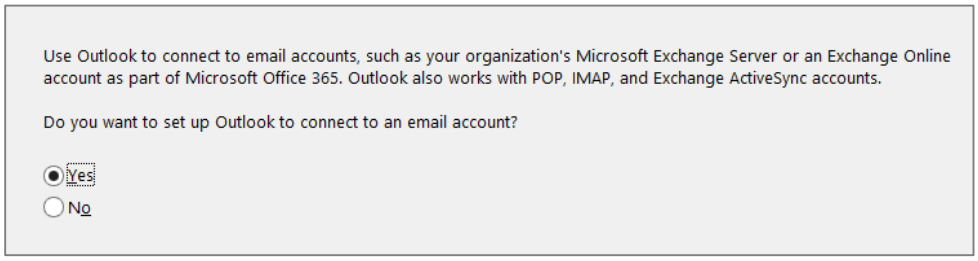


11. Close the browser windows and return to Visual Studio and terminate the debugging session.

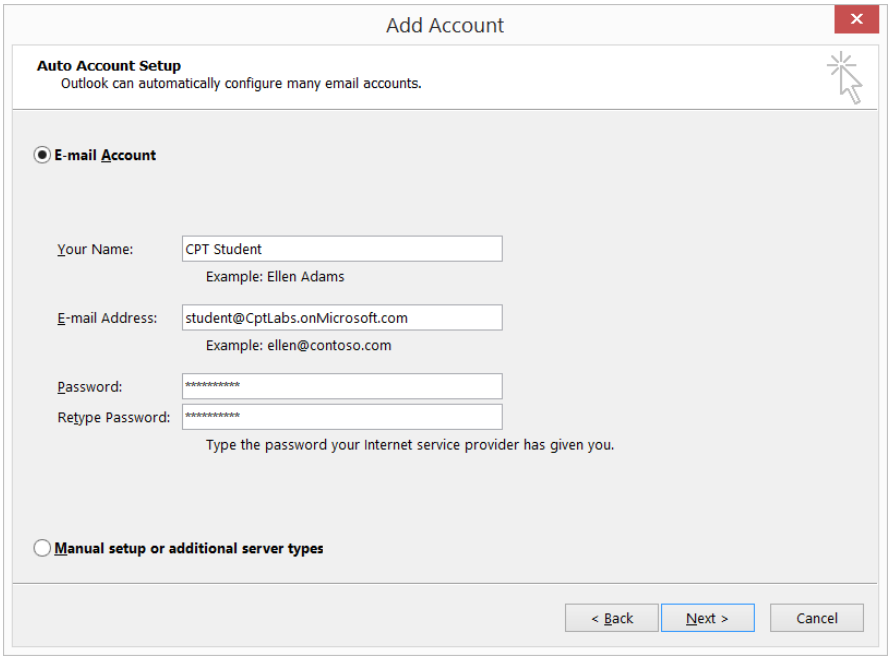
Now it's time to deploy the Outlook Add-in to the local Outlook client.

12. Before you test your add-in with the Outlook desktop client in the Visual Studio debugger, you must ensure that the Outlook desktop client has already been configured to use the inbox associated with your Office 365 developer account.

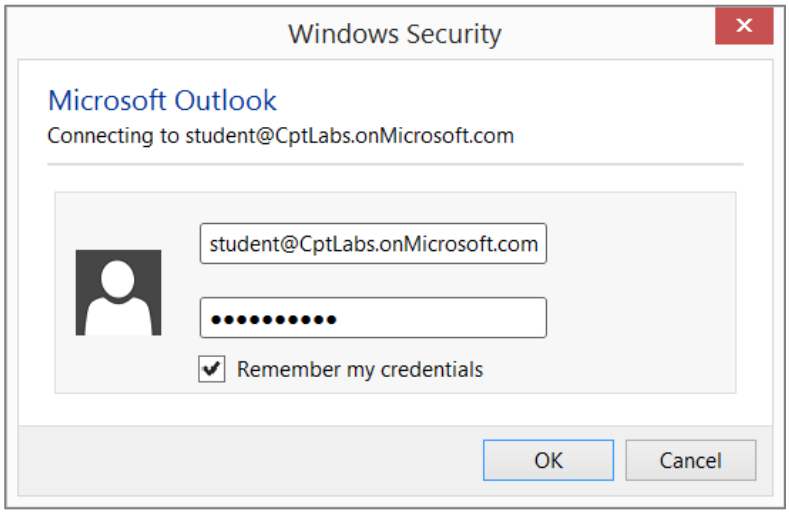
a) Start up the Outlook desktop client and see whether you are prompted to connect to an email account. If the Outlook desktop client has already been configured to open the inbox of your Office 365 developer account, you can move ahead to the next step. If not, select **Yes** and click next to configure the settings for the Outlook desktop client.



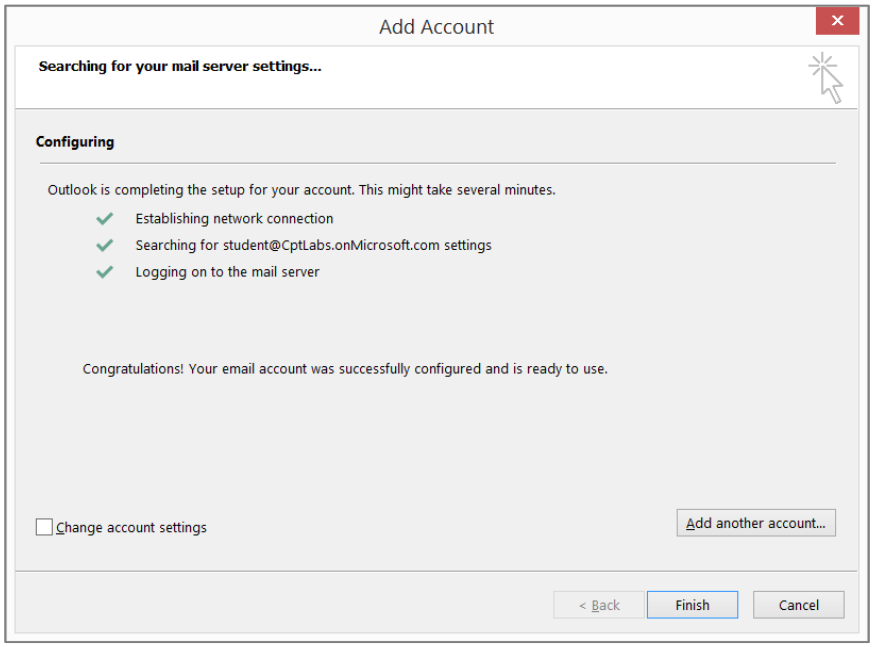
b) On the **Auto Account Setup** page, enter the email address and password for your Office 365 developer account and click **Next**.



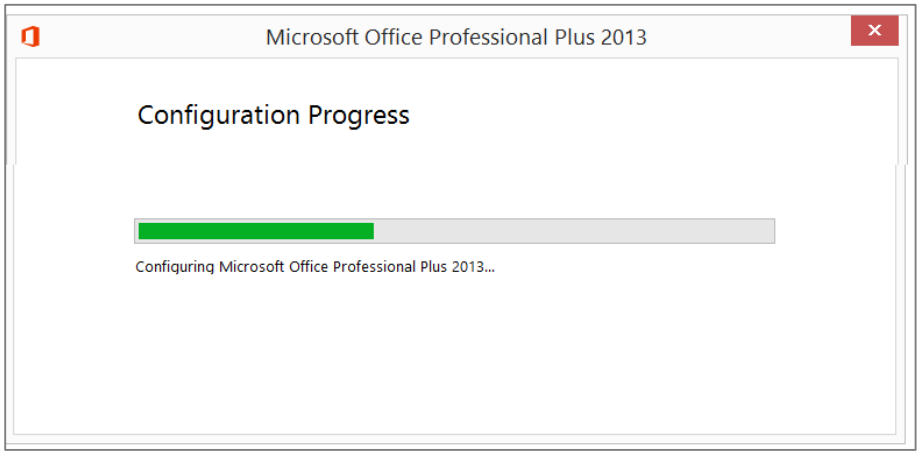
c) At this point, the Outlook desktop client will attempt to open the inbox and will prompt you again for the email address and password of your Office 365 developer account. Enter the email address and password and check the option to **Remember my credentials**. Click **OK** to log in and see your inbox.



d) At this point you should see a dialog indicating that you have successfully configured your email settings. Click **Finish**.



e) Wait for the Outlook client to start and go through the Configuration Process for setting up a new mailbox. Be patient because this might take several minutes to complete.



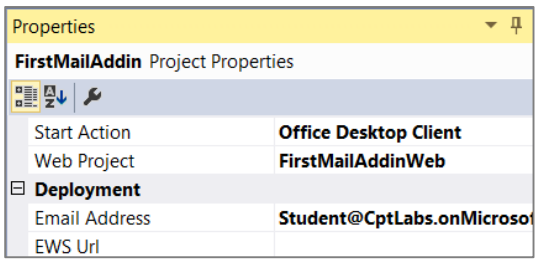
f) Once Outlook has completed the configuration process, close the Outlook desktop client.

13. Now it's time to test the add-in in the Outlook desktop client.

a) Return to Visual Studio and select the top-level project node in the Solution Explorer

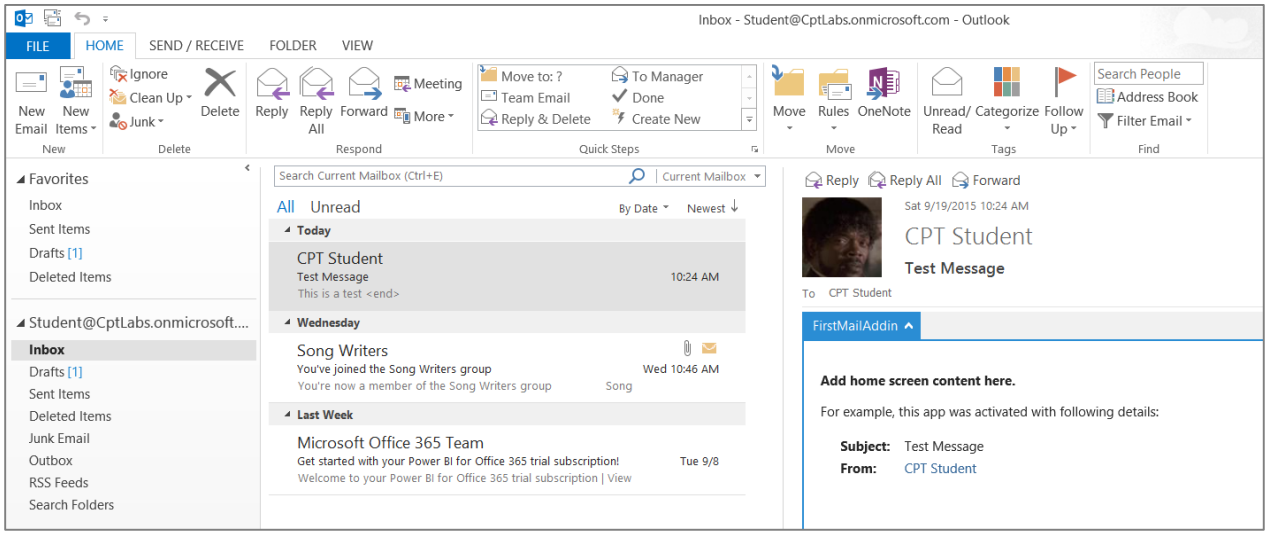
b) Inspect the project's property sheet and locate the **Start Action** property.

c) Set the **Start Action** property to **Office Desktop Client**.

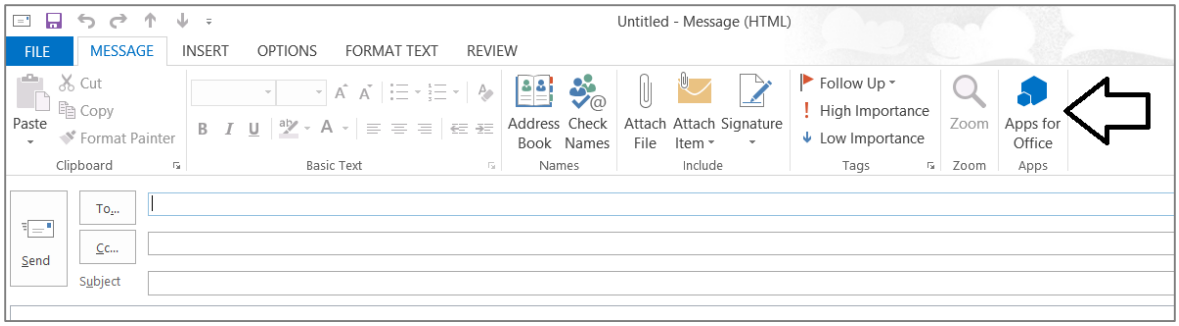


d) Press **{F5}** to begin another debugging session. You should observe that the Visual Studio debugger launches the Outlook desktop client.

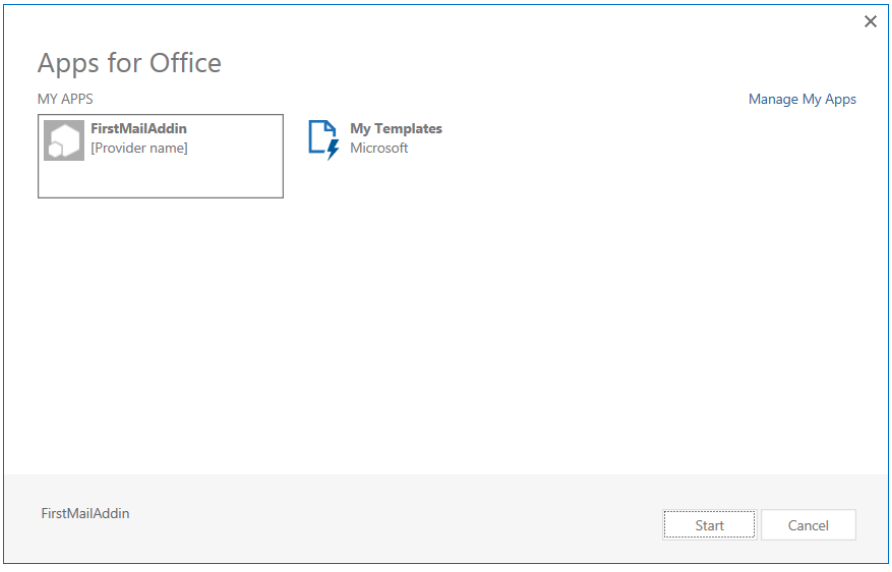
e) When the Outlook desktop client opens, select a message. You should see the link button for FirstMailAddin just under the message header just like it appears in the Outlook web client.



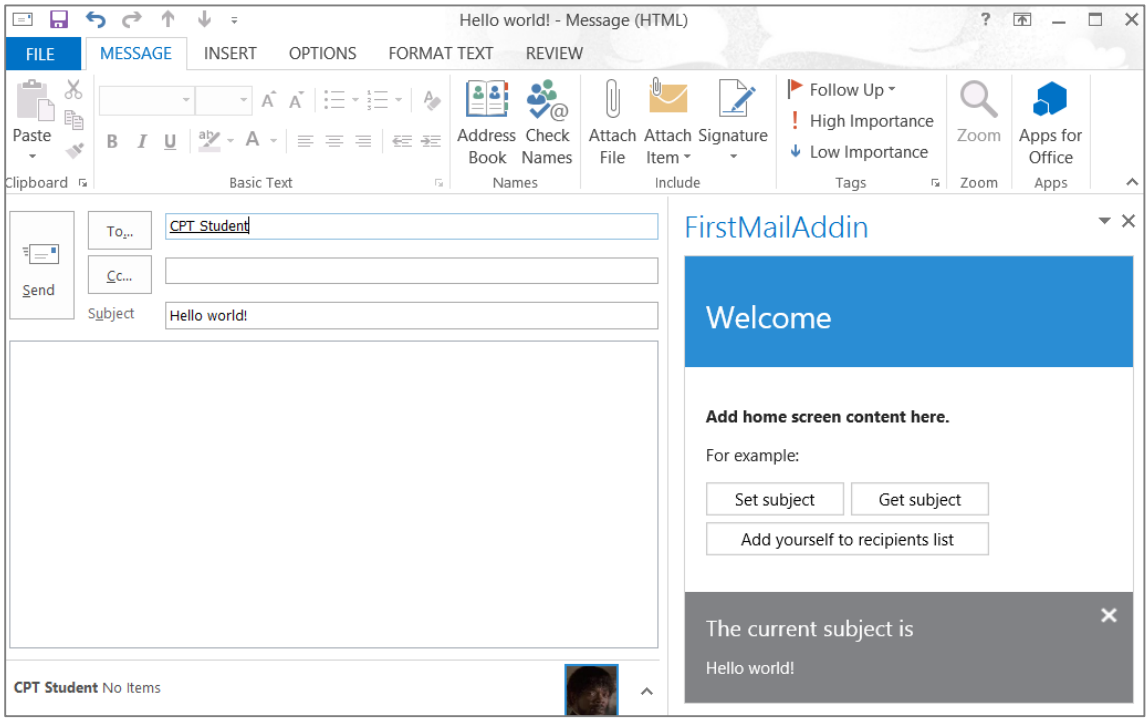
14. Now it's time to check out the compose form.
    a) Create a new message in the Outlook desktop client.
    b) On the toolbar of the new message form, locate and click the Apps for Office button to show the list of add-in that provide compose forms.



c) On the Apps for Office dialog, double click the **FirstMailAddin** button to load its compose form.



d) Test out the functionality of the compose form. It should behave just as it did in the Outlook web client.



You have now completed this lab in which you create a very simple Outlook add-in and tested it in both the Outlook web client and the Outlook desktop client.