|  |
| --- |
| Microsoft SharePoint 2013 - Hands-on Lab |
| Developing Document-centric Apps for Office 2013 |
| Verified Against Build 15.0.4420.1017 |

|  |
| --- |
| Microsoft  Version 1.0  August 14, 2012 |

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links are provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Copyright © 2012 Microsoft Corporation. All rights reserved.

Microsoft and SharePoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

# Introduction

## Estimated time to complete this lab

60 minutes

## Objectives

After completing this lab, you will be able to:

* Create Document-centric Apps for Office with Visual Studio 2012
* Learn to read and modify document content using the Office JavaScript API
* Call out across the Internet to the Bing Translation service
* Learn to create document bindings and register document event handlers

## Virtual Machine Technology

The computers in this lab are virtual machines that are implemented using Microsoft Hyper-V. Before starting each virtual machine, ensure you apply the **Start-Lab** snapshot. When you have started a virtual machine, log on by pressing **CTRL+ALT+END** and supply the credentials listed in the lab instructions.

## Computers in this lab

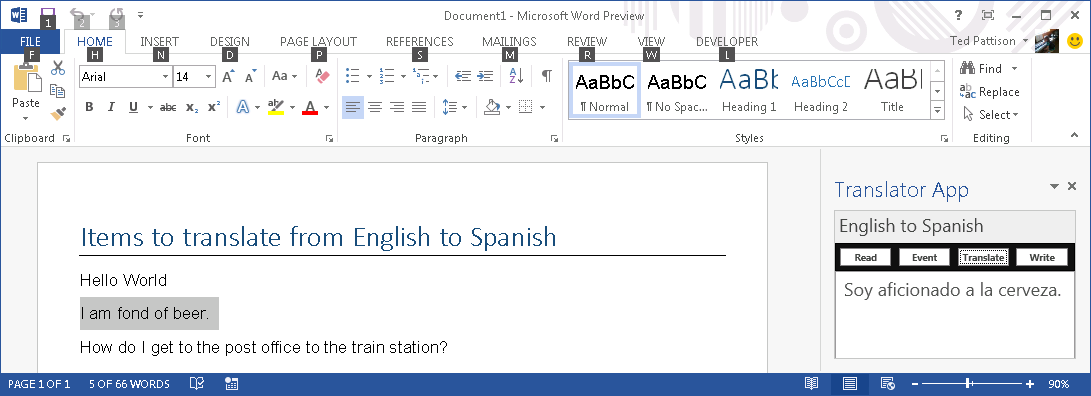
This lab uses virtual machines as described in the following table. Before you begin the lab, you must start the virtual machines and then log on to the computers.

|  |  |
| --- | --- |
| Virtual Machine | Role |
| {Supplied by Instructor} | Domain Controller |
| {Supplied by Instructor} | Actual SharePoint environment with Office client and other required software. |

All user accounts in this lab use the password {Supplied by Instructor}.

# Exercise 1: Creating the Translator App

In this exercise you will create a new Apps for Office project in Visual Studio 2012 named the **TranslatorApp**. The user interface of the app you will create will contain four command buttons and a display area. The app itself will give the user the ability to read and translate content from English to Spanish using the online Bing Translation service. The user then has the ability to write the translated content back to the document.

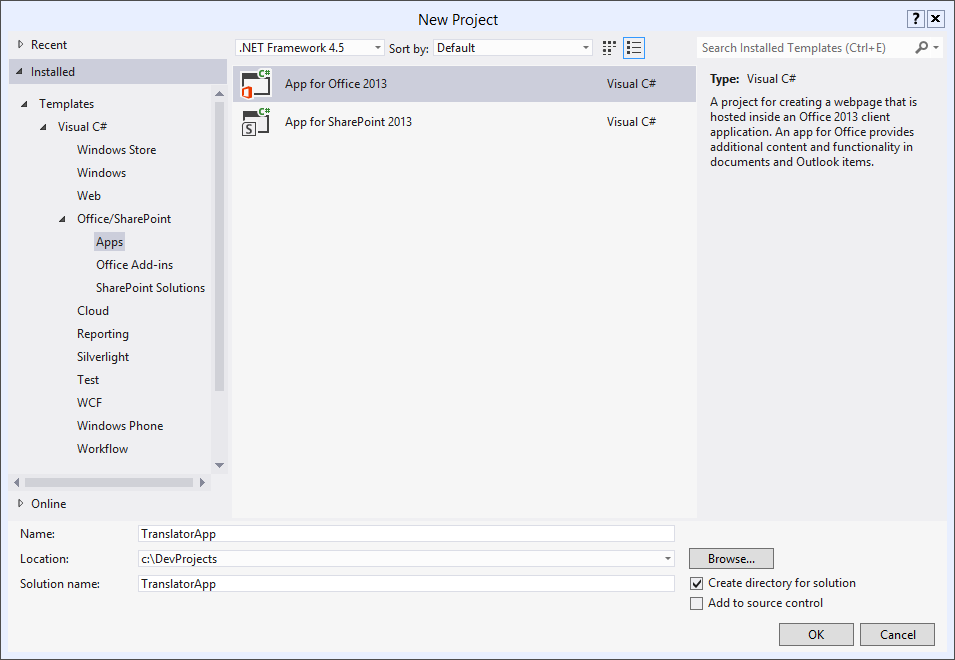


You will start this exercise by creating a new Visual Studio project to create a Microsoft Word task pane app. You will be provided with HTML and CSS you need from starter files so you can quickly copy-and-paste the user interface together. However, you will write the JavaScript code required to bring the app to life.

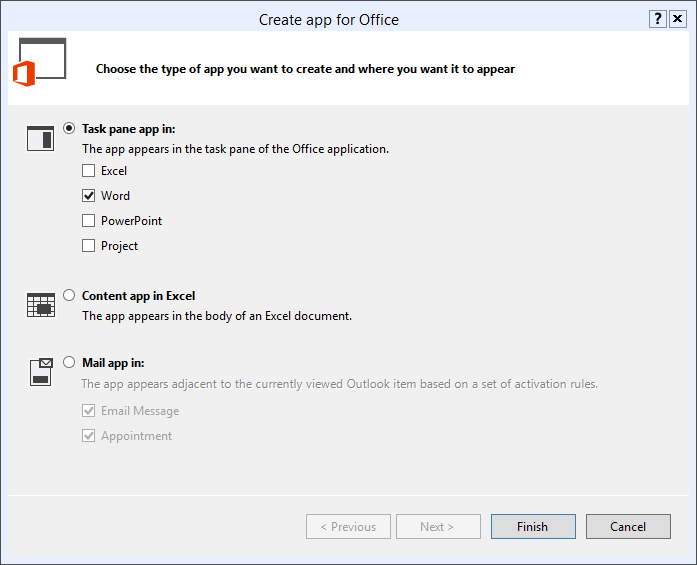
## Step 1 – Create the Translator App

In this Step, you will develop a Word task pane app that reads the selected input in the current document. It allows the user and allows the user to then insert the translated content back into the current document.

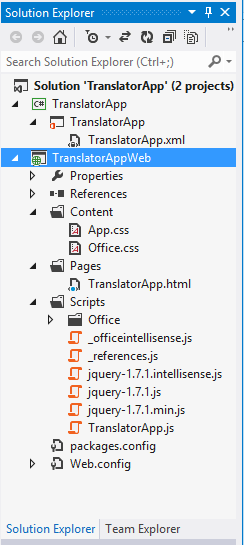
* + - 1. Open Microsoft Visual Studio 2012
      2. From the File Menu select the New Project command. When the **New Project** dialog appears, select the **Apps for Office 2013** project template from the **Office/SharePoint > Apps** template folder as shown below. Name the new project **TranslatorApp** and click **OK** to create the new project.



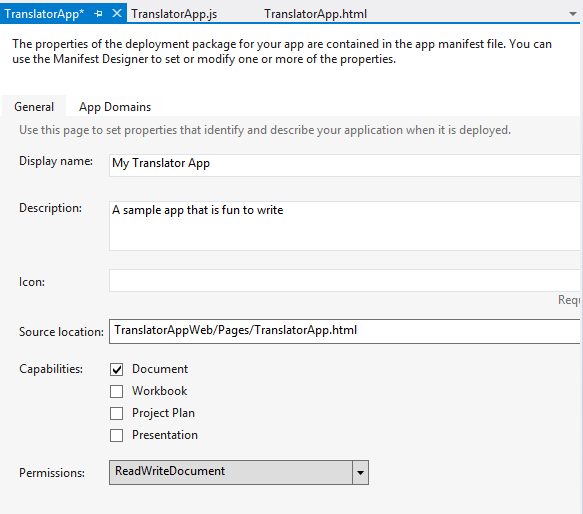
* + - 1. Fill out the next dialog as the one shown below in order to create a new task pan app that is designed to work just with Microsoft Word. Click **Finish** when you are done.



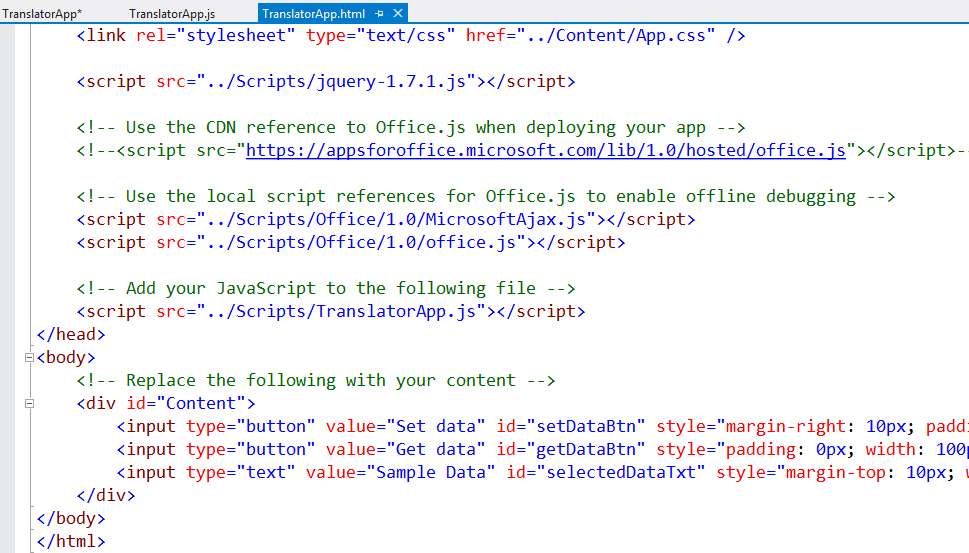
* + - 1. Once the solution has been created, take a moment and examine the two projects inside. The first project named **TranslatorApp** just contains an xml file for the app manifest named **TranslatorApp.xml**. The other project is the "remote web" project which is named **TranslatorAppWeb**. The remote web project is where you will implement the user interface and JavaScript behavior for this app.



* + - 1. Review the layout of the remote web project named **TranslatorAppWeb**. Examine the structure of folders and files inside the new project. You will notice that a top-level folder structure has been created with folders named **Content**, **Pages** and **Scripts**. You should also point out that the JavaScript files and links have been added to support the jQuery library which means you can begin using the jQuery library without any additional work.
      2. Inside the top project, double-click on **TranslatorApp.xml** to open the app manifest inside the special designer provided by Visual Studio 2012. Inside the app manifest designer, modify the property values for app **Display** property and **Description** property using custom values.



* + - 1. Save and close **TranslatorApp.xml**.
      2. As you implement this first app over the next few steps, you will mainly be making modifications to three primary files that are named as follows:
         1. **TranslatorApp.html** - used to link css/js files and provide HTML for the app's UI.
         2. **App.css** - used to layout and style HTML elements of the app's UI.
         3. **TranslatorApp.js** - used to implement the app's behavior and functionality.
      3. Open the HTML source file **TranslatorApp.html**. Note that you will find some pre-existing HTML that is supplied by Visual Studio. Learn how to switch between the three modes of the HTML editor which are **Source View**, **Design View** and **Split View**.



* + - 1. Now you will create a simple user interface using HTML. Start by deleting all the content inside the body section of **TranslatorApp.html**. Make sure you leave everything in the head section just as it is.
      2. Add the following HTML code shown in the code listing below or copy-and-paste the code from the **body\_ TranslatorApp.html.txt file** located in the **StarterFiles** folder.

<body>

<h2>English to Spanish</h2>

<div id="toolbar">

<input type="button" id="cmdReadSelection" value="Read" />

<input type="button" id="cmdRegisterEventHandler" value="Event" />

<input type="button" id="cmdTranslate" value="Translate" />

<input type="button" id="cmdWriteSelection" value="Write" />

</div>

<div id="display" />

</body>

* + - 1. Open the source CSS file named **App.css**. Delete the existing content and copy-and-paste the following CSS code from **App.css.txt** located in the **StarterFiles** folder.

body {

padding: 0px;

background-color: #eee;

}

h2 {

margin: 0px;

padding: 4px;

height: 32px;

box-sizing: border-box;

color: #444;

font-size: 1.5em;

}

#toolbar {

padding: 2px;

background-color: #111;

border: 1px solid black;

}

#toolbar input[type="button"] {

margin: 2px;

width: 22%;

font-size: 0.8em;

font-weight: bold;

padding: 1px;

height: 18px;

background-color: white;

border: 1px solid grey;

}

#display {

padding: 8px;

min-height: 80px;

font-size: 1.5em;

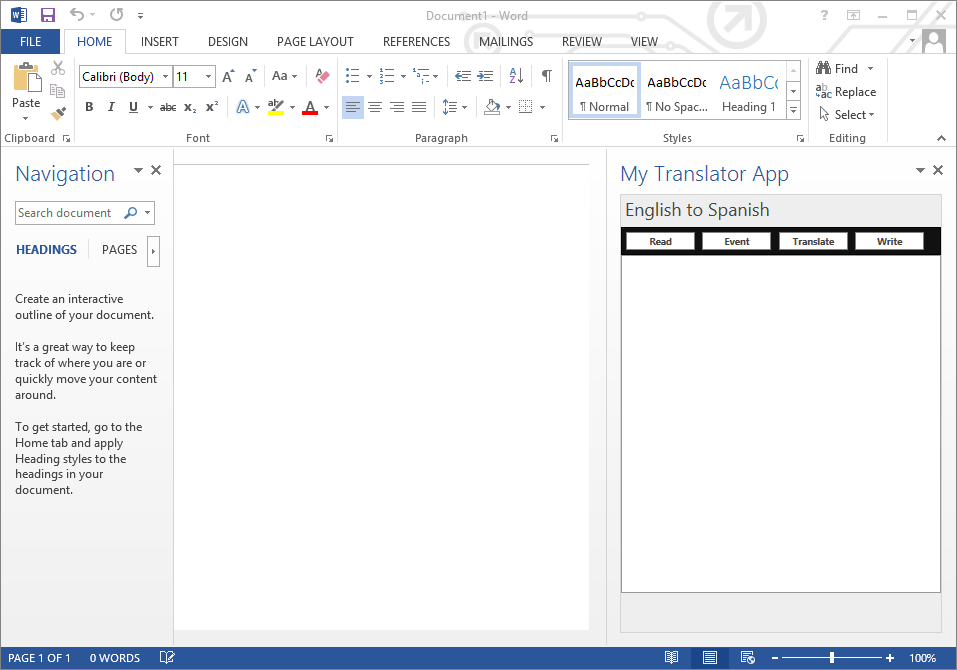
border: 1px solid #999;

background-color: #fff;

min-height: 320px;

}

* + - 1. Before writing any JavaScript code you should test the app in debug mode to ensure the HTML and CSS added over the last few steps results in a good-looking user interface. Press the **{F5}** key in Visual Studio to begin a debugging session. The debugging session should start up Microsoft Word and display the app as shown.



* + - 1. Open the JavaScript source file named **TranslatorApp.js** and examine the code that has been added automatically by Visual Studio. Delete the code from this file and replace it with this following generic starting point. You can optionally copy-and-paste the contents from **starter\_TranslationApp.js.txt** in the **StarterFiles** folder.

**Office.initialize = function (reason) {**

**// Initialize variables using Office JavaScript API**

**// Add handler function for jQuery document ready event**

**$(function () {**

**// UI initialization code goes here**

**});**

**}**

* + - 1. Start by adding some code inside the **Office.initialize** function to initialize a variable that will be used to track an object in the Office JavaScript API. Do this by adding a top-level variable named **officeDoc** and initialize the variable inside **Office.intialize**.

**var officeDoc;**

Office.initialize = function (reason) {

**// Initialize variables using Office JavaScript API**

**officeDoc = Office.context.document;**

// Add handler function for jQuery document ready event

$(function () {

// UI initialization code goes here

});

}

* + - 1. In this step you will add code to initialize the app's UI by adding code to register and implement event handlers for each of the four buttons. You can update the code by hand using this listing below or you can copy-and-paste the contents of the helper file named **AddHandlers\_TranslatorApp.js.txt** in the **StarterFiles** folder.

var officeDoc;

Office.initialize = function (reason) {

// Initialize variables using Office JavaScript API

officeDoc = Office.context.document;

// Add handler function for jQuery document ready event

$(function () {

**// UI initialization code goes here**

**$("#cmdReadSelection").click(onReadSelection);**

**$("#cmdRegisterEventHandler").click(onRegisterEventHandler);**

**$("#cmdTranslate").click(onTranslate);**

**$("#cmdWriteSelection").click(onWriteSelection);**

});

}

**function onReadSelection() {**

**}**

**function onRegisterEventHandler() {**

**}**

**function onTranslate() {**

**}**

**function onWriteSelection() {**

**}**

* + - 1. Write the implementation of the **onReadSelection** function to read the current selection in the document by calling the **getSelectedDataAsync** method on the Office document referenced by **officeDoc** variable. Pass the literal string "text" as the first parameter to indicate the coercion type. Pass an empty JavaScript object (i.e. **{}** ) as the second parameter to accept the default settings. The third parameter should reference a new callback method named **onReadSelectionComplete** which you should add after the **onReadSelection** function

function onReadSelection() {

officeDoc.getSelectedDataAsync("text", {}, onReadSelectionComplete);

}

function onReadSelectionComplete(asyncResult) {

}

* + - 1. Ensure you have defined the **onReadSelectionComplete** function as taking a single **paramater** named **asyncResult**. Implement **onReadSelectionComplete** by acquiring the value of the selection made available through **asyncResult.value**. Write the value into the HTML element with the id of **display**.

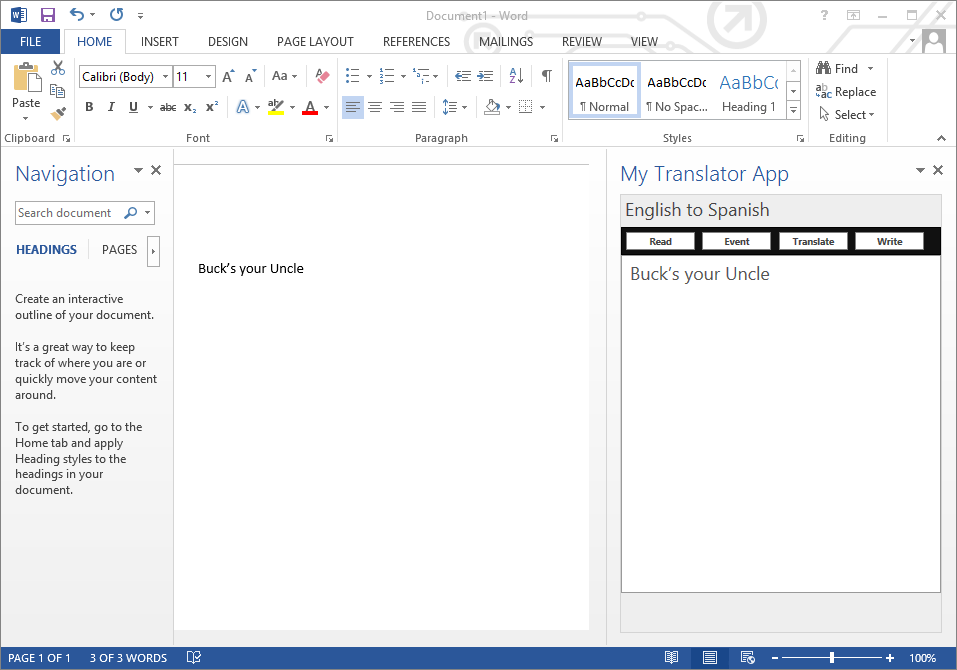
function onReadSelectionComplete(asyncResult) {

var selection = asyncResult.value;

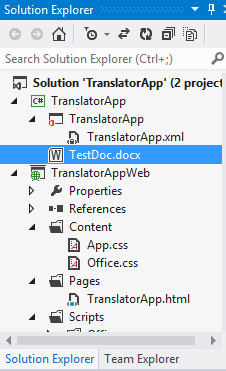
$("#display").text(selection);

}

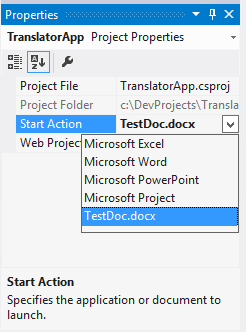
* + - 1. You have added enough functionality where you can test your code . Press the **{F5}** key in Visual Studio 2012 to begin debugging the project. Visual Studio should automatically launch Microsoft Word. You should see the app initialize inside the Word task pane. Type some text into the new document and select it. When you click the **Read** button, your code should read the user's selection and display it inside the app.



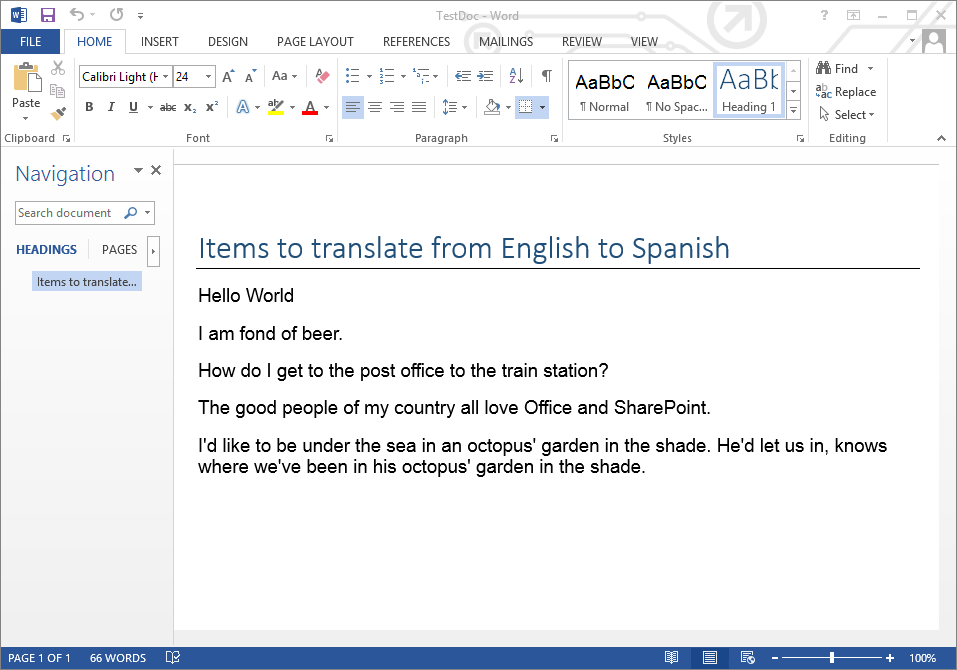
* + - 1. Now it is time to add a test document to make testing the app easier during a debugging session. Locate the Word document named **TestDoc.docx** in the **StarterFiles** folder and add it to the **TranslatorApp** project as a top level file.



* + - 1. Now find the property sheet for the **TranslatorApp** project and locate the project property named **Start Action**. Change the value of the **Start Action** property from **Microsoft Word** to **TestDoc.docx**.



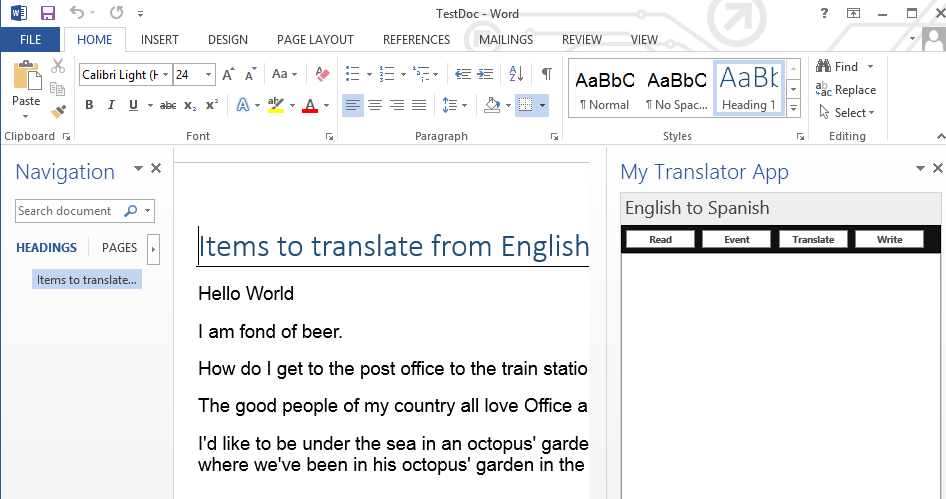
* + - 1. Now press the **{F5}** key to start the app with the new test document. The debugging session should start Microsoft Word with the document shown in the screenshot below. Note that the **TranslatorApp** will not automatically be added at this point. You will fix that in the next step.



* + - 1. With the test document open in a debugging session, use the Apps for Office drop down menu in the **Insert** tab to add the **TranslatorApp** to the document. Once you have added the app to the test document, you then must save this updated version of the test document on top of the document you added into the Visual Studio project structure. Use the **File > Save As** command in Microsoft Word to save current version of the Word document and to overwrite the **TestDoc.docx** file inside the project directory for the **TranslatorApp** project**.**



* + - 1. Once you have saved the updated version of document over the test document named **TestDoc.docx**, close the document and Microsoft Word. Return to Visual Studio and ensure the debugging session has stopped. Now press the **{F5}** key to start a new debugging session. This time the **TranslatorApp** should load automatically along with the test document.



* + - 1. Now it is time to add code to the **Event** command button to register an event handler that fires whenever the user changes the selection. Do this by adding code inside the **onRegisterEventHandler** that calls the **addHandlerAsync** method on the **document** object referenced by the **oficeDoc** variable. For the first parameter, pass the string value of **"documentSelectionChanged"** to bind to the correct event. For the second parameter pass the name of a callback **onSelectionChanged** and also add a function of the same name below.

function onRegisterEventHandler() {

officeDoc.addHandlerAsync("documentSelectionChanged", onSelectionChanged);

}

function onSelectionChanged() {

}

* + - 1. Implement the **onSelectionChanged** callback method with the same code that is used to implement the **onReadSelection** method by calling **getSelectedDataAsync** and passing **onReadSelectionComplete** as the callback function.

function onSelectionChanged() {

officeDoc.getSelectedDataAsync("text", {}, onReadSelectionComplete);

}

* + - 1. Test your work by pressing the **{F5}** key to debug the app. When the app starts up, you should test by clicking the **Event** command button. After clicking this button, you should be able to select different sections of text from the Word document and see that the app automatically displays the content from the selection in the app display area. The key point is that your code has registered an event handler that automatically fires whenever the user changes the current selection.
      2. Now it is time to add code to the **Translate** command button to send the content in the app's display area for translation by the Bing Translation Service from English to Spanish. Do this by first adding code inside the **onTranslate** function that retrieves the text content from the div with the id of **display**. Next, create an string variable named **url** and build the URL required to call the Bing Translation Service. After building the URL, use jQuery syntax to create a script link which uses the URL as the src attribute value and add the script link to the head section of the page using the code shown below or by copying-and-pasting the contents of **translate\_TranslatorApp.js.txt**.

**function onTranslate() {**

**// get value of text inside display element**

**var sourceText = $("#display").text();**

**// create URL to get JSONP script from bing translation service**

**var url = "http://api.microsofttranslator.com/V2/ajax.svc/Translate" +**

**"?oncomplete=onTranslateComplete" + // add callback method name**

**"&appId=5D28780ED5302B3F6F5E85CE7B7872F76735EBAE" +**

**"&from=en" + // from English**

**"&to=es" + // to Spanish**

**"&text=" + sourceText;**

**// create script link element using jQuery sytax**

**var script = $("<script>", { "src": url });**

**// add script to end of head which will run script and execute callback**

**$("head").append(script);**

**}**

* + - 1. At the bottom of **TranlatorApp.js**, place your cursor after the **onTranslate** function and add a new callback function named **onTranslateComplete**. Note that the callback function named **onTranslateComplete** is passed by name using a query string parameter in the URL that was built in the previous step. The key point is that this callback method will be called automatically when the Bing Translation service returns a translated result. Implement the **onTranslateComplete** function as shown below.

function onTranslateComplete(translatedText) {

if (translatedText) {

$("#display").text(translatedText);

}

}

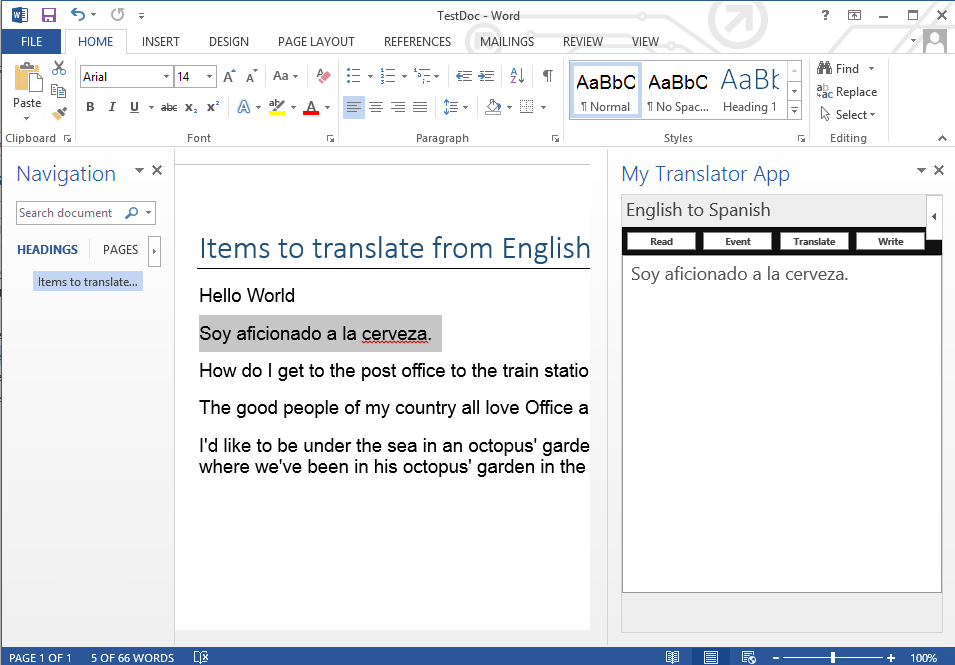
* + - 1. Now finish the code by implementing the **onWriteSelection** function by writing the content of the of the div with the id of display back to the document selection by calling the **setSelectedDataAsync** method of the document object.

function onWriteSelection() {

officeDoc.setSelectedDataAsync($("#display").text() + "\n");

}

* + - 1. Test your work. Press the **{F5}** key to debug the app. When the app starts up, you should test by pressing the **Event** command button. After that you should be able to select different sections of text from the text document and see that the app automatically displays the content from the selection in the app display area. Then you should be able to press the **Translate** button and see the English content in the app's display area replaced by the same content translated to Spanish. After that you should be able to press the **Write** button and see the translated Spanish content overwrite the original English content in the current selection.



* + - 1. You have now completed all the steps for this exercise

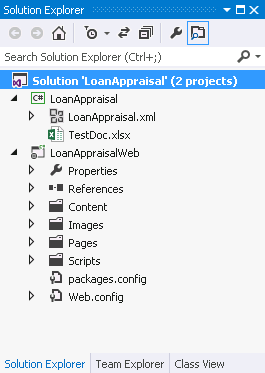
# Exercise 2: Working with Excel and Document Bindings

In this exercise you will work with a pre-created app for Office project named **LoanAppraisal**. The user interface and much of the functionality for this app has already been written. You job is to add extra JavaScript code to create bindings and to move data back and forth between the app and an Excel spreadsheet.

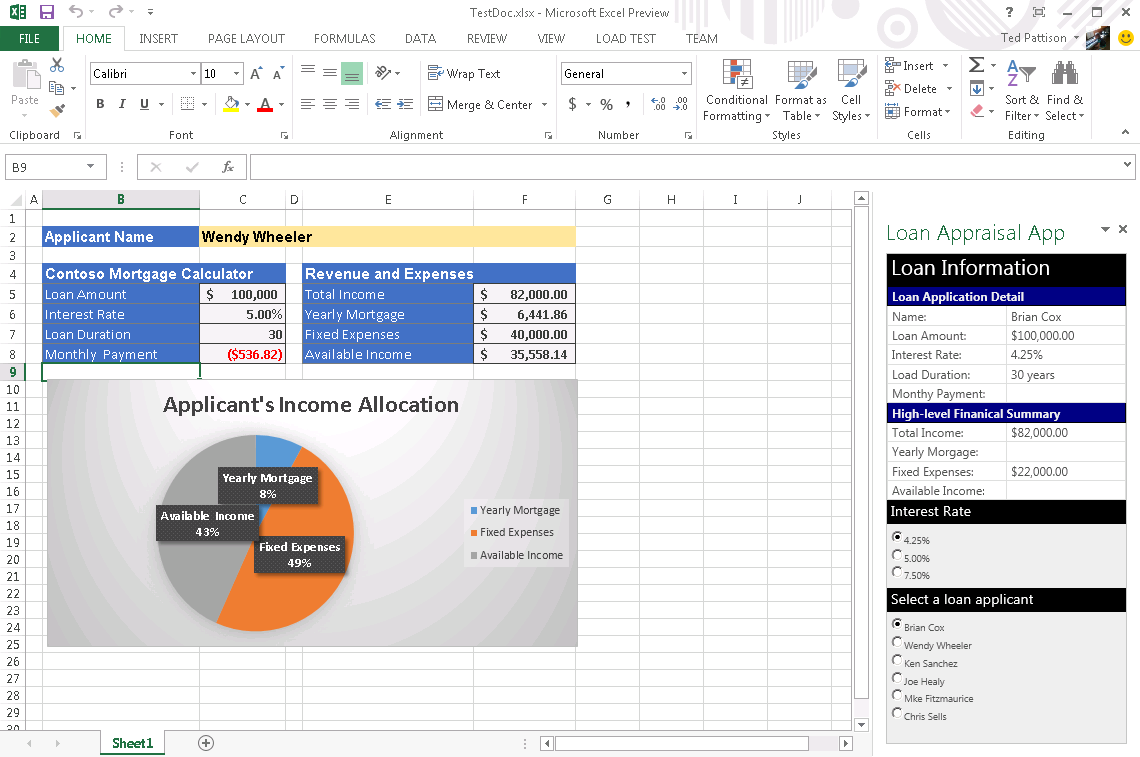
## Step 1 – Open and test the Loan Appraisal App

In this Step, you will open and beginning working with an Excel task pane app project.

1. Open Microsoft Visual Studio 2012
2. Open the pre-existing Visual Studio solution named **LoanAppraisal.sln** which is located inside the **LoanAppraisal** folder inside the **StarterFiles** folder for this lab.
3. Take a brief moment to examine the two projects in this solution. The top project is named **LoanAppraisal** which contains the app manifest and a test document named **TestDoc.xslx**. Note that the **Start Action** for this project has been configured to use **TestDoc.xslx**.



1. Now it's time to see what the app looks like and how it behaves. Press the **{F5}** and launch the app in a debugging session. Note that the app allows the user to select a different interest rate and a different loan applicant in the task pane. When you change the interest rate or the current applicant, the app updates the user interface in it display area.



1. With the app in its current state, it can update its own user interface when the user changes the interest rate or the applicant. However, the app has no code to communicate with the Excel spreadsheet. Over the next few steps you will add the code required to create bindings from the app to named ranges inside the spreadsheet which will allow your code to pass data back and forth between the app and the spreadsheet.
2. Inside the **LoanAppraisalWeb** project, open the JavaScript source file named **LoanAppraisal.js** in the **Scripts** folder. Scroll to the bottom of the file and locate the following comment and a set of empty functions that you will implement in this lab.

**// all work for this lab should be done below this comment**

**function createBindings() {}**

**function onAllBindingCreated(asyncResult) {}**

**function updateBindingsToDocument() {}**

**function onBindingUpdated(asncResult){ }**

**function updateBindingsFromDocument() {}**

**function onBindingReadFromDocument(asyncResult) {}**

1. Implement the **createBindings** function to create bindings to cells which have already been defined in the Excel spreadsheet using named ranges. Note that spreadsheet already has named ranged defined with names such as **Applicant\_Name** and **Loan\_Amount.** Your code must create a binding with a unique id for each of these named ranges**.** You can either type the code in the following listing by hand or copy-and-paste the code from **createBindings\_LoanAppraisal.js.txt** inside the **StarterFiles** folder.

**function createBindings() {**

**bindings.addFromNamedItemAsync("Sheet1!Applicant\_Name", "text",**

**{ id: "applicant\_name" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Loan\_Amount", "text",**

**{ id: "loan\_amount" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Interest\_Rate", "text",**

**{ id: "interest\_rate" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Loan\_Duration", "text",**

**{ id: "loan\_duration" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Monthly\_\_Payment", "text",**

**{ id: "monthly\_payment" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Total\_Income", "text",**

**{ id: "total\_income" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Yearly\_Mortgage", "text",**

**{ id: "yearly\_mortgage" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Fixed\_Expenses", "text",**

**{ id: "fixed\_expenses" }, function(){});**

**bindings.addFromNamedItemAsync("Sheet1!Available\_Income", "text",**

**{ id: "available\_income" }, onAllBindingCreated);**

**}**

1. Note that the **onAllBindingCreated** method is passed as a callback method in the final line of code in the **createBindings** function. Implement the **onAllBindingCreated** function to call **updateBindingsToDocument**.

function onAllBindingCreated(asyncResult) {

updateBindingsToDocument();

}

1. Implement the **updateBindingsToDocument** function to push input values from the app to the appropriate cells in the spreadsheet. Start by retrieving the bindings by name using the **Office.select** function and then calling **setDataAsync** on each named binding to transfer the appropriate value. You can either type the code below of copy-and-paste from **updateBindingsToDocument\_LoanAppraisal.js.txt**.

**function updateBindingsToDocument() {**

**Office.select("bindings#applicant\_name")**

**.setDataAsync(currentApplicant.name, function () { });**

**Office.select("bindings#loan\_amount")**

**.setDataAsync(currentApplicant.loan\_amount, function () { });**

**Office.select("bindings#interest\_rate")**

**.setDataAsync(currentRate, function () { });**

**Office.select("bindings#loan\_duration")**

**.setDataAsync(currentApplicant.loan\_duration, function () { });**

**Office.select("bindings#total\_income")**

**.setDataAsync(currentApplicant.total\_income, function () { });**

**Office.select("bindings#fixed\_expenses")**

**.setDataAsync(currentApplicant.fixed\_expenses, onBindingUpdated);**

**}**

1. Implement **onBindingUpdated** to call **updateBindingsFromDocument.**

**function onBindingUpdated(asncResult){**

**updateBindingsFromDocument();**

**}**

1. Implement **updateBindingsFromDocument** to retrieve the output from the three calculated cells though the bindings. In particular, retrieve output from the bindings the ids of **monthly\_payment**, **yearly\_mortgage** and **available\_income**. To retrieve the values from the spreadsheet call the **getDataAsync** method with parameters to pass the id as the **asyncContext** parameter and to keep the values in their existing Excel format. For the last parameter pass the name of the callback function named **onBindingReadFromDocument.** You can either type the code listed below or copy-and-paste it from **updateBindingsFromDocument\_LoanAppraisal.js.txt**.

**function updateBindingsFromDocument() {**

**Office.select("bindings#monthly\_payment")**

**.getDataAsync({ asyncContext: "monthly\_payment",**

**valueFormat: Office.ValueFormat.Formatted },**

**onBindingReadFromDocument);**

**Office.select("bindings#yearly\_mortgage")**

**.getDataAsync({ asyncContext: "yearly\_mortgage",**

**valueFormat: Office.ValueFormat.Formatted },**

**onBindingReadFromDocument);**

**Office.select("bindings#available\_income")**

**.getDataAsync({ asyncContext: "available\_income",**

**valueFormat: Office.ValueFormat.Formatted },**

**onBindingReadFromDocument);**

**}**

1. Implement the **onBindingReadFromDocument** function to write the value from the cell in the associated Excel spreadsheet the correct table cell using the proper id.

**function onBindingReadFromDocument(asyncResult) {**

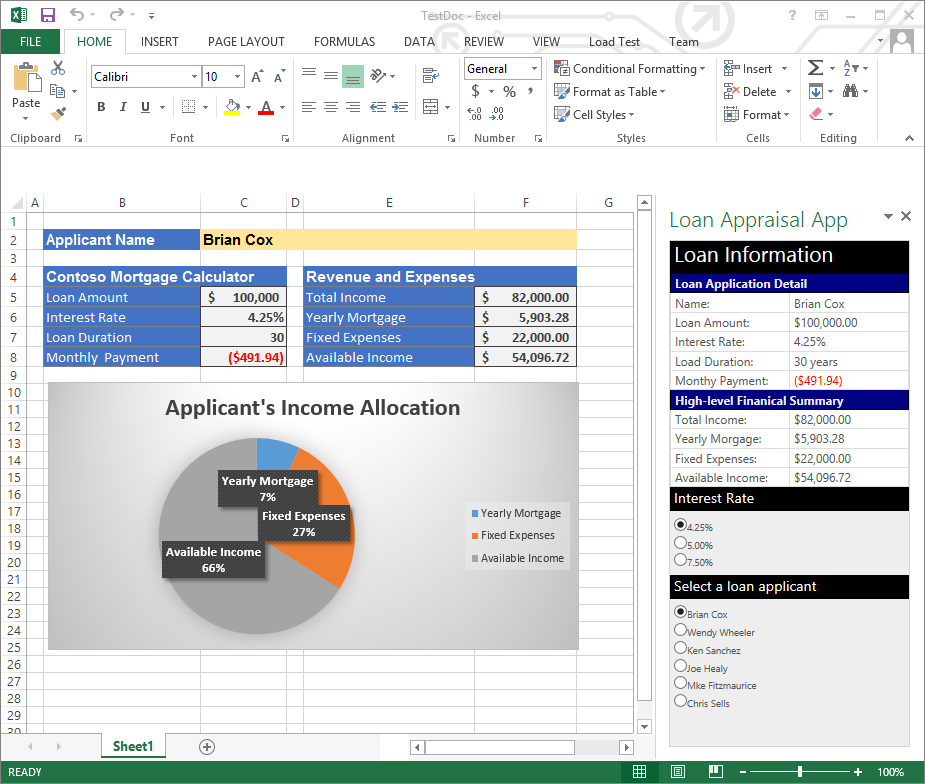
**var value = asyncResult.value;**

**var targetDiv = "#" + asyncResult.asyncContext;**

**$(targetDiv).text(value);**

**}**

1. Test the app. You should be able to run it and see that the spreadsheet has been integrated into the data flow of the application. When you change to a different interest rate or applicant, the new values should be sent to the spreadsheet. After the spreadsheet has updated the cells with calculated values for the monthly payment and year mortgage, these formatted values should then be transferred back to the app.



1. You are now done with this lab.