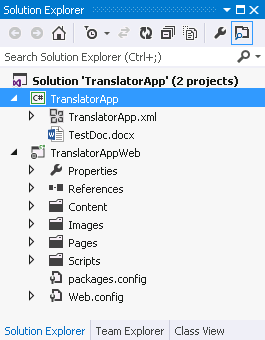
# Demo 1: Reading and Writing to a Document

In this Demo, you open and demo a pre-existing app known as the **Translator App**. This app will read the selected input from the current document and then send it off to the Bing translation service. The user is then able to insert the translated results back into the current document.

## Step 1 – Open and Run the Translator App

In this Step, you will demo 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 **Open Project** command ad open the solution named **TranslatorApp.sln** located inside the **Demo 1 TranslatorApp** folder for this demo.
      3. Once the project has opened, take a moment to show students 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**.



* + - 1. Point out to students that there is a test document named **TestDoc.docx** which has been configured as the **Start Action** of the top project named **TranslatorApp**. The key point is that each document this document will be used each time you press the **{F5}** key to start a debugging session.
      2. Open **TranslatorApp.html** inside the **Pages** folder and show students how the user interface for the app has been created using a simple set of HTML elements in the body section.

<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" />

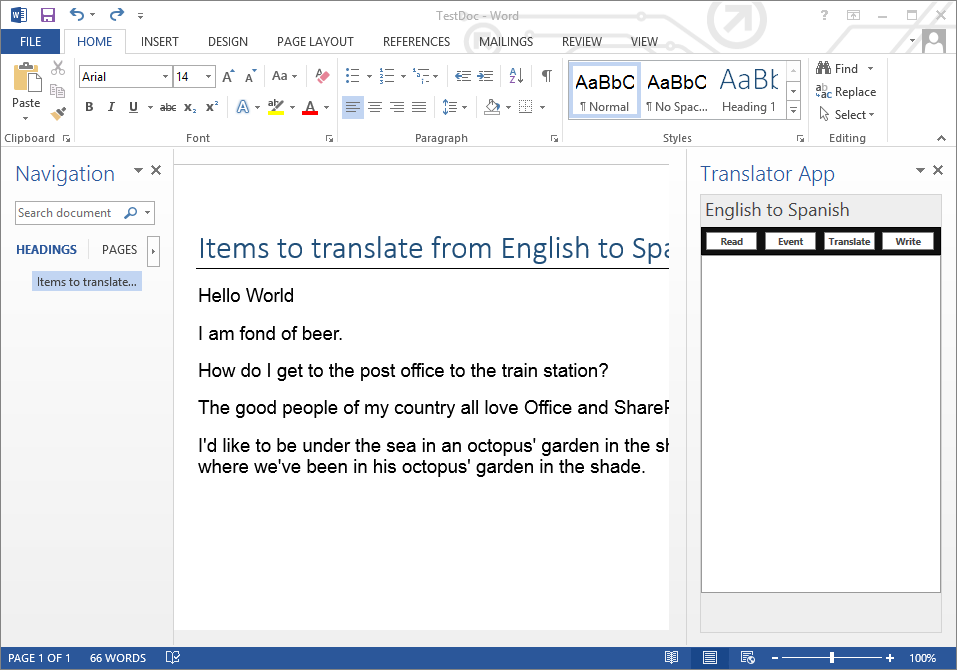
<input type="button" id="cmdWriteSelection" value="Write" />

</div>

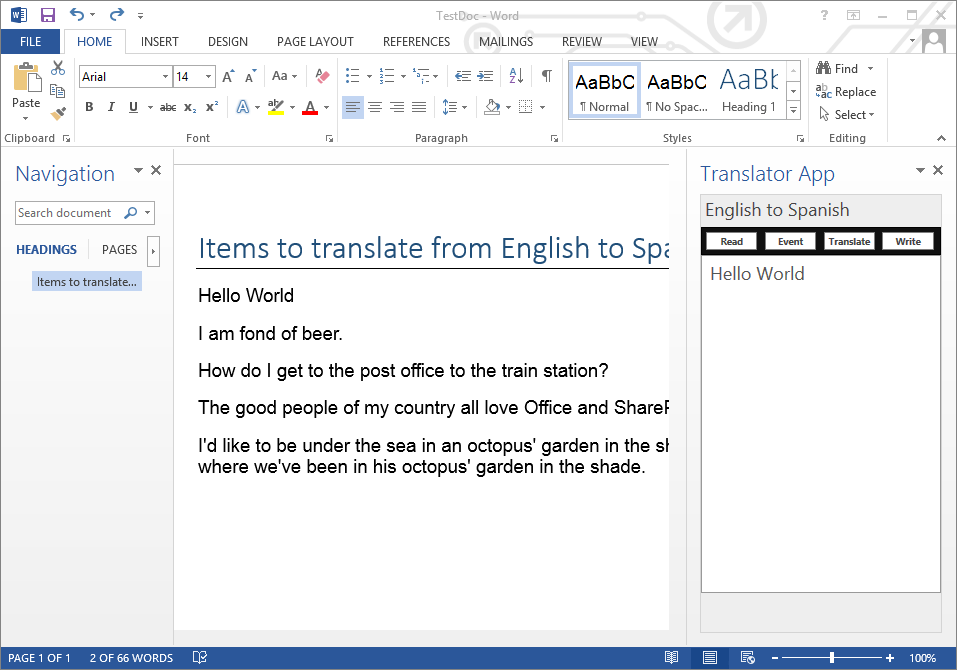
<div id="display" />

</body>

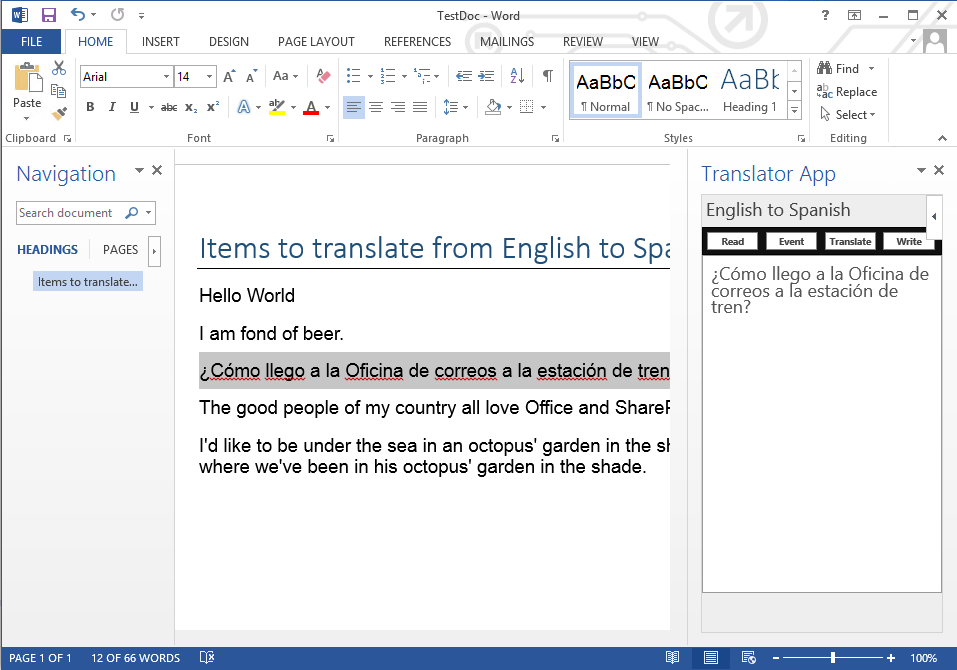
* + - 1. Open the **App.css** file inside the **Content** folder. Briefly walk through the CSS rules that are being used to style the elements of the user interface.
      2. Now it is time to demo the app and show students what it does before walking through any JavaScript code. Press the **{F5}** key to begin a debugging session which should start Microsoft Word, open the test document and start the app.



* + - 1. Select the first paragraph in the document with the contents of **"Hello World"** and then click the **Read** button. You should see that the app reads the value of the select region in the document and updates the user interface of the app.



* + - 1. Next click the **Event** button which will register an event handler that will execute whenever the user changes the selection. After you click the Event button, you should be able to select any text in the document and see the app automatically update its self with the newly selected value.
      2. Next, click the **Translate** button and show students that this will run code which will translate the text value displayed by the app from English to Spanish.
      3. Next, click the **Write** button to write the translated back to the document.
      4. Finish the demo by select one paragraph at a time and translating the content and replacing the English versions with the translated Spanish versions.



* + - 1. Now that you have shown the student how the app behaves, it is time to walk through the JavaScript code behind it. Return to Visual Studio and open the **TranslatorApp.js** file. Walk through the code inside and explain to students what it does. Start by discussing how there is code used to register an event handler callback function for each of the four command buttons.
      2. Walk through the implementation of the **onReadSelection** function. Explain how it calls **getSelectedDataAsync** passing a callback named **onReadSelectionComplete**. The callback function named **onReadSelectionComplete** is able to read the select region of the current document using **asyncResult.value** and write it to **display** div in the app's user interface.

function onReadSelection() {

officeDoc.getSelectedDataAsync("text", {},

onReadSelectionComplete);

}

function onReadSelectionComplete(asyncResult) {

var selection = asyncResult.value;

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

}

* + - 1. Walk through the implementation of the **onRegisterEventHandler** function which registers an event handler and implements a callback function that executes when the user changed the text selected within the current document.

function onRegisterEventHandler() {

officeDoc.addHandlerAsync("documentSelectionChanged",

onSelectionChanged);

}

function onSelectionChanged() {

officeDoc.getSelectedDataAsync("text", {},

onReadSelectionComplete);

}

* + - 1. Walk through the implementation of the **onTranslate** function. This is the method which creates an URL to download a JSONP script from the Bing Translation service which is able to pass a translated value back using a JSONP callback function.

**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. Point out that the script returned from the Bing Translation service return a JSONP script which automatically calls a callback function named **onTranslateComplete**. The **onTranslateComplete** has been implemented to take the translated result returned and to use it to update the **display** div in the app.

**function onTranslateComplete(translatedText) {**

**if (translatedText) {**

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

**}**

**}**

* + - 1. Walk through the **onWriteSelection** function and show how it writes the translated content back to the Word document using a call to **setSelectedDataAsync**.

function onWriteSelection() {

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

}

* + - 1. You have now reach the end of the demo. Make any closing comments and return slides to resume the lecture.

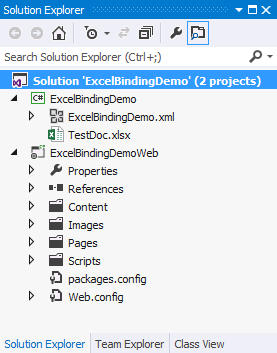
# Demo 2: Binding to Cells and Tables in Excel Spreadsheets

In this demo you will demo a pre-written app named **ExcelBindingDemo** to demonstrate writing code to create bindings and register event handlers.

## Step 1 – Open and Run the ExcelBindingDemo App

In this Step, you will open and demo an Excel task pane app.

* + - 1. Open Microsoft Visual Studio 2012
      2. From the **File** menu select the **Open Project** command and open the solution named **ExcelBindingDemo.sln** located inside the **Demo 2 ExcelBindingDemo** folder for this demo.
      3. Once the project has opened, take a moment to show students the two projects inside. The first project named **ExcelBindingDemo** just contains an xml file for the app manifest named **ExcelBindingDemo.xml**. The other project is the "remote web" project which is named **ExcelBindingDemo Web**.



* + - 1. Point out to students that there is a test document named **TestDoc.xslx** which has been configured as the Start Action of the top project named **ExcelBindingDemo**. The key point is that each document this document will be used each time you press the {F5} key to start a debugging session.
      2. Open **ExcelBindingDemo.html** inside the **Pages** folder and show students how the user interface for the app has been created using a simple set of HTML elements in the body section.

<body>

<h1>Excel Binding Demo</h1>

<div id="toolbar">

<input type="button" id="cmdBindCell" value="Bind Cell" />

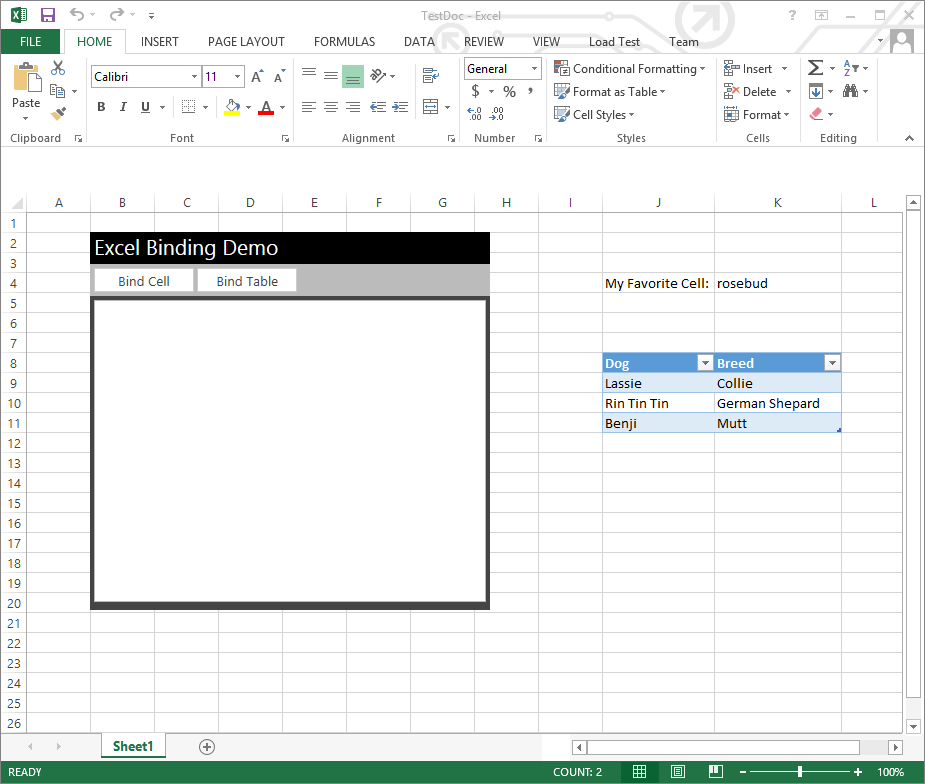
<input type="button" id="cmdBindTable" value="Bind Table" />

</div>

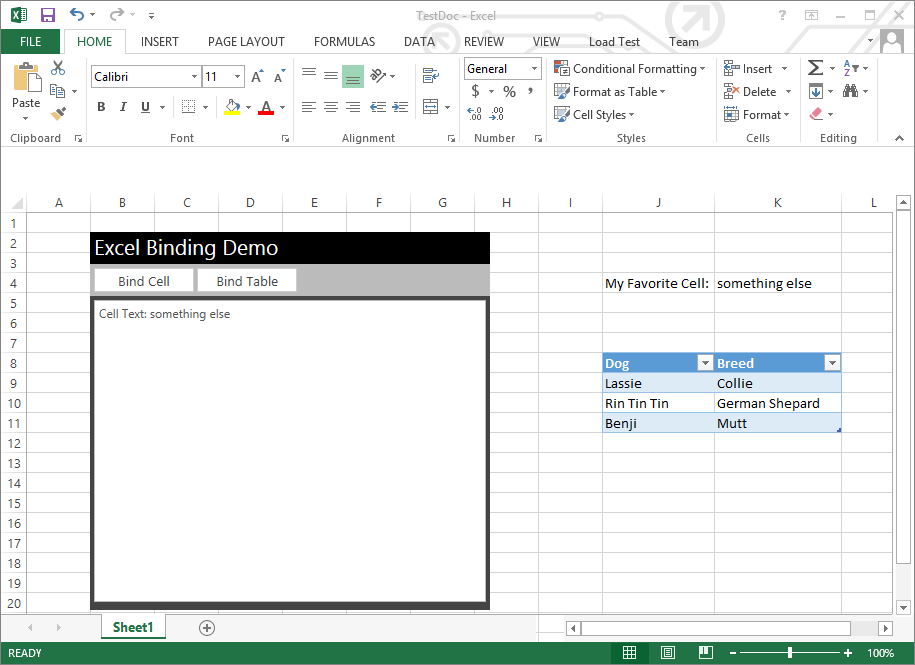
<div id="display" />

</body>

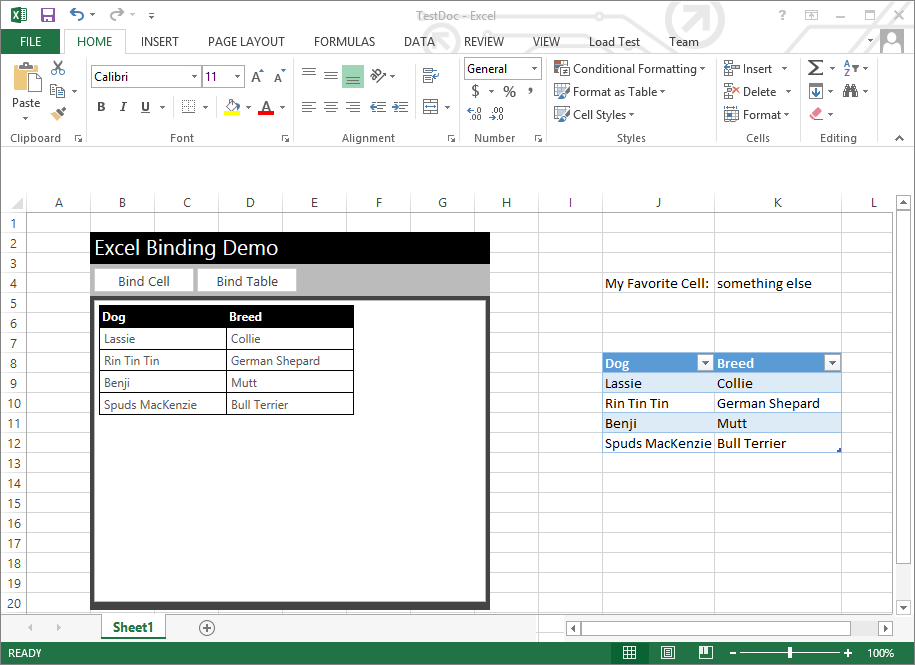
* + - 1. Open the **App.css** file inside the **Content** folder. Briefly walk through the CSS rules that are being used to style the elements of the user interface.
      2. Now it is time to demo the app and show students what it does before walking through any JavaScript code. Press the **{F5}** key to begin a debugging session which should start Microsoft Excel, open the test document and start the app.



* + - 1. Now you will establish the first binding. Start by selecting the cell in the spreadsheet that has the value of **"rosebud"**. Next click the **Bind Cell** button which will run code to create a binding to that cell and to wire up and event handler that runs any time the cell value changes. After clicking the Bind Cell button, you should be able to change the cell value from rosebud to something else and see that the app reacts by updating its user interface with the new value.



* + - 1. Now click on the **Bind Table** button which runs the code to create a binding to the table in the Excel document that contains data concerning a set of dog. After you click the Bind Table button, you should then be able to make changes to data in the table and have the app write the new values into the display area of the app.



* + - 1. Now that you have shown students how the app behaves, open the JavaScript file named ExcelBindingDemo.js and walk through the code in each of these functions to discuss how the apps binding behavior is implemented.
* onCellBindingDataChanged
* onNewCellDataRead
* onBindTable
* onBindTableComplete
* onTableBindingDataChanged
* onNewTableDataRead
  + - 1. You are now done with the demo. Return to the slides and resume the lecture.

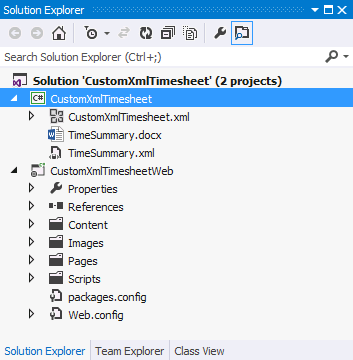
# Demo 3: Working with Custom XML and Content Controls

In this demo you will demo a pre-written app named **CustomXmlTimesheet** to demonstrate writing code to working with custom XML in a Word App for Office.

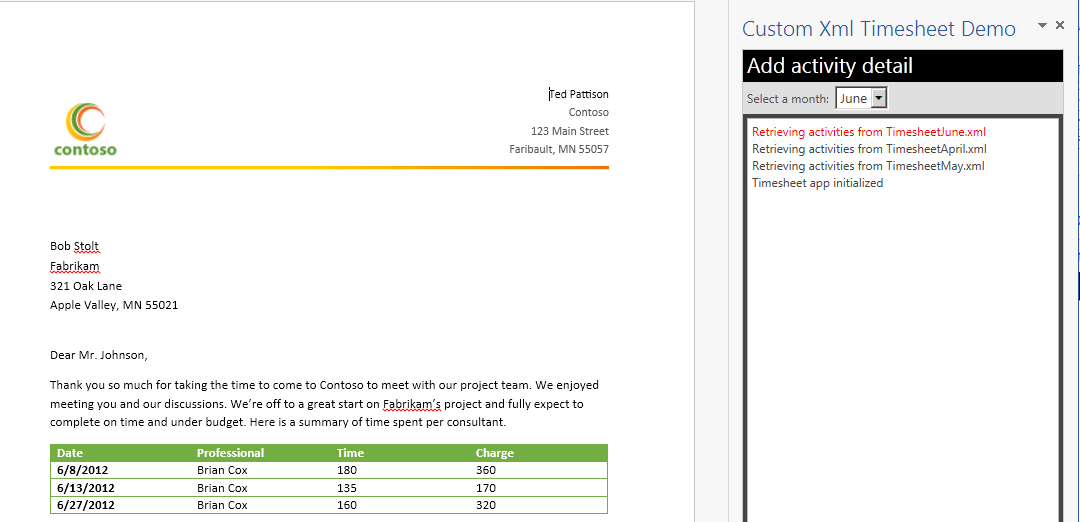
## Step 1 – Open and Run the CustomXmlTimesheet App

In this Step, you will open and demo an Excel task pane app.

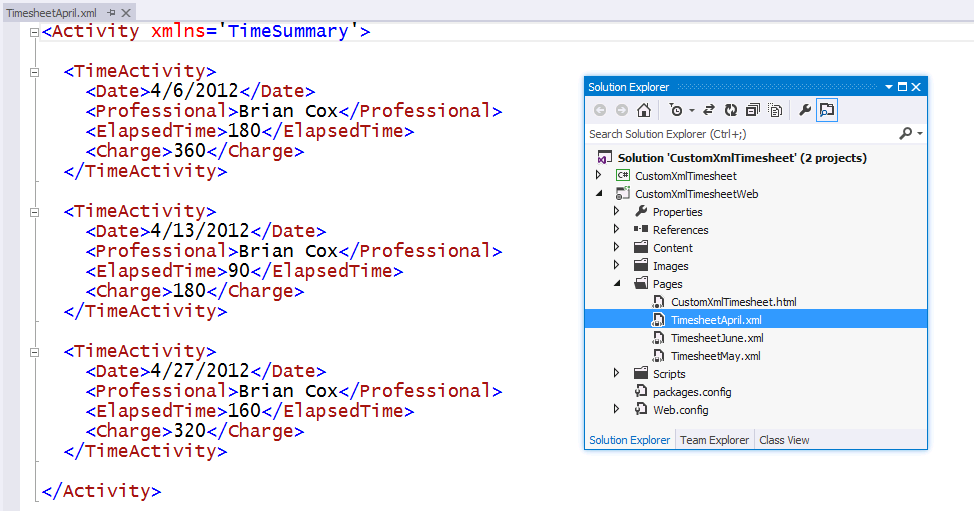
* + - 1. Open Microsoft Visual Studio 2012
      2. From the **File** menu select the **Open Project** command ad open the solution named **CustomXmlTimesheet.sln** located inside the **CustomXmlTimesheet** folder for this demo.
      3. Once the project has opened, take a moment to show students the two projects inside. The first project named **CustomXmlTimesheet** just contains an xml file for the app manifest named **CustomXmlTimesheet.xml**. The other project is the "remote web" project which is named **CustomXmlTimesheet Web**.



* + - 1. Point out to students that there is a test document named **TestDoc.docx** which has been configured as the Start Action of the top project named **CustomXmlTimesheet**. The key point is that each document this document will be used each time you press the {F5} key to start a debugging session.
      2. Now it is time to demo the app and show students what it does before walking through any JavaScript code. Press the **{F5}** key to begin a debugging session which should start Microsoft Word, open the test document and start the app.



* + - 1. When you select a different month, you should see that the activity detail in the Word document is updated with new activities for that month. Explain to students that the activity is being displayed using a Word Content Control that is reading a custom XML document that is embedded inside the document. Whenever you select a new month, the app acquires the XML=based activities for that and overwrite the node in the activities node in the custom XML behind the document. This causes an automatic refresh.
      2. Now that you have shown students how the app behaves, stop the debugging session and return to Visual Studio to walk through this demo app's implementation. Begin by showing students that the app have three XML files in the pages directory which contain the XML-based activity data for each of the three months.



* + - 1. Next, open the JavaScript file named **CustomXmlTimesheet.js** and walk through the code in each of these functions to discuss how the apps behavior is implemented.
* onMonthChanged
* onGotXml
* onGotNode
* onXmlRetrieved
* onXmlWrite
* postMessage
  + - 1. You are now done with the demo. Return to the slides and resume the lecture.