

Hand­s-on lab

Lab: Hosted Web Apps and WebViews

September 2015

Contents

[Overview 3](#_Toc430877270)

[Exercise 1: Interact with Platform APIs 5](#_Toc430877271)

[Task 1 – Create a blank Universal Windows JavaScript app 5](#_Toc430877272)

[Task 2 – Create the native project 6](#_Toc430877273)

[Task 2 – Add a webview to the web project 9](#_Toc430877274)

[Task 3 – Add a new HTML page and a script to generate toast 11](#_Toc430877275)

[Exercise 2: Create a Hosted Web App 15](#_Toc430877276)

[Task 1 – Create the Azure Website 15](#_Toc430877277)

[Task 2 – Publish to Azure 18](#_Toc430877278)

[Task 3 – Host the web app 24](#_Toc430877279)

[Task 4 – Send a toast notification 26](#_Toc430877280)

[Exercise 3: Support Additional Platforms and Devices with ManifoldJS (optional) 29](#_Toc430877281)

[Task 1 – Install ManifoldJS and create a manifest 29](#_Toc430877282)

[Summary 30](#_Toc430877283)

Overview

Following the release of Windows 10, the Windows Bridge toolkits are opening up the UWP platform to Android, iOS, Classic Windows, and web developers. The Windows Bridge for web apps allows you to easily transition your code for the web into the app space by publishing your responsive website to the Windows Store. Web apps and hosted web apps have the ability to call UWP APIs directly from JavaScript to integrate with features such as lives tiles, active notifications, contacts, Cortana voice commands, and Windows Store in-app purchases.

Hosted web apps immediately reflect changes made in your web codebase, making it easy to keep your content up to date. You may mix local and remote content to provide offline backups for the site.

After creating web apps for Windows, you may be interested in expanding to other platforms. ManifoldJS is a new open source framework that generates hosted web apps for major platforms.

# Objectives

* 1. This lab will show you how to:
  + Created a hosted web app
  + Serve up remote and local content
  + Mix online and offline content
  + Call platform APIs from the web app
  + Generate hosted web apps with ManifoldJS

# System requirements

* 1. You must have the following to complete this lab:
  + Microsoft Windows 10
  + Microsoft Visual Studio 2015
  + Azure subscription or free trial

# Optional add-ons

* 1. If you wish to complete the optional tasks in this lab, you will need:
  + The Node Package Manager (npm)
  + ManifoldJS

# Setup

* 1. You must perform the following steps to prepare your computer for this lab:
  2. Install Microsoft Windows 10.
  3. Install Microsoft Visual Studio 2015. Choose a custom install and ensure that the Universal Windows App Development Tools are selected from the optional features list.
  4. Optional: Install npm.
  5. Optional: Install ManifoldJS

*Instructions and links to install npm and ManifoldJS can be found in Exercise 3: Task 1.*

# Exercises

* 1. This Hands-on lab includes the following exercises:
  2. Add a Simple Hosted Web App
  3. Integrate with Platform APIs
  4. Support Additional Platforms and Devices with ManifoldJS (Optional)
  5. Estimated time to complete this lab:  **45 to 60 minutes**.

Exercise 1: Interact with Platform APIs

* 1. In Windows 10, you have the ability to create a seamless user experience for web apps by interacting with platform APIs. In this exercise, you will trigger a toast notification from a web app that includes content displayed through a WebView. Although this content is local rather than hosted, it demonstrates how JavaScript running in the WebView can call C# code in your native project.

Task 1 – Create a blank Universal Windows JavaScript app

We will begin by creating a project from the UWP Blank App JavaScript template.

1. In a new instance of Visual Studio 2015, use **File > New> Project** to open the New Project dialog. Navigate to **Installed >** Templates **> JavaScript** and select the **Blank App (Universal Windows)** template.
2. Name your project **WebApp** and select the file system location where you save your Hands-on Lab solutions. We have created a folder in our **C:** directory called **HOL** that you will see referenced in screenshots throughout the labs.
3. Leave the options selected to **Create new solution** and **Create directory for solution**. You may deselect **Add to source control** if you don't wish to version your work. Click **OK** to create the project.

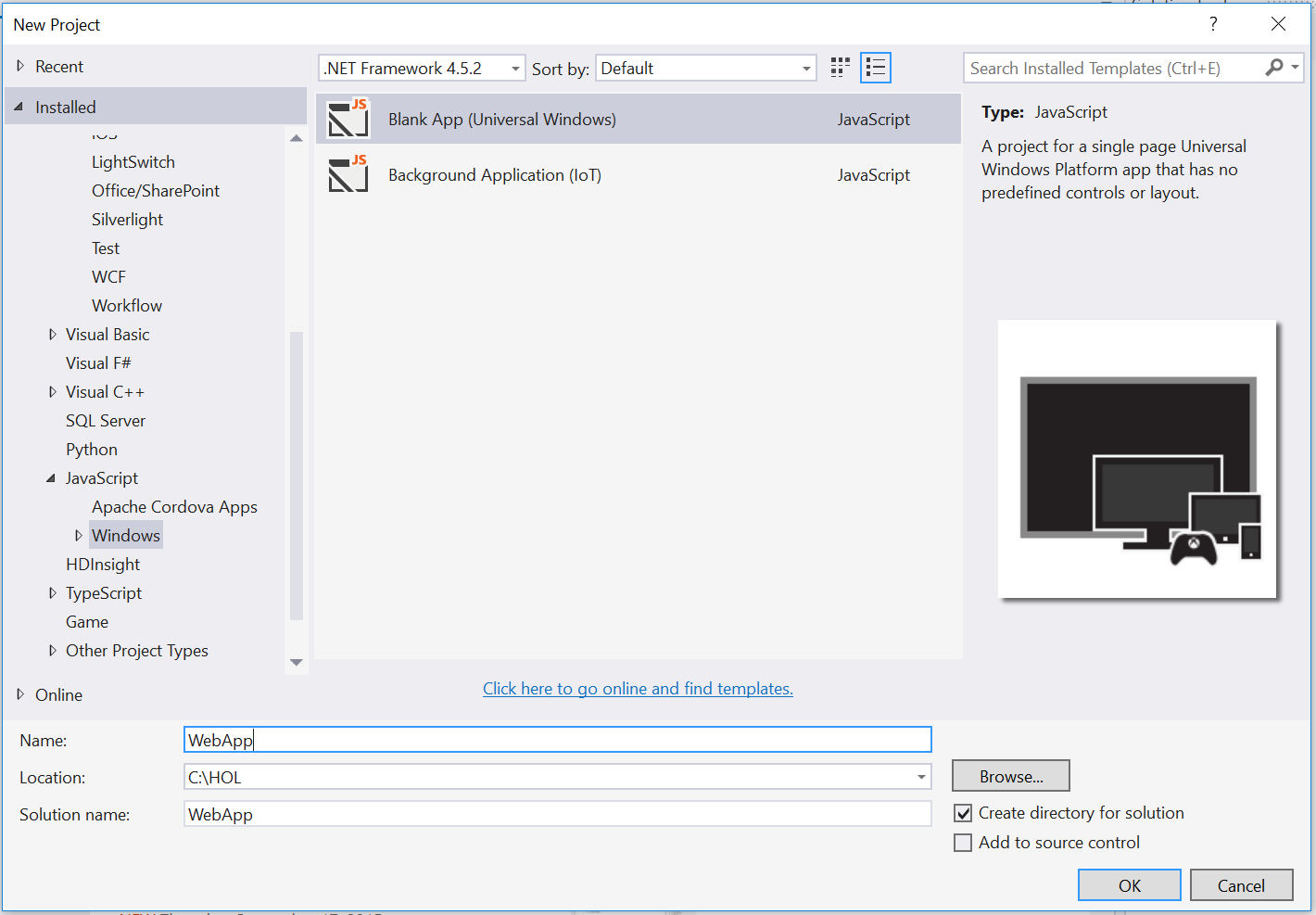


Figure 1

Create a new Blank App project in Visual Studio 2015.

1. Set your **Solution Configuration** to **Debug** and your **Solution Platform** to **x86**. Select **Local Machine** from the Debug Target dropdown next to the Start Debugging Button.

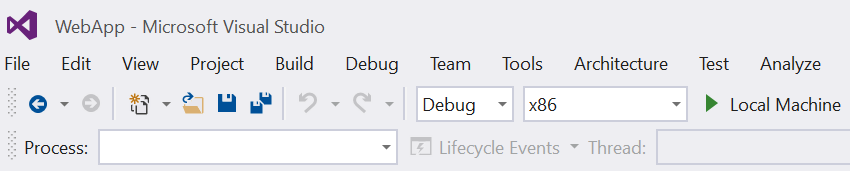


Figure 2

* + 1. Configure your app to run on the Local Machine.
  1. **Note:**  is the Start Debugging button.

1. Use the Start Debugging button to build and run your app. You will see a black app background with the text “Content goes here.”

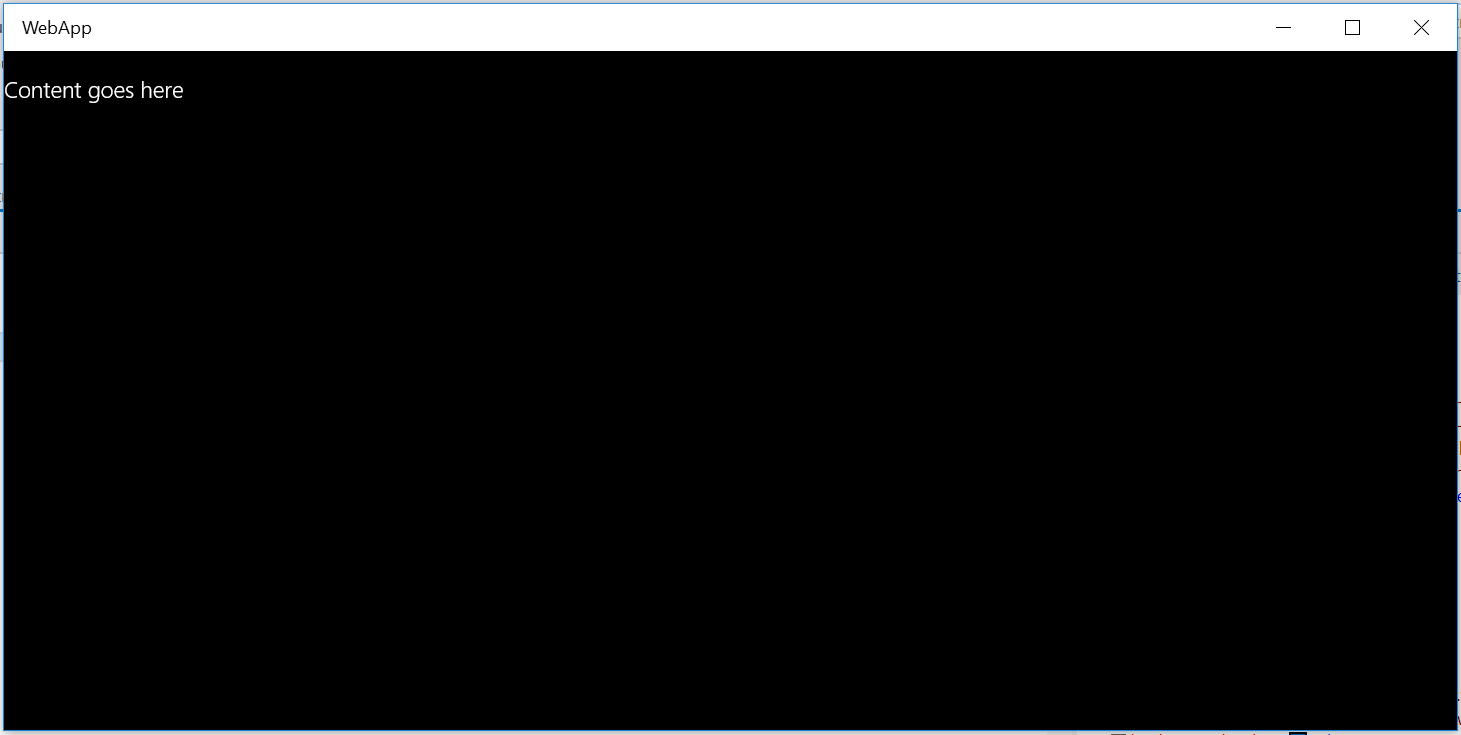


Figure 3

The blank universal JavaScript app running in Desktop mode.

1. Stop debugging and return to Visual Studio.

Task 2 – Create the native project

Our goal in this exercise is to trigger a toast notification from a webview. In this task, you will create a Windows Runtime Component to handle the toast notification through Platform APIs.

1. In your WebApp project, right-click on the solution name in the Solution Explorer and choose **Add > New Project**.
2. Select the project type **Visual C# > Windows > Windows Runtime Component (Universal Windows)** and give it the name **CommunicationWinRT**. Click **OK** to create the project.

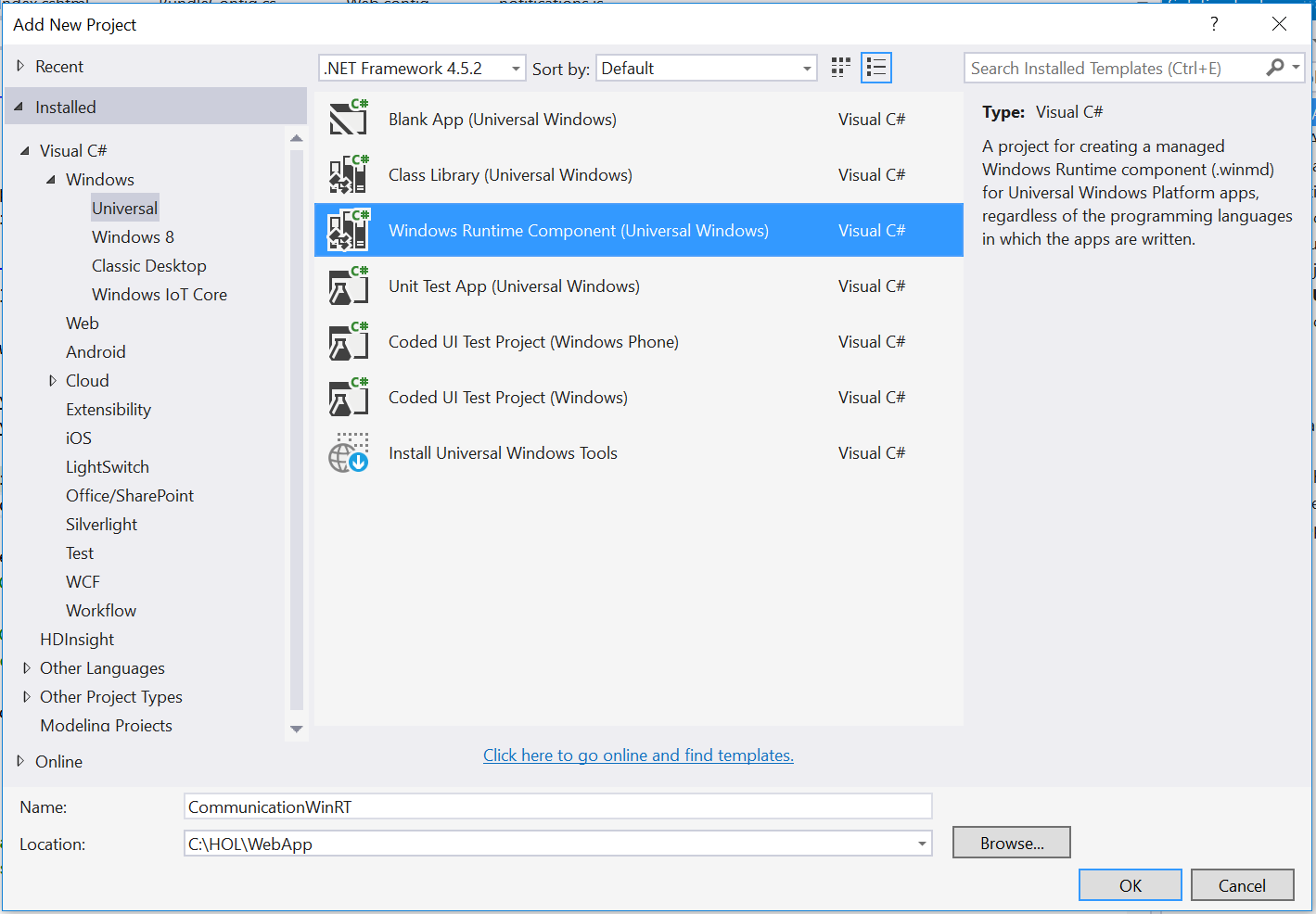


Figure 4

Add the CommunicationWinRT project.

1. Right-click on the **Class1.cs** file in the Solution Explorer and choose **Rename** to rename it to **CommunicationWinRT**. If prompted to perform a rename in the project of all references to Class1, choose **Yes**.

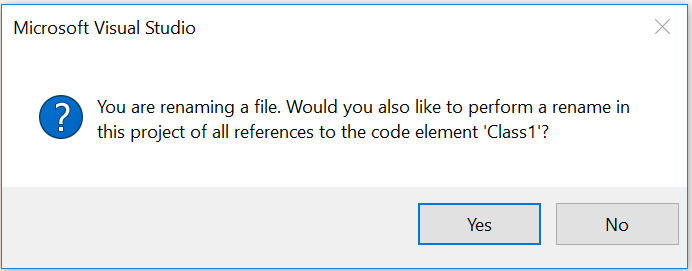


Figure 5

Rename Class1.cs to CommunicationWinRT.cs.

1. Open **CommunicationWinRT.cs**. Add the Windows.UI.Notifications and Windows.Data.Xml.Dom namespaces to the class.
   * 1. C#
   1. using Windows.UI.Notifications;
   2. using Windows.Data.Xml.Dom;
2. Add the **AllowForWeb** meta content and a constructor to the sealed class **CommunicationWinRt**.
   * 1. C#
   1. namespace CommunicationWinRT
   2. {
   3. [Windows.Foundation.Metadata.AllowForWeb]
   4. public sealed class CommunicationWinRT
   5. {
   6. public CommunicationWinRT()
   7. {
   8. }
   9. **Note:** A sealed class cannot be inherited from or expanded with properties from JavaScript. WinRT objects are sealed from JavaScript so that JavaScript can’t interfere with their behavior.
   10. For a Windows Runtime object to be projected and available to a WebView, it must have the AllowForWeb attribute.
3. Add an **async** method below the constructor that accepts a string and delay value which will be used to create a toast notification.
   * 1. C#
   1. public CommunicationWinRT()
   2. {
   3. }

public async void toastMessage(String message, int delay)  
{  
 ToastTemplateType toastTemplate = ToastTemplateType.ToastText01;  
 XmlDocument toastXml = ToastNotificationManager.GetTemplateContent(toastTemplate);

* 1. XmlNodeList toastTextElements = toastXml.GetElementsByTagName("text");  
      toastTextElements[0].AppendChild(toastXml.CreateTextNode(message));
  2. ToastNotification toast = new ToastNotification(toastXml);
  3. await Task.Delay(delay);  
      ToastNotificationManager.CreateToastNotifier().Show(toast);  
     }

1. Save **CommunicationWinRT.cs** and return to the web project.
   1. **Note:** You may add additional sealed public classes to the **CommunicationWinRT** namespace to handle other API integrations.
   2. For a deeper look at notifications, check out the **Live Tiles and Notifications Lab**.
2. Right-click on the **References** directory in the **WebApp** project and choose **Add > Reference**. Add the **CommunicationWinRT** project as a reference.

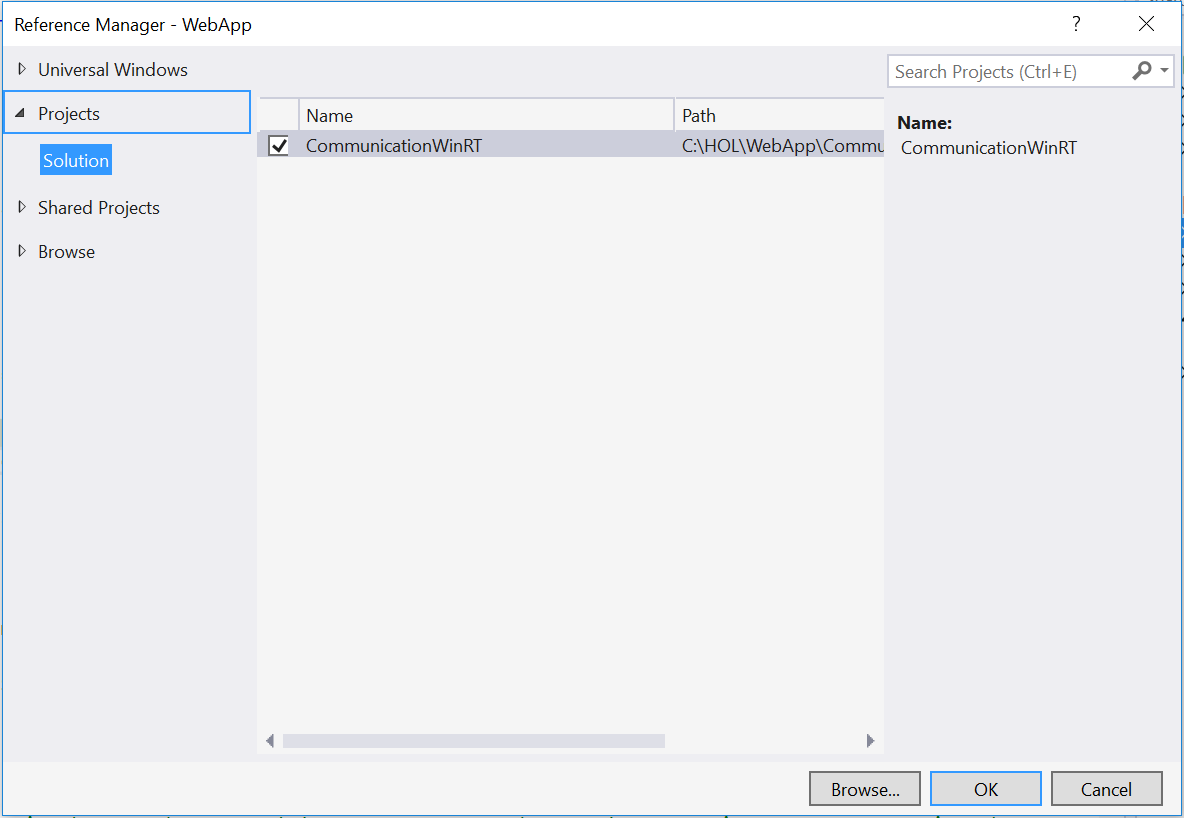


Figure 6

Add the native project as a reference to the web project.

Task 2 – Add a webview to the web project

In Windows 10, your JavaScript app can directly host navigation to a remote URI. However, a WebView may still be necessary when you need more granular control. In this task, you will add a webview element to default.html and prepare your project to display local content.

* 1. **Note:** A WebView can run locally and point to content on the web server and vice versa. In Windows 10, the WebView is now separated off the UI thread to improve performance and responsiveness. The WebView enables Windows Runtime access and uses the current Edge rendering engine to serve up content.

1. In **default.html**, replace the content of the **<body>** tag with a **WebView** element. Set its id to **toastView**. The id will make it easier to target the webview from your JavaScript.
   * 1. HTML
   1. <body class="win-type-body">

<x-ms-webview id="toastView"></x-ms-webview>

</body>

1. Open **default.js**. Give your webview a height and width in the **ActivationKind.Launch** conditional.
   * 1. JavaScript
   1. if (args.detail.kind === activation.ActivationKind.launch) {

var webview = document.getElementById('toastView');

* 1. webview.style.height = "100%";
  2. webview.style.width = "100%";

1. Instantiate the **CommunicationWinRT** class as a JavaScript object and use **addWebAllowedObject** to expose and name it within the webview. We have chosen to give it the name **toastApi** within the context of the webview.
   * 1. JavaScript
   1. if (args.detail.kind === activation.ActivationKind.launch) {

var webview = document.getElementById('toastView');

* 1. webview.style.height = "100%";
  2. webview.style.width = "100%";

var communicationWinRT = new CommunicationWinRT.CommunicationWinRT();

webview.addWebAllowedObject("toastApi", communicationWinRT);

1. Save and close **default.js**..
   1. **Note:** The webview has access to Windows APIs, so you don’t need to specify a Content URI to achieve WinRT access in this case. If you host remote content without a webview, you will need to specify Content URIs and access levels for that content in the app manifest. You will learn more about Content URIs in Exercise 2.

Task 3 – Add a new HTML page and a script to generate toast

Your app is now set up to host content in your webview. In this task, you will add an HTML page that will display in the webview and run a script to generate a toast notification.

1. Right-click on the WebApp project and select **Add > New Item**. Choose the **HTML Page** item type and give it the name **toastGenerator.html**. Add the page to the project.

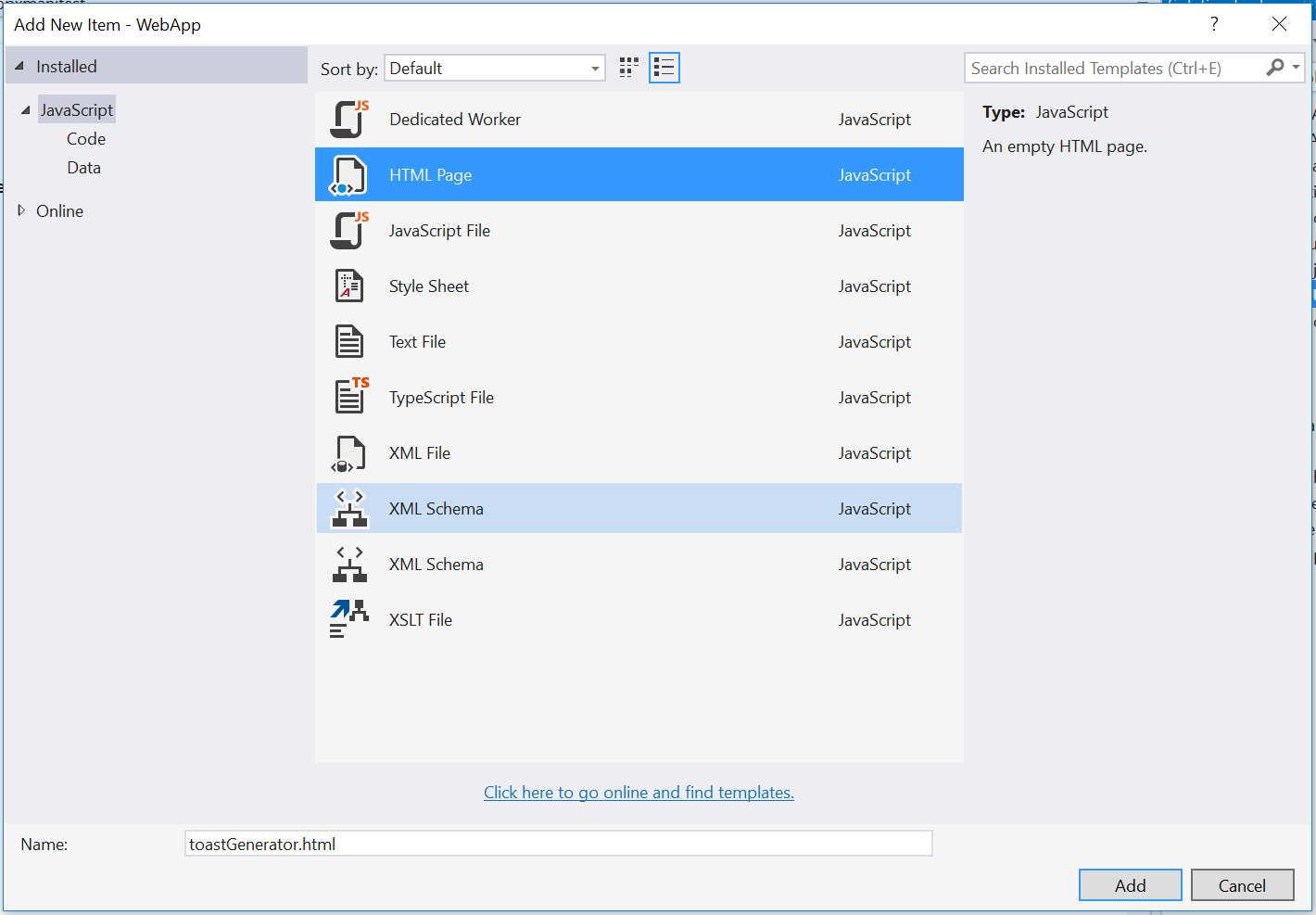


Figure 8

Add a new HTML page to the web project.

1. In **toastGenerator.html**, add a page title to the **<title>** element. Create a button in the **<body>** whose **onclick** event calls the **sendToast()** function. You will create the sendToast() handler in the next step.
   * 1. HTML
   1. <!DOCTYPE html>  
      <html>  
      <head>  
          <title>Toast Integration</title>  
      </head>  
      <body>  
          <button onclick="sendToast()">Notify</button>  
      </body>  
      </html>
2. Add the **sendToast()** handler in a **<script>** tag. This function looks for the **toastApi** object and calls its **toastMessage()** method if found. If the method toastApi object is not found, your console log will show an error message.
   * 1. HTML
   1. <!DOCTYPE html>  
      <html>  
      <head>  
          <title>Toast Integration</title>  
          <script type="text/javascript">  
              function sendToast() {  
                  var object = window.toastApi;

            if (object) {  
                console.log(object + ' found');  
                object.toastMessage('Hello Toast', 0);  
            }  
            else {  
                console.log(object + " not found in the DOM.")  
            }  
        }  
    </script>  
</head>  
<body>  
    <button onclick="sendToast()">Notify</button>  
</body>  
</html>

1. Return to **default.js** to navigate your webview to the **toastGenerator.html** page.
   * 1. JavaScript
   1. webview.addWebAllowedObject("toastApi", communicationWinRT);

webview.navigate("ms-appx-web:///toastGenerator.html");

1. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Build and run your app on the **Local** Machine.

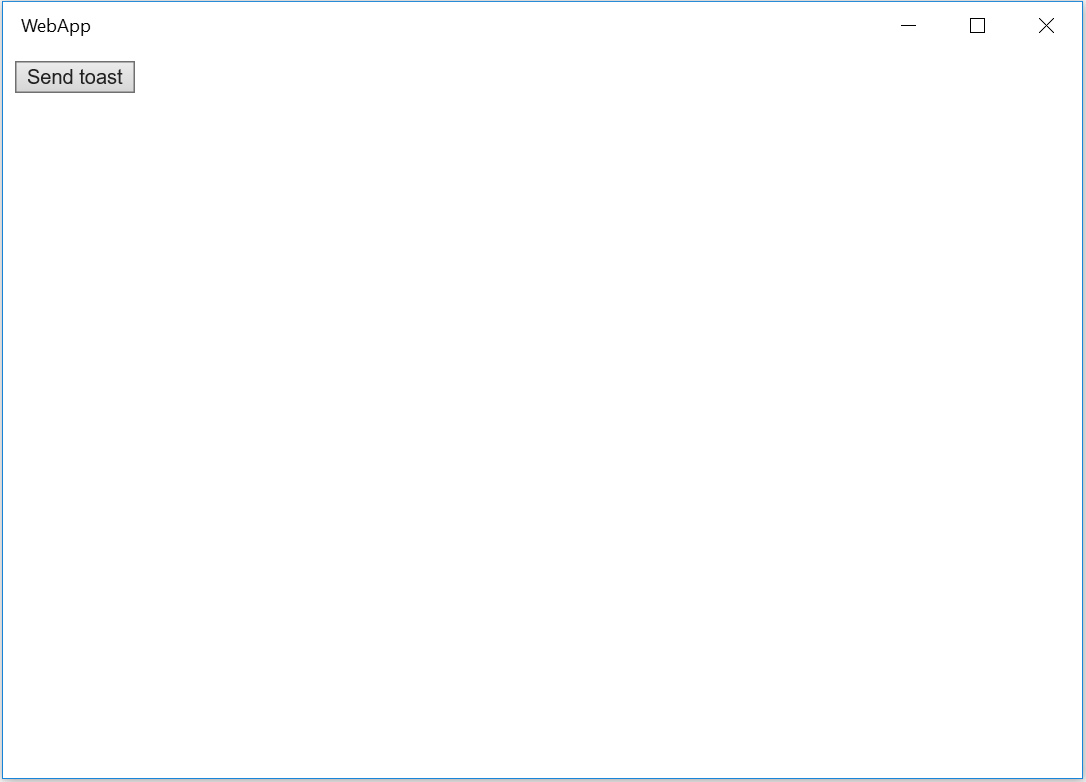


Figure 9

The web app displays the **Send toast** button.

1. Use the **Send toast** button to generate a toast notification. Your **Hello Toast** message will pop up near the Action Center icon in the task bar.

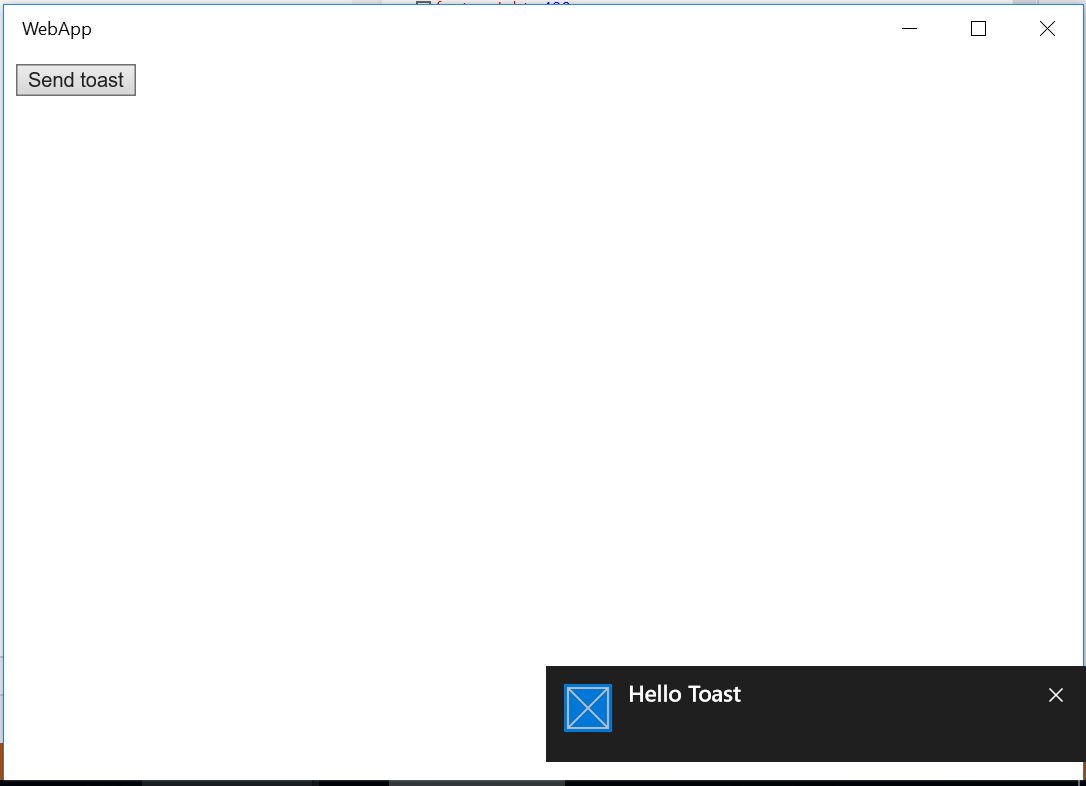


Figure 10

The web project has successfully integrated with the native project to generate toast.

1. Stop debugging and return to Visual Studio.

Exercise 2: Create a Hosted Web App

* 1. With a responsive site already online, you can create a hosted web app for the Windows Store in minutes. A hosted web app essentially functions as a browser using the Edge rendering engine. In this exercise, you will create an MVC website project, publish it to Azure, host it in a web app, and trigger a toast notification from the hosted content.

Task 1 – Create the Azure Website

In this task, you will create an MVC website project.

1. Right-click on the WebApp solution name in your Solution Explorer and choose **Add > New Project**. Choose a project type of **Visual C# > Web > ASP.NET Web Application**. Leave the Application Insights option deselected and give it the name **AzureWebApp**.

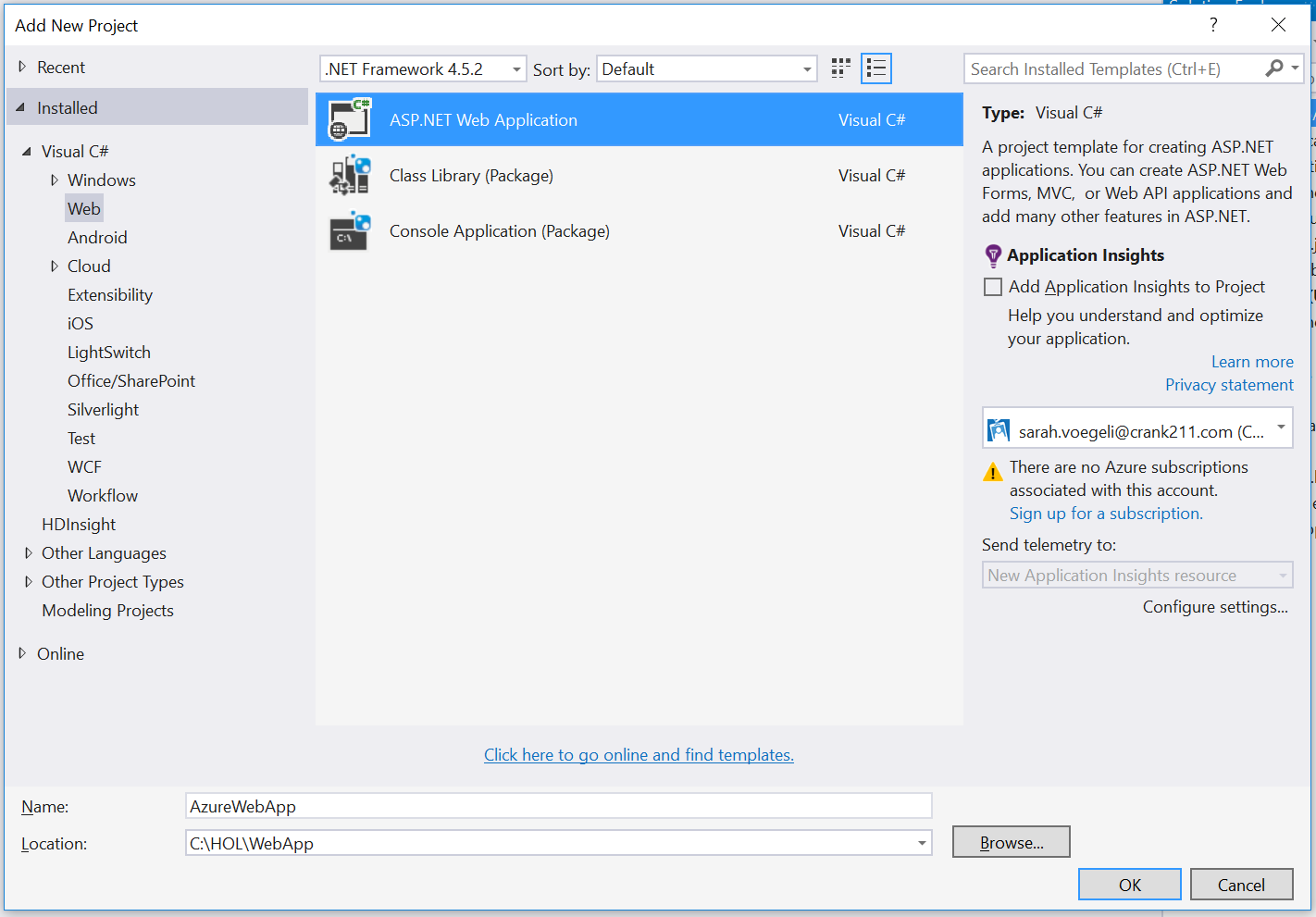


Figure 11

Add a website project.

1. When the **New ASP.NET Project** dialog appears, select the **MVC** template. Check the box to **Add folders and core references** for **MVC**. Leave **Web Forms**, **Web API**, **Add unit tests**, and **Host in the cloud** deselected. You will publish to Azure in the next task.

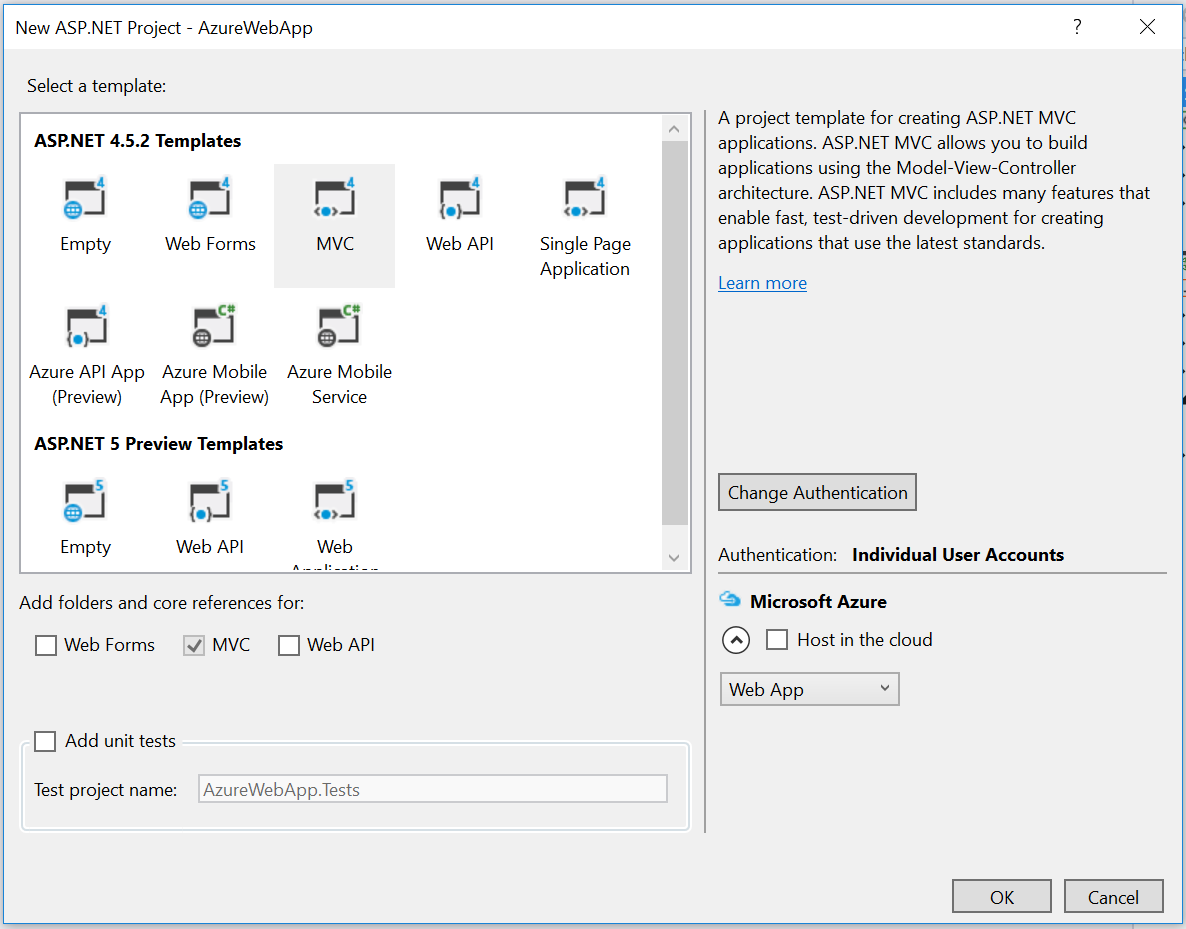


Figure 12

Select MVC options.

1. Right-click on the **AzureWebApp** project in the Solution Explorer and select **Set as StartUp project.**
2. With your Solution Platform set to **x86** and the Debug Target set to **Internet Explorer**, build and run your app. You will see the default MVC index page running on localhost.

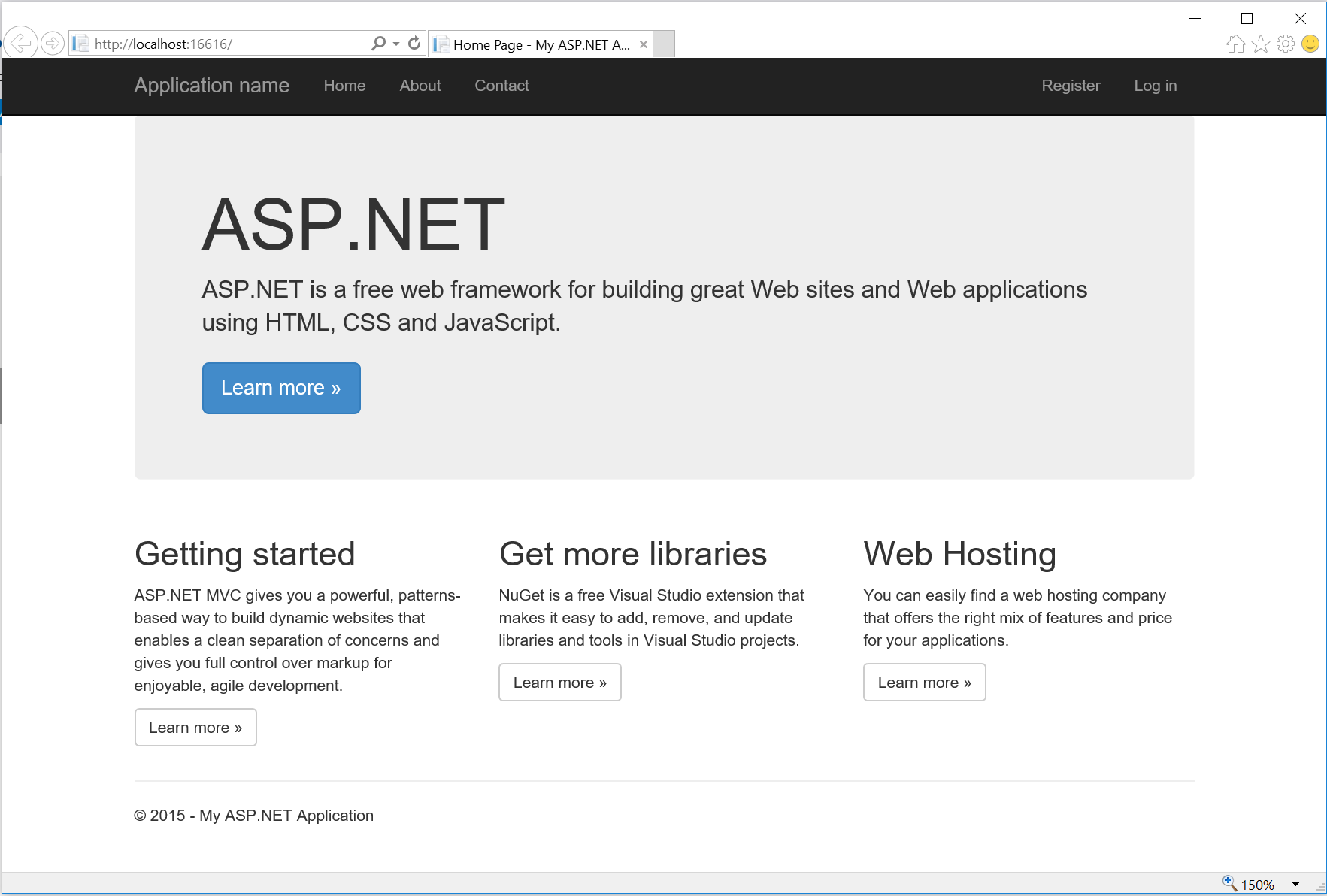


Figure 13

The default MVC website running in Internet Explorer.

1. Stop debugging and return to Visual Studio.

Task 2 – Publish to Azure

To create a true hosted web app, it needs to be published to a server. In this task, you will follow the steps to publish the site to an existing Azure subscription.

* 1. **Note:** If you don’t already have an Azure subscription, you can sign up for a free 30-day trial at <https://azure.microsoft.com/en-us/pricing/free-trial/>. The free trial will work for this demo.

1. Right-click on the **AzureWebApp** project in the Solution Explorer and choose **Publish**.
2. When the Publish Web dialog opens, select **Microsoft Azure Web Apps** as the publish target.

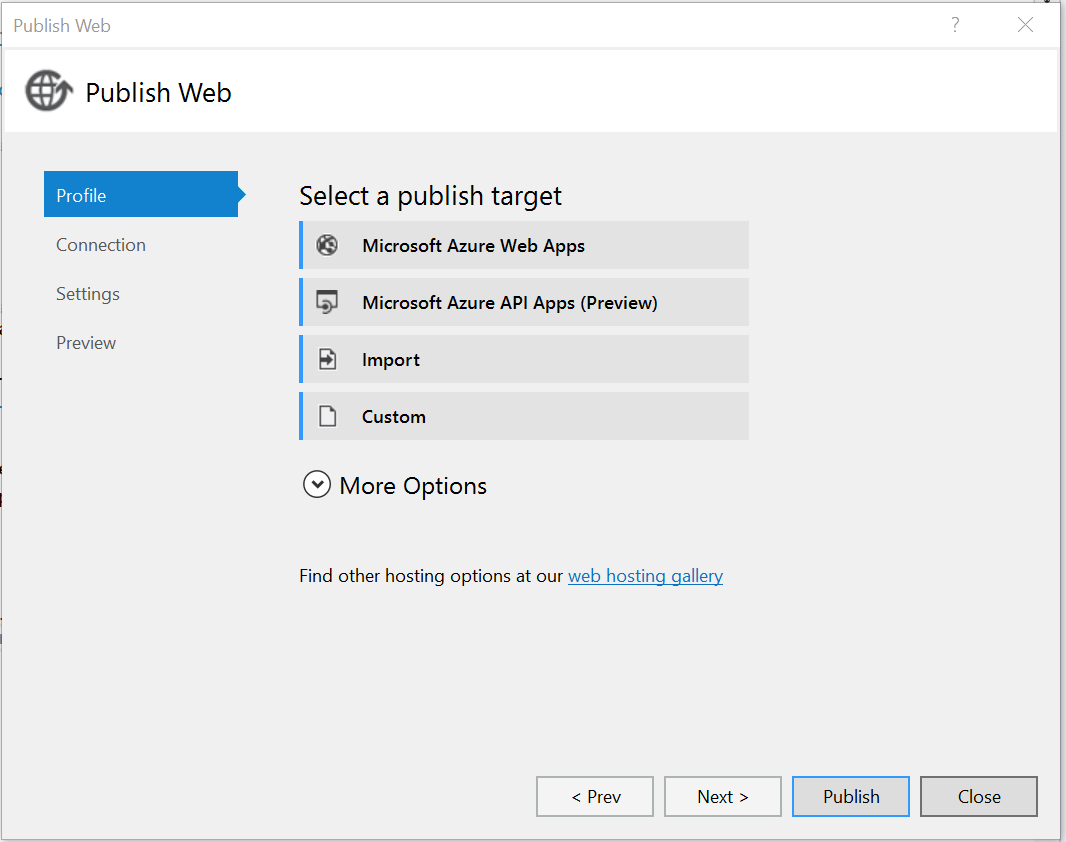


Figure 14

The Publish Web dialog.

1. A dialog will open to connect with your Azure account. Use the dropdown menu to select or add the account you used to sign up with Azure. Reenter your credentials if prompted.

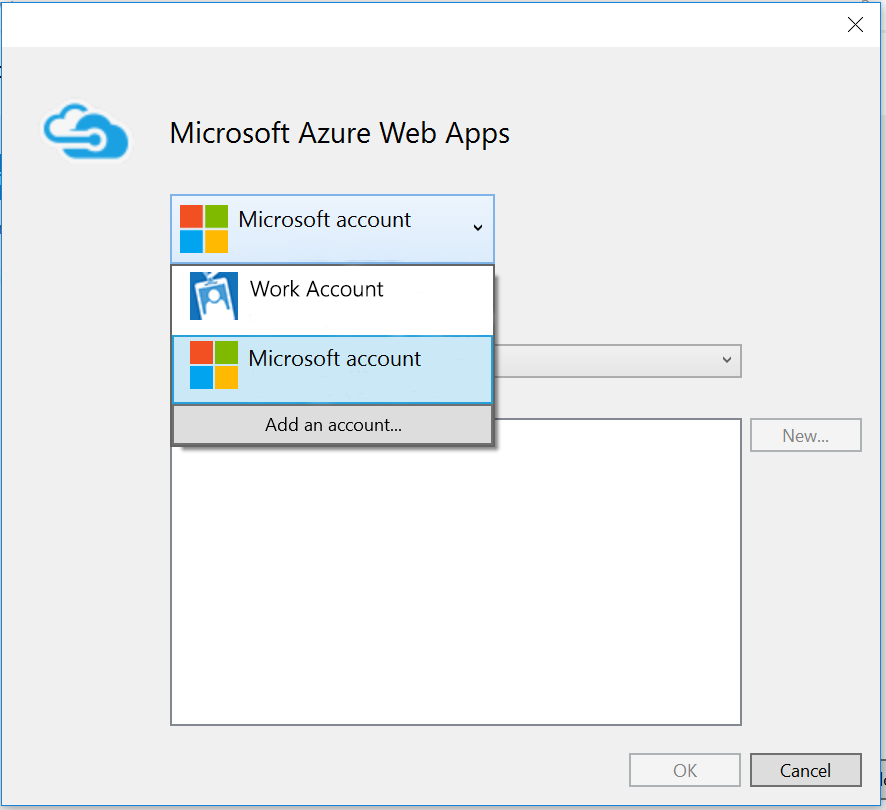


Figure 15

Sign in to Azure.

* 1. **Note:** If you signed up for Azure but still see a message in the dialog saying **This account has no subscriptions**, you may have more than one Microsoft account. Microsoft accounts and Microsoft Work/School accounts are distinct, even if you signed up for each with the same email address.
  2. Check which account is linked to your Azure subscription and add its credentials to proceed.

1. Once you are logged in, use the Subscription drop-down menu to select the subscription you would like to use. If you are on the free trial, it will be the only subscription available.

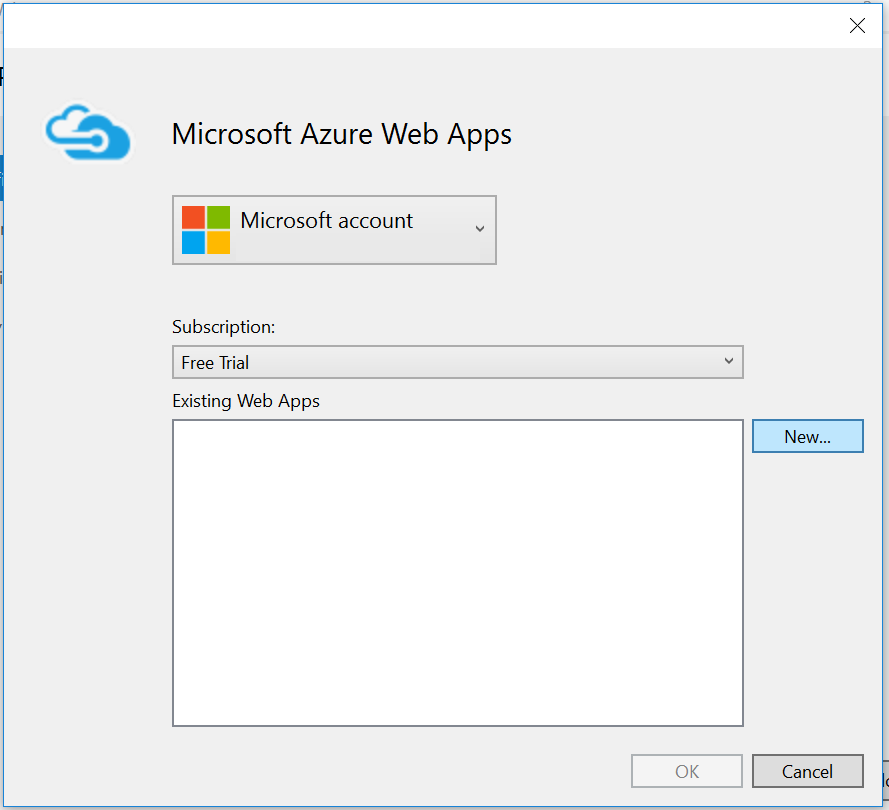


Figure 16

Select a subscription.

1. While still in the Microsoft Azure Web Apps dialog, use the **New** button to add a Web App to your account.

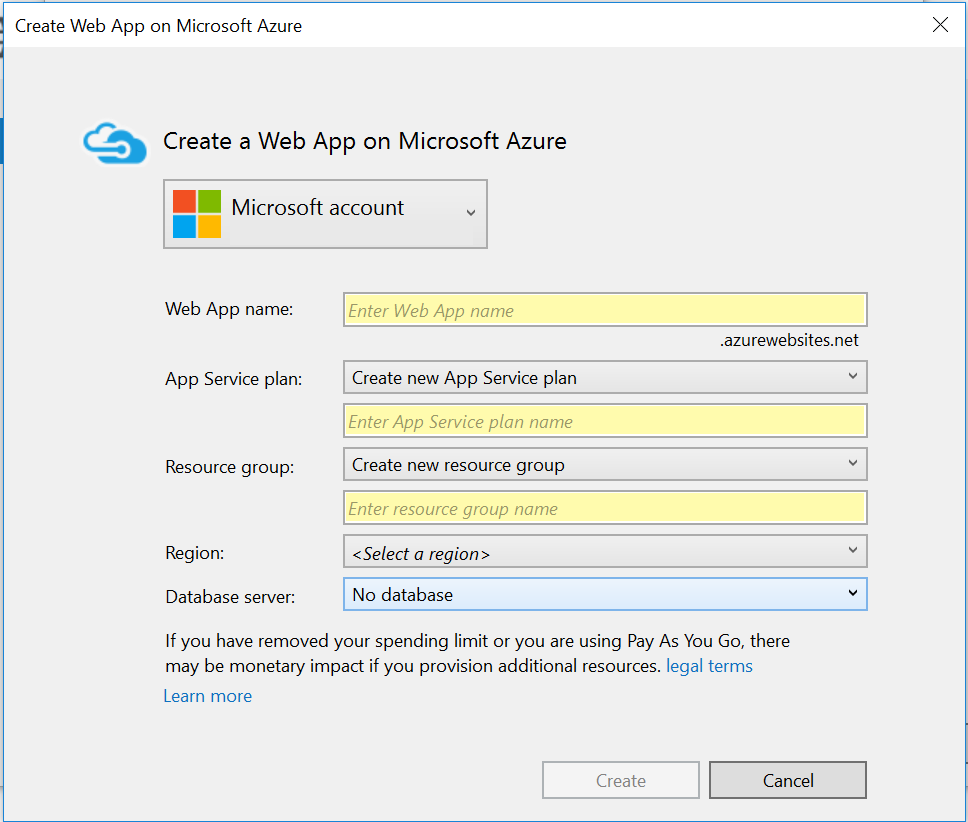


Figure 17

Set up a Web App on Azure.

1. Give your web app a unique name. Your app will be hosted at azurewebsites.net. A green checkmark will appear if the name is not in use.
2. Use the dropdown menus to select or create an **App Service plan** and **resource group**. Choose a region near you from the **Region** dropdown.
3. Leave the **Data server** option set to **No database**. Click **Create** to add your web app. You may need to wait a few minutes for the process to complete.
4. After adding the web app, you will return to the Publish Web dialog. Leave the publish method set to Web Deploy and keep all the default settings in the **Connection**, **Settings**, and **Preview** tabs.

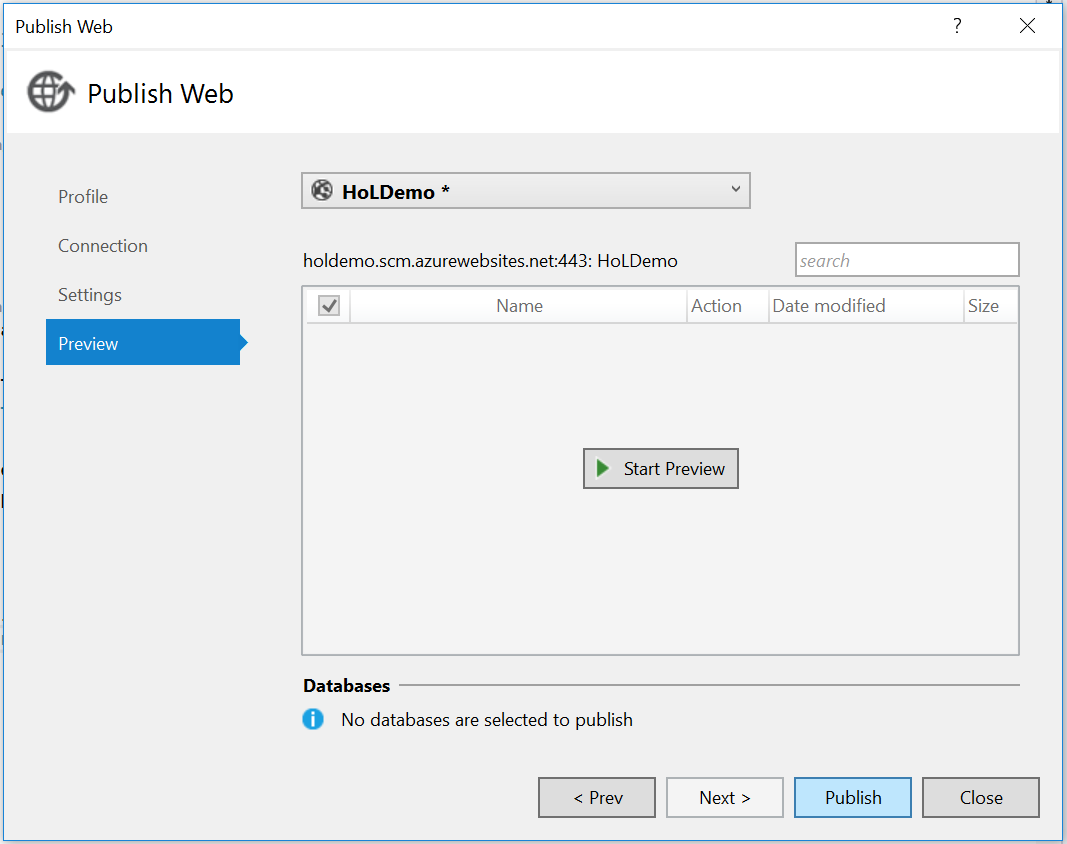


Figure 18

Publish the web app.

1. Use the **Publish** button to publish your web app.
2. Your web app will open in the browser with a URI at **azurewebsites.net**. Resize the browser window to view its responsive behavior.

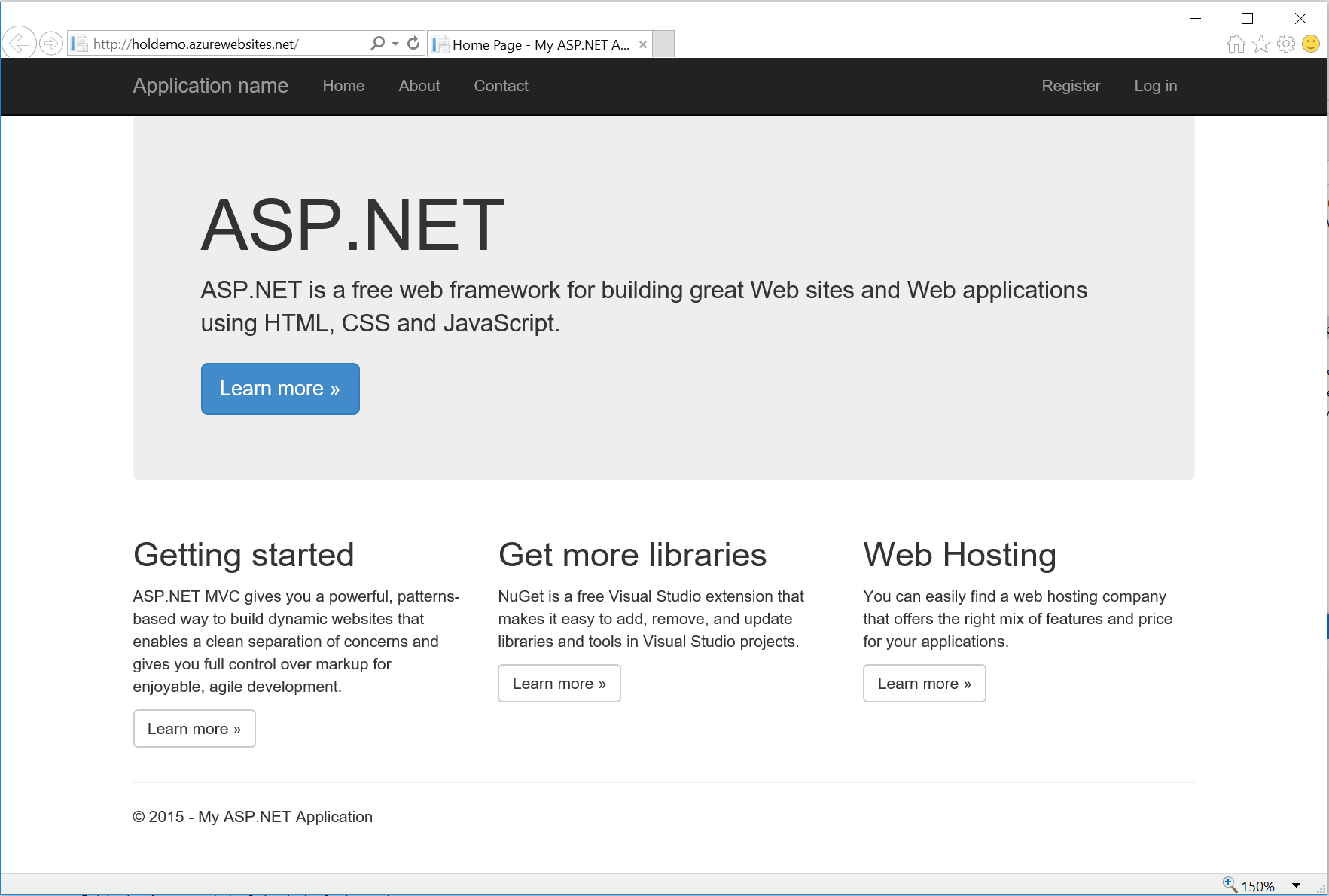


Figure 19

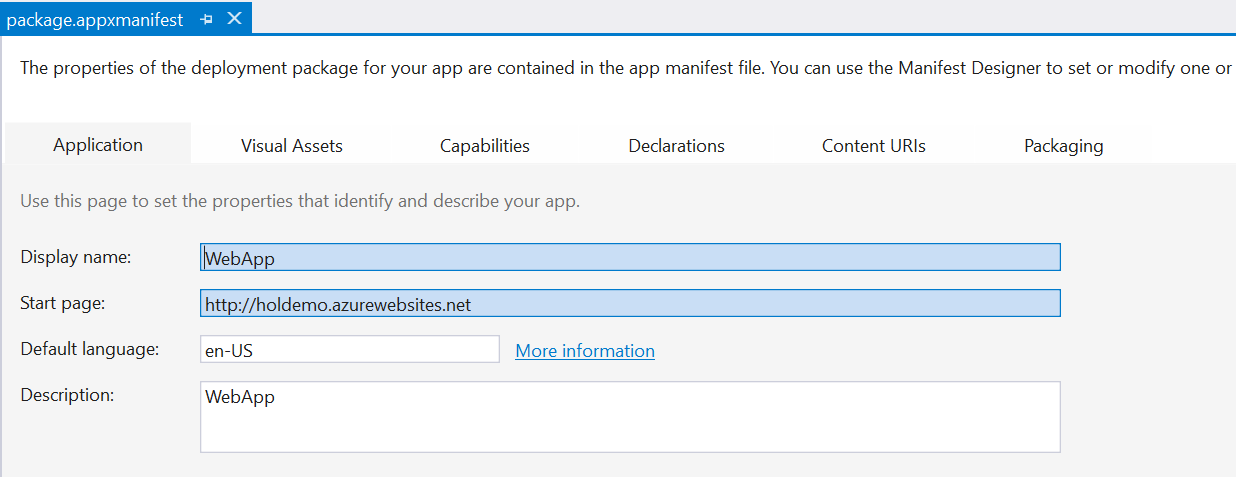
The published web app.

1. Close the browser and return to Visual Studio.

Task 3 – Host the web app

You now have a live, responsive site that is ready to be hosted in an app. In this task, you will set up your WebApp project to host the HostedWebApp site.

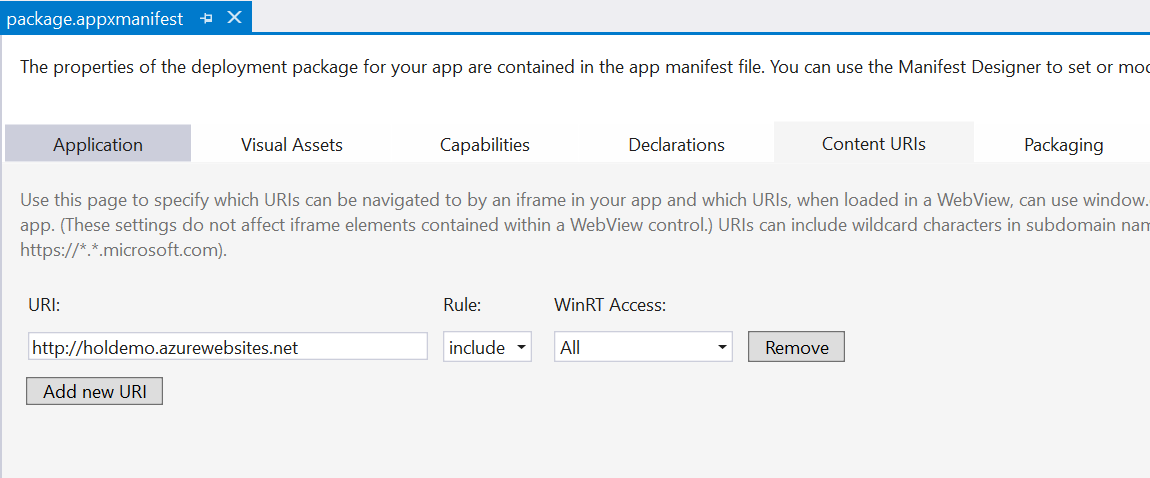
1. Set the **WebApp** project as your **StartUp** project in the Solution Explorer. Open its **Package.appxmanifest** in the manifest editor.
   1. **Note:** When creating a web app that contains only hosted content, you may delete the **css**, **js**, and **WinJS** folders as well as the **default.html** file. However, you may wish to use these files to create an offline backup of your site in case your server is not available. It is good practice to create a local error page.
2. On the **Application** tab, change the **Start page** to the URI of your Azure site.



Figure

Set your Azure site as the Start page.

1. Navigate to the the **Content URIs** tab. Add your Azure website URI as a content URI. Set its **Rule** to **Include** and **WinRT access** to **All**. You will use the WinRT access in the next task.

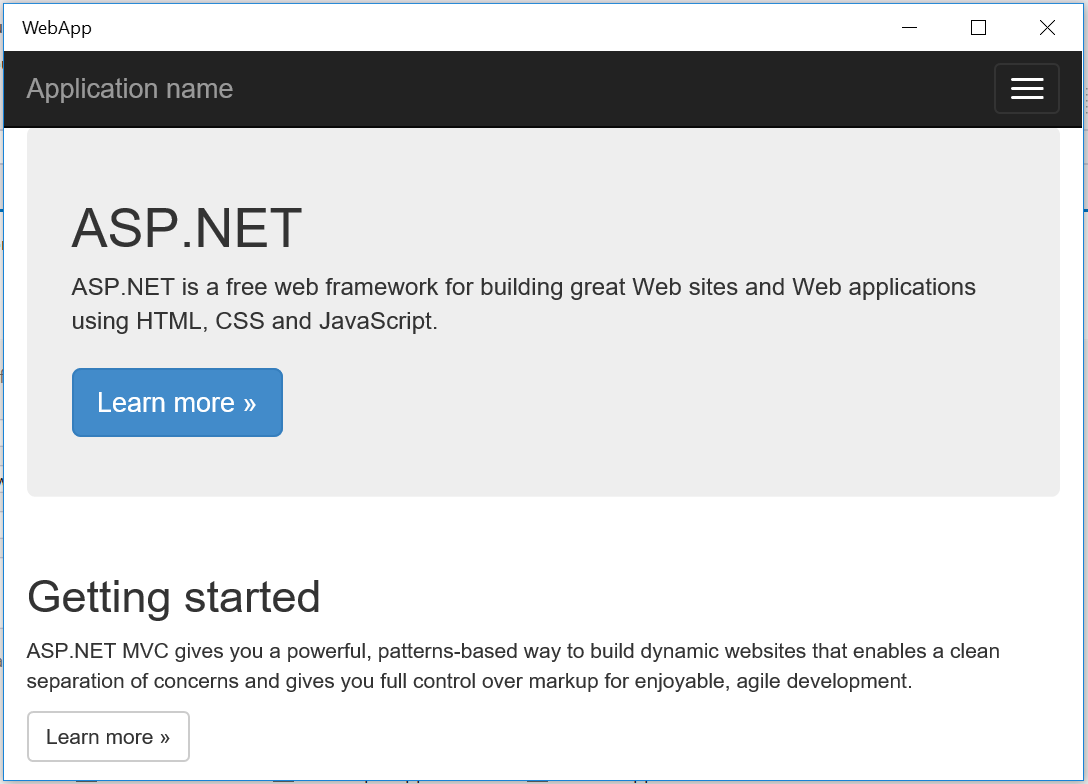


Figure

Add the Content URI for your Azure site and give it WinRT access.

* 1. **Note:** The Application Content URI Rules (ACURS) for your app dictate the pages that are hosted or allowed by the app. For instance, you may wish your users to be able to browse your website within the app but force external links to open in a browser. These inclusions and exclusions allow you to control the boundaries of your app and prevent it from behaving like a standard web browser. Content URIs also give you the ability to turn Windows Runtime access on or off for different parts of the app and to decide if that access should be given for **None**, **All**, or **Allow for web only**.
  2. To specify a remote URI, use the **http://** protocol. To specify a local URI, use the **ms-appx-web:///** protocol.

1. Save and exit the manifest. Build and run the **WebApp** project. You will see the Azure site in the app window. The hosted web app pulls live content from the Azure server, so you will see the latest version of the site every time you run the app. Resize the app window to see how the responsive design of the site carries over to the app.



Figure

The Azure site as a hosted web app.

1. Stop debugging and return to Visual Studio.

Task 4 – Send a toast notification

Now that we have a hosted web app up and running, let’s set it up to send a toast when it detects it is running as hosted app content.

1. Return to the **AzureWebApp** project. Right-click on the **Scripts** folder and choose **Add > JavaScript file**. Give it the name **notifications.js**.
2. Open **App\_Start > BundleConfig.cs**. Add **notifications.js** to the Bundle containing bootstrap.js and respond.js. Remember to add a comma after the previous script in the list.
   * 1. C#
   1. bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
   2. "~/Scripts/bootstrap.js",
   3. "~/Scripts/respond.js",
   4. "~/Scripts/notifications.js"));
3. Open notifications.js and copy in the following functions to create a toast notification.
   1. **Note:** The **if (window.Windows)** conditional feature detects if the API surface is available. This code will only run inside the hosted web app. You gave the hosted web app permission to use WinRT in the Content URI in the previous task.
      1. JavaScript
   2. if (window.Windows) {
   3. console.log('this printed from the file')
   4. function createToast(title, message, imgUrl, imgAlt, tag) {
   5. console.log('create toast triggered')
   6. // Namespace: Windows.UI.Notifications
   7. if (typeof Windows !== 'undefined' &&
   8. typeof Windows.UI !== 'undefined' &&
   9. typeof Windows.UI.Notifications !== 'undefined') {
   10. // Setup variables for shorthand
   11. var notifications = Windows.UI.Notifications,
   12. templateType = notifications.ToastTemplateType.toastImageAndText02,
   13. templateContent = notifications.ToastNotificationManager.getTemplateContent(templateType),
   14. toastMessage = templateContent.getElementsByTagName('text'),
   15. toastImage = templateContent.getElementsByTagName('image'),
   16. toastElement = templateContent.selectSingleNode('/toast');
   17. var launchParams = {
   18. type: 'toast',
   19. id: tag || 'demoToast',
   20. heading: title || 'Demo title',
   21. body: message || 'Demo message'
   22. };
   23. var launchString = JSON.stringify(launchParams);
   24. // Set message & image in toast template
   25. toastMessage[0].appendChild(templateContent.createTextNode(message || 'Demo message'));
   26. toastImage[0].setAttribute('src', imgUrl || 'https://unsplash.it/150/?random');
   27. toastImage[0].setAttribute('alt', imgAlt || 'Random sample image');
   28. toastElement.setAttribute('duration', 'long');
   29. toastElement.setAttribute('launch', launchString); // Optional Launch Parameter
   30. // Show the toast
   31. var toast = new notifications.ToastNotification(templateContent);
   32. var toastNotifier = new notifications.ToastNotificationManager.createToastNotifier();
   33. toast.tag = 'demoToast';
   34. console.log(toast);
   35. toastNotifier.show(toast);
   36. } else {
   37. var alertText = title || 'Demo Title';
   38. alert(alertText);
   39. }
   40. }

function notify() {

* 1. var title, message, imgUrl, imgAlt, tag
  2. title = 'Alert'
  3. message = "Hello from your hosted web app"
  4. imgUrl = 'http://images.itechpost.com/data/images/full/2094/windows-phone-8.jpg';
  5. imgAlt = 'this is image alt'
  6. tag = 'tag'
  7. createToast(title, message, imgUrl, imgAlt, tag);
  8. }
  9. }

1. Open **Views > Home > Index.cshtml**. Add a button to trigger the toast notification, and set the onclick to your notify() function.
   * 1. CSHTML
   1. <div class="row">
   2. <div class="col-md-4">
   3. <button onclick="notify()">Send toast</button>
   4. <h2>Getting started</h2>
2. Right-click on the **AzureWebApp** project in the Solution Explorer and choose **Publish**. This time, you will not need to go through the set up process. Reenter your credentials if prompted.
3. Build and run the **WebApp** project on the local machine. You will see your **Send toast** button now appears on the home page. Click the button to trigger the toast notification.
   1. **Note:** Click the Send toast button only once. If you would like to send the toast again, open the Notifications center and dismiss the previous toast.
   2. To troubleshoot your toast, open the JavaScript console in Visual Studio while the hosted web app is running. Click the **Send toast** button. If the **createToast** function has been triggered, you will see the log message **create toast triggered**.
4. Stop debugging and return to **Visual Studio**.

Exercise 3: Support Additional Platforms and Devices with ManifoldJS (optional)

* 1. Hosted web apps are a great way to quickly bring your existing responsive web projects to new platforms. ManifoldJS is a tool that uses existing metadata from your website to generated native hosted apps for a variety of platforms, including iOS, Android, Windows 10, Chrome OS, and Firefox OS. For platforms that don’t support hosted web apps natively, ManifoldJS uses Cordova.
  2. The manifest generated by ManifoldJS follows the W3C standard for web app manifests and includes metadata such as the start page for the site, URL whitelist, site name, theme color, and app images.

**Note:** For the latest on ManifoldJS, visit <http://www.manifoldjs.com/>. You can read more about the W3C manifest for web apps at <https://w3c.github.io/manifest/>.

Task 1 – Install ManifoldJS and create a manifest

Install ManifoldJS to and generate a manifest.

1. Open a command prompt as Administrator. With the node package manager installed, use the command **npm install –g manifoldjs** to install ManifoldJS globally on your development machine.
   * 1. Command Prompt
   1. > npm install –g manifoldjs
   2. **Note:** Visit <https://nodejs.org/> to download and install the node package manager (npm).
2. Generate a manifest for your site at <http://www.manifoldjs.com/generator>. You may also upload a manifest, and the generator tool will fix and alert you to any gaps it may have.
   1. **Note:** If your site doesn’t have a manifest, ManifoldJS will generate one for you. However, you may still wish to create your own to take advantage of your site’s branding and provide app images.
3. Upload the manifest to the root directory of your site on the server. The manifest typically lives in the same location as your index.html file.

Summary

* 1. Hosted web apps and webviews provide powerful options to integrate your existing web projects with the Windows Store and platform APIs. In this lab, we created a web app to host local and remote content and learned how to integrate native Windows APIs with JavaScript. We also explored options for generating hosted web apps for a variety of platforms.