Warnings and dependencies in the Node.js and npm world

07/04/2025

COM and VSTO add-in developers who are new to the world of Node Package Manager (npm) and open source development are often surprised and alarmed at certain aspects of this kind of development. This article is intended to reassure such developers about things they may find disconcerting.

npm dependency tree

npm is the standard package manager for the JavaScript runtime environment Node.js. It's used to streamline JavaScript development workflows by enabling developers to install open source libraries and tools (collectively called "packages"), and to manage package dependencies.

The npm dependency tree is a hierarchical structure that represents all the packages your Node.js project depends on. Each node in the tree is a package, and its children are the packages it depends on. This structure can become deeply nested, especially in large projects or when using packages with many transitive dependencies.

When you run `npm install`, npm reads the package.json and package-lock.json files in a development project to build this tree and fetch the required packages.

Understand `npm install` warnings

When you run `npm install`, it's common to see a flurry of warnings in the console. This can be surprising at first, but it's a normal part of working in the Node.js and open source ecosystem. Microsoft tools that call `npm install`, including the Yeoman Generator for Office Add-ins, report these same warnings.

It's beyond the scope of this article to discuss every kind of warning that `npm install` might report, but there are two kinds that are especially likely to be disconcerting to developers who are new to the Node.js world.

Deprecation warnings

These warnings mean that the managers of a package somewhere in the dependency tree are no longer maintaining it and they may remove it from the Internet at some future time. Neither

you, nor Microsoft, has any control over the package deprecation warnings, but you can almost always ignore them. Deprecation doesn't mean that the package has stopped working. It still works and because installation puts a copy of it on your computer, it'll continue to work with your project in the future even if the package is removed from the Internet. The package isn't a web service.

It's very unlikely, but possible, that you'll see a deprecation warning for a package that's at the *top* of the dependency tree. These are the packages that are explicitly listed in the "dependencies" or "devDependencies" sections of the project's package.json file. You can ignore deprecation warnings for "devDependencies" for the same reason given earlier: the code is copied to your development computer. Packages in the "dependencies" section are used by your add-in at runtime, but even deprecation warnings for these can be ignored in projects that are created with Microsoft tools like the Yeoman Generator for Office Add-ins and Microsoft 365 Agent Toolkit because these tools include copies of the libraries in bundles of JavaScript code that your add-in's web server will serve.

> **Note**
>
> One situation in which the deprecation of a library in the "dependencies" section is a matter of concern is the following:
>
> - The library is in the "dependencies" section only so you can use it while testing and debugging locally.
> - Your plan, when you deploy the add-in for staging or production, is to not include a copy of the library in the code that your server hosts.
> - Instead, you plan to have the add-in load the library from a web service that hosts npm libraries, such as unpkg.com or cdn.jsdelivr.net.
>
> If this describes your deployment strategy, then there's a danger that your deployed add-in will stop working if the deprecated library is removed from the web service. So, treat the deprecation warning as a notice that you need to redesign your add-in so that it doesn't use the deprecated library.

Security or audit warnings

Security warnings, also called audit warnings, mean that there's a version of the package in the dependency tree that has a known security vulnerability that a hacker could take advantage of. Microsoft periodically checks for these warnings in projects created by our tools and fixes them, usually by updating the library to a newer version that doesn't have the vulnerability. But new vulnerabilities are discovered and reported almost daily to the security alert databases that

`npm install` monitors, and Microsoft cannot always fix them right away. For this reason, it isn't uncommon that running `npm install` in an add-in project reports security vulnerabilities.

When the dependency can be traced to a top-level package in the "devDependencies" section of package.json, then you can ignore it. The code is only running on your computer, and you're not going to hack yourself.

If the dependency traces to a top-level package in the "dependencies" section, or you cannot determine the top-level package, then you should try to fix the vulnerability before you deploy the add-in to production. There's lots of good information on the Internet about how to deal with vulnerabilities in npm packages. We'll mention one technique here. Some vulnerabilities can be fixed automatically by npm. Just run the command `npm audit fix` in the folder where the package.json file is. If there's a newer version of the package that doesn't have the vulnerability and the newer version doesn't have any breaking changes relative to the vulnerable version, then npm will automatically update the package to the safe version.

Another strategy is to take a few minutes every couple of weeks to create a new add-in project with the same Microsoft tool as you created your original project. (Choose the same options for project type,, language, Office application, etc.) If `npm install` no longer reports the security vulnerability on the new project, then Microsoft has fixed it in the project template. You can move the fix to your project with the following steps.

1. Copy the "dependencies" section of from the new project over the same section in the **package.json** of the original project.
2. Delete the **node_modules** folder from the original project.
3. Run `npm install` in the original project.

Errors

An npm *error*, as distinct from a warning, immediately stops the processing of the npm command, including `npm install`. You must diagnose and fix the problem. Sometimes the error is a side effect of a temporary network problem when npm tries to fetch a package. Try rerunning `npm install`.

> **Note**
>
> Running `npm install` is the last thing that the Yeoman Generator for Office Add-ins does when it creates a project. If an error is reported, you don't need to rerun the generator because the project has been fully created. You can just rerun `npm install` at the command line.

Tutorial: Share code between both a VSTO Add-in and an Office Add-in with a shared code library

Article • 05/19/2025

Visual Studio Tools for Office (VSTO) Add-ins are great for extending Office to provide solutions for your business or others. They've been around for a long time and there are thousands of solutions built with VSTO. However, they only run on Office on Windows. You can't run VSTO Add-ins on Mac, on the web, or on mobile platforms.

> **Important**
>
> COM and VSTO add-ins aren't supported in the new Outlook on Windows. These add-ins are still supported in the classic Outlook on Windows desktop client. To learn more, see Develop Outlook add-ins for new Outlook on Windows.

Office Add-ins use HTML, JavaScript, and additional web technologies to build Office solutions on all platforms. Migrating your existing VSTO Add-in to an Office Add-in is a great way to make your solution available across all platforms.

You may want to maintain both your VSTO Add-in and a new Office Add-in that both have the same functionality. This enables you to continue servicing your customers that use the VSTO Add-in on Office on Windows. This also enables you to provide the same functionality in an Office Add-in for customers across all platforms. You can also Make your Office Add-in compatible with the existing VSTO Add-in.

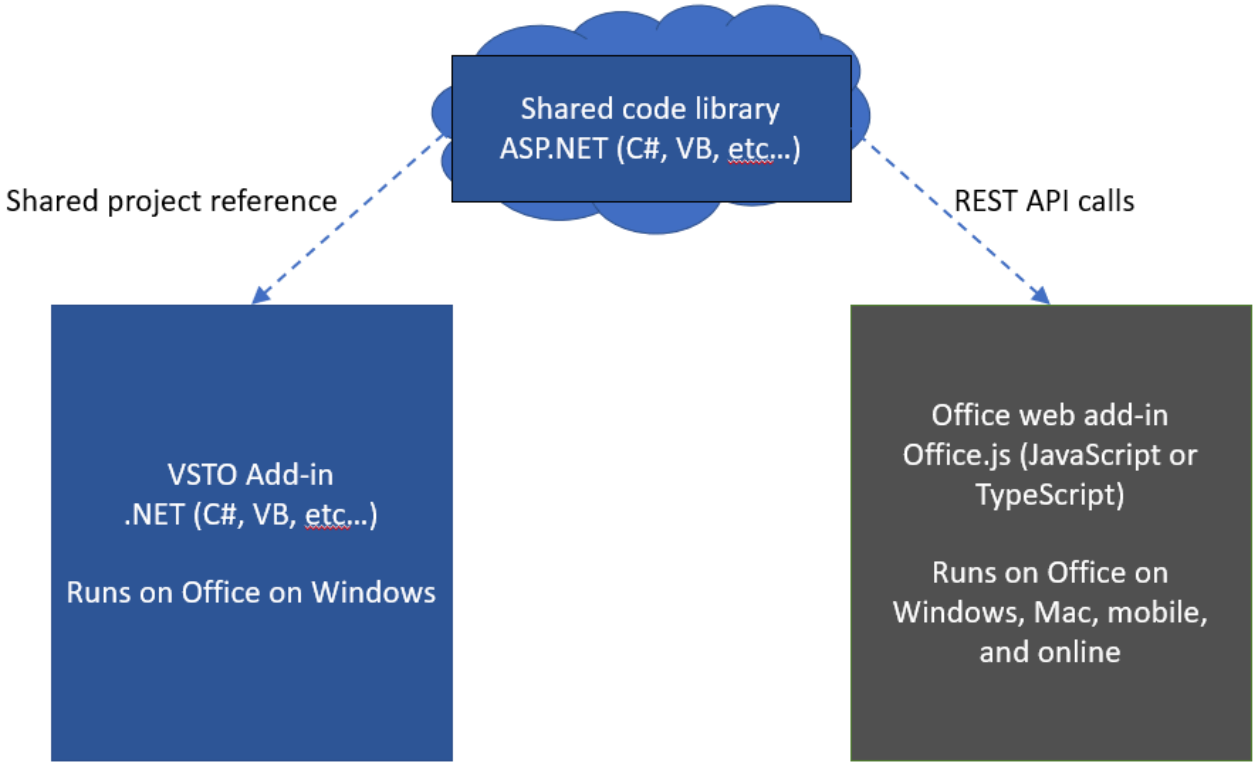
However, it's best to avoid rewriting all the code from your VSTO Add-in for the Office Add-in. This tutorial shows how to avoid rewriting code by using a shared code library for both add-ins.

Shared code library

This tutorial walks you through the steps of identifying and sharing common code between your VSTO Add-in and a modern Office Add-in. It uses a very simple VSTO Add-in example for the steps so that you can focus on the skills and techniques you'll need to work with your own VSTO Add-ins.

The following diagram shows how the shared code library works for migration. Common code is refactored into a new shared code library. The code can remain written in its original language, such as C# or VB. This means you can continue using the code in the existing VSTO

Add-in by creating a project reference. When you create the Office Add-in, it will also use the shared code library by calling into it through REST APIs.



Skills and techniques in this tutorial:

- Create a shared class library by refactoring code into a .NET class library.
- Create a REST API wrapper using ASP.NET Core for the shared class library.
- Call the REST API from the Office Add-in to access shared code.

Prerequisites

To set up your development environment:

1. Install Visual Studio 2022 .
2. Install the following workloads.

    - ASP.NET and web development
    - .NET Core cross-platform development
    - Office/SharePoint development
    - The following **Individual** components.
        - Visual Studio Tools for Office (VSTO)
        - .NET Core 9.0 Runtime

You also need the following:

- A Microsoft 365 account. You might qualify for a Microsoft 365 E5 developer subscription, which includes Office apps, through the Microsoft 365 Developer Program; for details, see the FAQ. Alternatively, you can sign up for a 1-month free trial or purchase a Microsoft 365 plan.
- A Microsoft Azure Tenant. A trial subscription can be acquired here: Microsoft Azure.

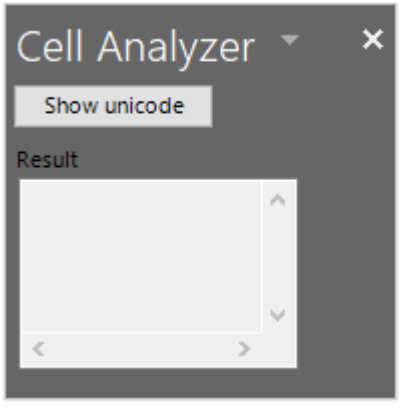
The Cell analyzer VSTO Add-in

This tutorial uses the VSTO Add-in shared library for Office Add-in PnP solution. The **/start** folder contains the VSTO Add-in solution that you'll migrate. Your goal is to migrate the VSTO Add-in to a modern Office Add-in by sharing code when possible.

> **Note**
>
> The sample uses C#, but you can apply the techniques in this tutorial to a VSTO Add-in written in any .NET language.

1. Download the VSTO Add-in shared library for Office Add-in sample to a working folder on your computer.
2. Start Visual Studio and open the **/start/Cell-Analyzer.sln** solution.
3. On the **Debug** menu, choose **Start Debugging**.

The add-in is a custom task pane for Excel. You can select any cell with text, and then choose the **Show unicode** button. In the **Result** section, the add-in displays a list of each character in the text along with its corresponding Unicode number.



Analyze types of code in the VSTO Add-in

The first technique to apply is to analyze the add-in for which parts of code can be shared. In general, the project breaks down into three types of code.

UI code

UI code interacts with the user. In VSTO UI code works through Windows Forms. Office Add-ins use HTML, CSS, and JavaScript for UI. Because of these differences, you can't share UI code with the Office Add-in. The UI needs to be recreated in JavaScript.

Document code

In VSTO, code interacts with the document through .NET objects, such as `Microsoft.Office.Interop.Excel.Range`. However, Office Add-ins use the Office JavaScript library (also called Office.js). Although these are similar, they aren't exactly the same. So again, you can't share document interaction code with the Office Add-in.

Logic code

Business logic, algorithms, helper functions, and similar code often make up the heart of a VSTO Add-in. This code works independently of the UI and document code to perform analysis, connect to backend services, run calculations, and more. This is the code that can be shared so that you don't have to rewrite it in JavaScript.

Let's examine the VSTO Add-in. In the following code, each section is identified as DOCUMENT, UI, or ALGORITHM code.

C#

```
// *** UI CODE ***
private void btnUnicode_Click(object sender, EventArgs e)
{
    // *** DOCUMENT CODE ***
    Microsoft.Office.Interop.Excel.Range rangeCell;
    rangeCell = Globals.ThisAddIn.Application.ActiveCell;

    string cellValue = "";

    if (null != rangeCell.Value)
    {
        cellValue = rangeCell.Value.ToString();
    }

    // *** ALGORITHM CODE ***
    //convert string to Unicode listing
    string result = "";
    foreach (char c in cellValue)
    {
        int unicode = c;

        result += $"{c}: {unicode}\r\n";
```

```
    }

    // *** UI CODE ***
    //Output the result
    txtResult.Text = result;
}
```

Using this approach, you can see that one section of code can be shared with the Office Add-in. The following code needs to be refactored into a separate class library.

C#

```
// *** ALGORITHM CODE ***
//convert string to Unicode listing
string result = "";
foreach (char c in cellValue)
{
    int unicode = c;

    result += $"{c}: {unicode}\r\n";
}
```

Create a shared class library

VSTO Add-ins are created in Visual Studio as .NET projects, so we'll reuse .NET as much as possible to keep things simple. Our next technique is to create a class library and refactor shared code into that class library.

1. If you haven't already, start Visual Studio 2019 and open the **\start\Cell-Analyzer.sln** solution.

2. Right-click (or select and hold) the solution in **Solution Explorer** and choose **Add** > **New Project**.

3. In the **Add a new project dialog**, choose **Class Library (.NET Framework)**, and choose **Next**.

   > **Note**
   >
   > Don't use the .NET Core class library because it won't work with your VSTO project.

4. In the **Configure your new project** dialog, set the following fields.

   - Set the **Project name** to **CellAnalyzerSharedLibrary**.
   - Leave the **Location** at its default value.

- Set the **Framework** to **4.7.2**.

5. Choose **Create**.

6. After the project is created, rename the **Class1.cs** file to **CellOperations.cs**. A prompt to rename the class appears. Rename the class name so that it matches the file name.

7. Add the following code to the `CellOperations` class to create a method named `GetUnicodeFromText`.

```
C#

public class CellOperations
{
    static public string GetUnicodeFromText(string value)
    {
        string result = "";
        foreach (char c in value)
        {
            int unicode = c;

            result += $"{c}: {unicode}\r\n";
        }
        return result;
    }
}
```

Use the shared class library in the VSTO Add-in

Now you need to update the VSTO Add-in to use the class library. This is important that both the VSTO Add-in and Office Add-in use the same shared class library so that future bug fixes or features are made in one location.

1. In **Solution Explorer**, right-click (or select and hold) the **Cell-Analyzer** project and choose **Add Reference**.

2. Select **CellAnalyzerSharedLibrary**, and choose **OK**.

3. In **Solution Explorer**, expand the **Cell-Analyzer** project, right-click (or select and hold) the **CellAnalyzerPane.cs** file and choose **View Code**.

4. In the `btnUnicode_Click` method, delete the following lines of code.

```
C#

//Convert to Unicode listing
string result = "";
```

```
foreach (char c in cellValue)
{
  int unicode = c;
  result += $"{c}: {unicode}\r\n";
}
```

5. Update the line of code under the `//Output the result` comment to read as follows:

```
C#

//Output the result
txtResult.Text =
CellAnalyzerSharedLibrary.CellOperations.GetUnicodeFromText(cellValue);
```

6. On the **Debug** menu, choose **Start Debugging**. The custom task pane should work as expected. Enter some text in a cell, and then test that you can convert it to a Unicode list with the add-in.

Create a REST API wrapper

The VSTO Add-in can use the shared class library directly since they are both .NET projects. However the Office Add-in won't be able to use .NET since it uses JavaScript. Next, you'll create a REST API wrapper. This enables the Office Add-in to call a REST API, which then passes the call along to the shared class library.

1. In **Solution Explorer**, right-click (or select and hold) the **Cell-Analyzer** project and choose **Add** > **New Project**.

2. In the **Add a new project dialog**, choose **ASP.NET Core Web Application**, and choose **Next**.

3. In the **Configure your new project** dialog, set the following fields.

   - Set the **Project name** to **CellAnalyzerRESTAPI**.
   - In the **Location** field, leave the default value.

4. Choose **Create**.

5. In the **Create a new ASP.NET Core web application** dialog, select **ASP.NET Core 3.1** for the version, and select **API** in the list of projects.

6. Leave all other fields at default values and choose the **Create** button.

7. After the project is created, expand the **CellAnalyzerRESTAPI** project in **Solution Explorer**.

8. Right-click (or select and hold) **Dependencies** and choose **Add Reference**.
9. Select **CellAnalyzerSharedLibrary**, and choose **OK**.
10. Right-click (or select and hold) the **Controllers** folder and choose **Add** > **Controller**.
11. In the **Add New Scaffolded Item** dialog, choose **API Controller - Empty**, then choose **Add**.
12. In the **Add Empty API Controller** dialog, name the controller **AnalyzeUnicodeController**, then choose **Add**.
13. Open the **AnalyzeUnicodeController.cs** file and add the following code as a method to the `AnalyzeUnicodeController` class.

    C#

    ```
    [HttpGet]
    public ActionResult<string> AnalyzeUnicode(string value)
    {
      if (value == null)
      {
        return BadRequest();
      }
      return CellAnalyzerSharedLibrary.CellOperations.GetUnicodeFromText(value);
    }
    ```

14. Right-click (or select and hold) the **CellAnalyzerRESTAPI** project and choose **Set as Startup Project**.
15. On the **Debug** menu, choose **Start Debugging**.
16. A browser will launch. Enter the following URL to test that the REST API is working: `https://localhost:<ssl port number>/api/analyzeunicode?value=test`. You can reuse the port number from the URL in the browser that Visual Studio launched. You should see a string returned with Unicode values for each character.

Create the Office Add-in

When you create the Office Add-in, it will make a call to the REST API. But first, you need to get the port number of the REST API server and save it for later.

Save the SSL port number

1. If you haven't already, start Visual Studio 2019, and open the **\start\Cell-Analyzer.sln** solution.
2. In the **CellAnalyzerRESTAPI** project, expand **Properties**, and open the **launchSettings.json** file.
3. Find the line of code with the **sslPort** value, copy the port number, and save it somewhere.

Add the Office Add-in project

To keep things simple, keep all the code in one solution. Add the Office Add-in project to the existing Visual Studio solution. However, if you're familiar with the Yeoman generator for Office Add-ins and Visual Studio Code, you can also run `yo office` to build the project. The steps are very similar.

1. In **Solution Explorer**, right-click (or select and hold) the **Cell-Analyzer** solution and choose **Add** > **New Project**.
2. In the **Add a new project dialog**, choose **Excel Web Add-in**, and choose **Next**.
3. In the **Configure your new project** dialog, set the following fields.

    - Set the **Project name** to **CellAnalyzerOfficeAddin**.
    - Leave the **Location** at its default value.
    - Set the **Framework** to **4.7.2** or later.

4. Choose **Create**.
5. In the **Choose the add-in type** dialog, select **Add new functionalities to Excel**, and choose **Finish**.

Two projects will be created:

- **CellAnalyzerOfficeAddin** - This project configures the manifest XML files that describes the add-in so Office can load it correctly. It contains the ID, name, description, and other information about the add-in.
- **CellAnalyzerOfficeAddinWeb** - This project contains web resources for your add-in, such as HTML, CSS, and scripts. It also configures an IIS Express instance to host your add-in as a web application.

Add UI and functionality to the Office Add-in

1. In **Solution Explorer**, expand the **CellAnalyzerOfficeAddinWeb** project.

2. Open the **Home.html** file, and replace the `<body>` contents with the following HTML.

    ```
    HTML
    ```

```
<button id="btnShowUnicode" onclick="showUnicode()">Show Unicode</button>
<p>Result:</p>
<div id="txtResult"></div>
```

3. Open the **Home.js** file, and replace the entire contents with the following code.

JavaScript

```
(function () {
  "use strict";
  // The initialize function must be run each time a new page is loaded.
  Office.initialize = function (reason) {
    $(document).ready(function () {
    });
  };
})();

function showUnicode() {
  Excel.run(function (context) {
    const range = context.workbook.getSelectedRange();
    range.load("values");
    return context.sync(range).then(function (range) {
      const url = "https://localhost:<ssl port number>/api/analyzeunicode?
value=" + range.values[0][0];
      $.ajax({
        type: "GET",
        url: url,
        success: function (data) {
          let htmlData = data.replace(/\r\n/g, '<br>');
          $("#txtResult").html(htmlData);
        },
        error: function (data) {
            $("#txtResult").html("error occurred in ajax call.");
        }
      });
    });
  });
}
```

4. In the previous code, enter the **sslPort** number you saved previously from the **launchSettings.json** file.

In the previous code, the returned string will be processed to replace carriage return line feeds with `<br>` HTML tags. You may occasionally run into situations where a return value that works perfectly fine for .NET in the VSTO Add-in will need to be adjusted on the Office Add-in side to work as expected. In this case, the REST API and shared class library are only concerned with returning the string. The `showUnicode()` function is responsible for formatting return values correctly for presentation.

Allow CORS from the Office Add-in

The Office.js library requires CORS on outgoing calls, such as the one made from the `ajax` call to the REST API server. Use the following steps to allow calls from the Office Add-in to the REST API.

1. In **Solution Explorer**, select the **CellAnalyzerOfficeAddinWeb** project.
2. From the **View** menu, choose **Properties Window**, if the window isn't already displayed.
3. In the properties window, copy the value of the **SSL URL**, and save it somewhere. This is the URL that you need to allow through CORS.
4. In the **CellAnalyzerRESTAPI** project, open the **Startup.cs** file.
5. Add the following code to the top of the `ConfigureServices` method. Be sure to substitute the URL SSL you copied previously for the `builder.WithOrigins` call.

```
C#

services.AddCors(options =>
{
  options.AddPolicy(MyAllowSpecificOrigins,
  builder =>
  {
    builder.WithOrigins("<your URL SSL>")
    .AllowAnyMethod()
    .AllowAnyHeader();
  });
});
```

> ⓘ Note
>
> Leave the trailing `/` from the end of the URL when you use it in the `builder.WithOrigins` method. For example, it should appear similar to `https://localhost:44000`. Otherwise, you'll get a CORS error at runtime.

6. Add the following field to the `Startup` class.

```
C#

readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";
```

7. Add the following code to the `configure` method just before the line of code for `app.UseEndpoints`.

```
C#

app.UseCors(MyAllowSpecificOrigins);
```

When done, your `Startup` class should look similar to the following code (your localhost URL may be different).

```
C#

public class Startup
{
  public Startup(IConfiguration configuration)
    {
      Configuration = configuration;
    }

    readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

    public IConfiguration Configuration { get; }

    // NOTE: The following code configures CORS for the localhost:44397 port.
    // This is for development purposes. In production code, you should update
this to
    // use the appropriate allowed domains.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(MyAllowSpecificOrigins,
            builder =>
            {
                builder.WithOrigins("https://localhost:44397")
                .AllowAnyMethod()
                .AllowAnyHeader();
            });
        });
        services.AddControllers();
    }

    // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
```

```
            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseCors(MyAllowSpecificOrigins);

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
```

Run the add-in

1. In **Solution Explorer**, right-click (or select and hold) the top node **Solution 'Cell-Analyzer'** and choose **Set Startup Projects**.
2. In the **Solution 'Cell-Analyzer' Property Pages** dialog, select **Multiple startup projects**.
3. Set the **Action** property to **Start** for each of the following projects.
    - CellAnalyzerRESTAPI
    - CellAnalyzerOfficeAddin
    - CellAnalyzerOfficeAddinWeb
4. Choose **OK**.
5. From the **Debug** menu, choose **Start Debugging**.

Excel will run and sideload the Office Add-in. You can test that the localhost REST API service is working correctly by entering a text value into a cell, and choosing the **Show Unicode** button in the Office Add-in. It should call the REST API and display the unicode values for the text characters.

Publish to an Azure App Service

You eventually want to publish the REST API project to the cloud. In the following steps you'll see how to publish the **CellAnalyzerRESTAPI** project to a Microsoft Azure App Service. See Prerequisites for information on how to get an Azure account.

1. In **Solution Explorer**, right-click (or select and hold) the **CellAnalyzerRESTAPI** project and choose **Publish**.
2. In the **Pick a publish target** dialog, select **Create New**, and choose **Create Profile**.

3. In the **App Service** dialog, select the correct account, if it isn't already selected.
4. The fields for the **App Service** dialog will be set to defaults for your account. Generally, the defaults work fine, but you can change them if you prefer different settings.
5. In the **App Service** dialog, choose **Create**.
6. The new profile will be displayed in a **Publish** page. Choose **Publish** to build and deploy the code to the App Service.

You can now test the service. Open a browser and enter a URL that goes directly to the new service. For example, use `https://<myappservice>.azurewebsites.net/api/analyzeunicode?value=test`, where *myappservice* is the unique name you created for the new App Service.

Use the Azure App Service from the Office Add-in

The final step is to update the code in the Office Add-in to use the Azure App Service instead of localhost.

1. In **Solution Explorer**, expand the **CellAnalyzerOfficeAddinWeb** project, and open the **Home.js** file.

2. Change the `url` constant to use the URL for your Azure App Service as shown in the following line of code. Replace `<myappservice>` with the unique name you created for the new App Service.

   JavaScript

   ```
   const url = "https://<myappservice>.azurewebsites.net/api/analyzeunicode?value=" + range.values[0][0];
   ```

3. In **Solution Explorer**, right-click (or select and hold) the top node **Solution 'Cell-Analyzer'** and choose **Set Startup Projects**.

4. In the **Solution 'Cell-Analyzer' Property Pages** dialog, select **Multiple startup projects**.

5. Enable the **Start** action for each of the following projects.

   - CellAnalyzerOfficeAddinWeb
   - CellAnalyzerOfficeAddin

6. Choose **OK**.

7. From the **Debug** menu, choose **Start Debugging**.

Excel will run and sideload the Office Add-in. To test that the App Service is working correctly, enter a text value into a cell, and choose **Show Unicode** in the Office Add-in. It should call the

service and display the unicode values for the text characters.

Conclusion

In this tutorial, you learned how to create an Office Add-in that uses shared code with the original VSTO add-in. You learned how to maintain both VSTO code for Office on Windows, and an Office Add-in for Office on other platforms. You refactored VSTO C# code into a shared library and deployed it to an Azure App Service. You created an Office Add-in that uses the shared library, so that you don't have to rewrite the code in JavaScript.