

# Excel add-ins overview

Article • 04/11/2023

An Excel add-in allows you to extend Excel application functionality across multiple platforms including Windows, Mac, iPad, and in a browser. Use Excel add-ins within a workbook to:

- Interact with Excel objects, read and write Excel data.
- Extend functionality using web based task pane or content pane
- Add custom ribbon buttons or contextual menu items
- Add custom functions
- Provide richer interaction using dialog window

The Office Add-ins platform provides the framework and Office.js JavaScript APIs that enable you to create and run Excel add-ins. By using the Office Add-ins platform to create your Excel add-in, you'll get the following benefits.

- **Cross-platform support:** Excel add-ins run in Office on the web, Windows, Mac, and iPad.
- **Centralized deployment:** Admins can quickly and easily deploy Excel add-ins to users throughout an organization.
- **Use of standard web technology:** Create your Excel add-in using familiar web technologies such as HTML, CSS, and JavaScript.
- **Distribution via AppSource:** Share your Excel add-in with a broad audience by publishing it to [AppSource](#).

## ⓘ Note

Excel add-ins are different from COM and VSTO add-ins, which are earlier Office integration solutions that run only in Office on Windows. Unlike COM add-ins, Excel add-ins don't require you to install any code on a user's device, or within Excel.

## Components of an Excel add-in

An Excel add-in includes two basic components: a web application and a configuration file, called a manifest file.

The web application uses the [Office JavaScript API](#) to interact with objects in Excel, and can also facilitate interaction with online resources. For example, an add-in can perform any of the following tasks.

- Create, read, update, and delete data in the workbook (worksheets, ranges, tables, charts, named items, and more).
- Perform user authorization with an online service by using the standard OAuth 2.0 flow.
- Issue API requests to Microsoft Graph or any other API.

The web application can be hosted on any web server, and can be built using client-side frameworks (such as Angular, React, jQuery) or server-side technologies (such as ASP.NET, Node.js, PHP).

The [manifest](#) is a configuration file that defines how the add-in integrates with Office clients by specifying settings and capabilities such as:

- The URL of the add-in's web application.
- The add-in's display name, description, ID, version, and default locale.
- How the add-in integrates with Excel, including any custom UI that the add-in creates (ribbon buttons, context menus, and so on).
- Permissions that the add-in requires, such as reading and writing to the document.

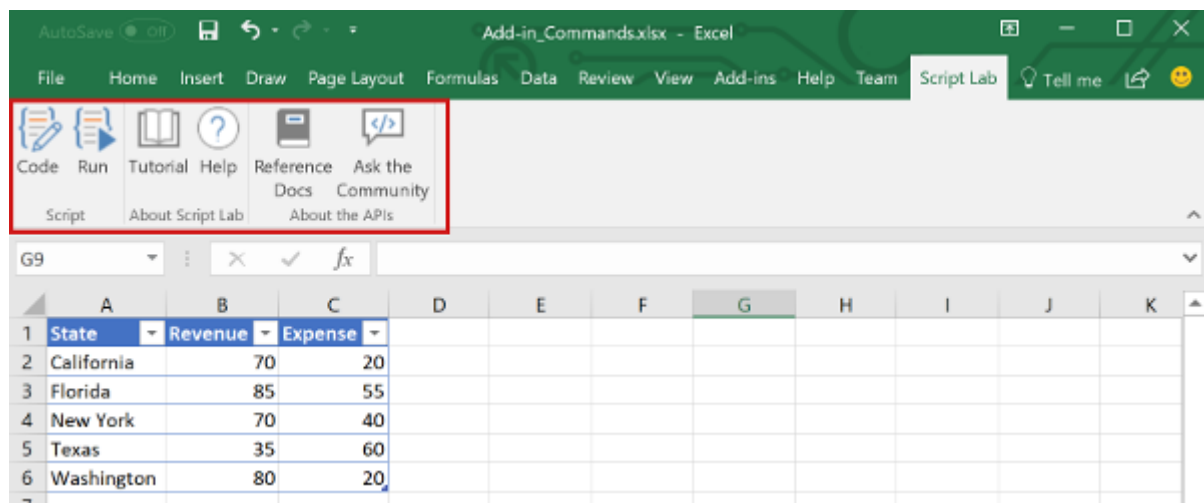
To enable end users to install and use an Excel add-in, you must publish its manifest either to AppSource or to an add-ins catalog. For details about publishing to AppSource, see [Make your solutions available in AppSource and within Office](#).

## Capabilities of an Excel add-in

In addition to interacting with the content in the workbook, Excel add-ins can add custom ribbon buttons or menu commands, insert task panes, add custom functions, open dialog boxes, and even embed rich, web-based objects such as charts or interactive visualizations within a worksheet.

## Add-in commands

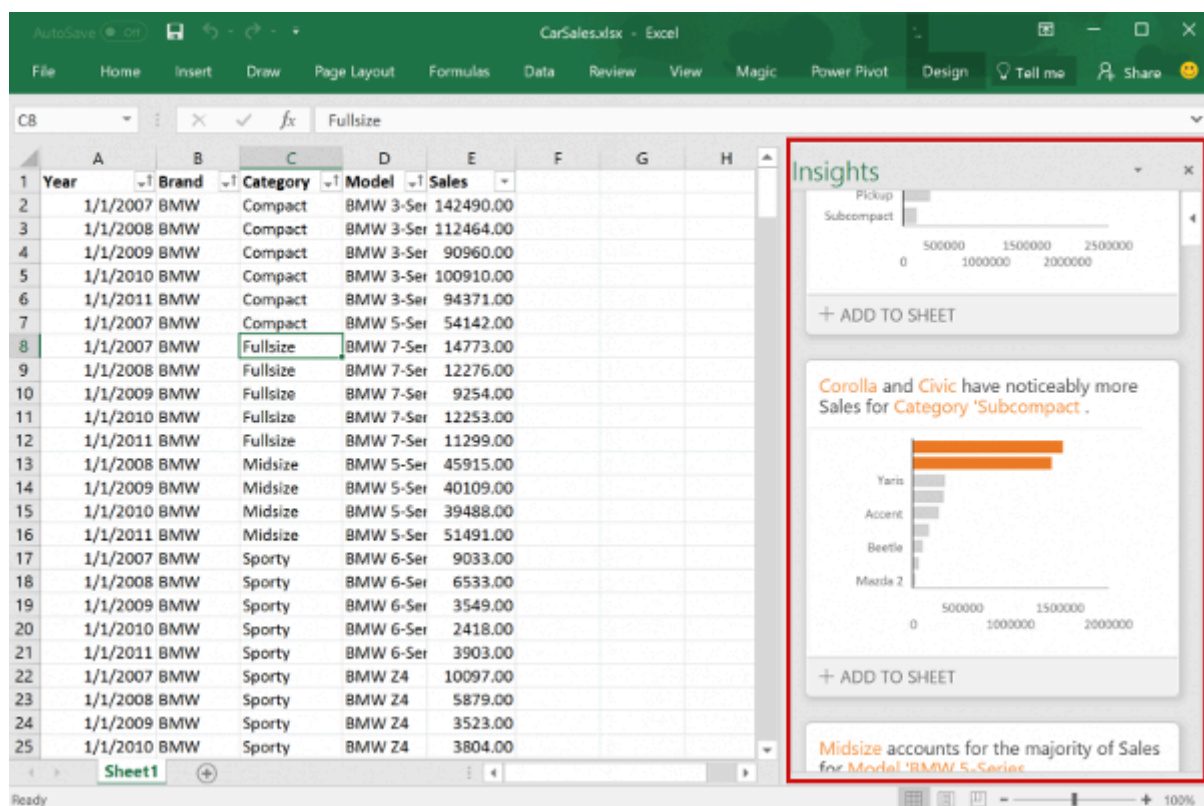
Add-in commands are UI elements that extend the Excel UI and start actions in your add-in. You can use add-in commands to add a button on the ribbon or an item to a context menu in Excel. When users select an add-in command, they initiate actions such as running JavaScript code, or showing a page of the add-in in a task pane.



For more information about command capabilities, supported platforms, and best practices for developing add-in commands, see [Add-in commands for Excel, Word, and PowerPoint](#).

## Task panes

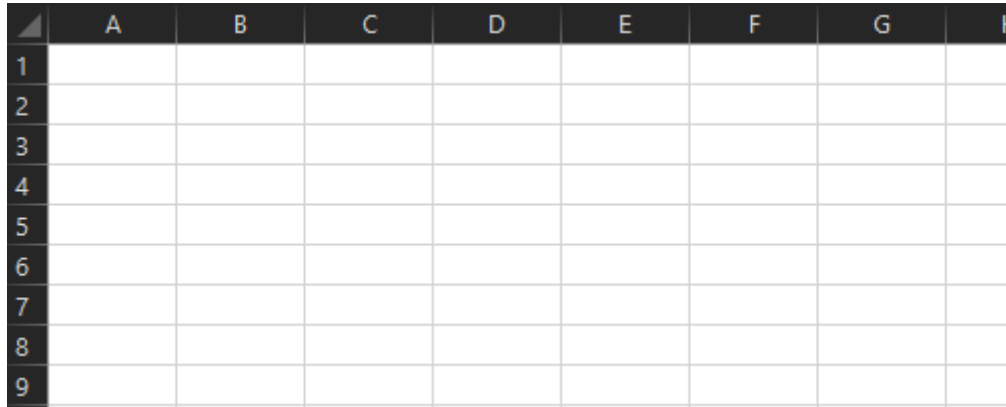
Task panes are interface surfaces that typically appear on the right side of the window within Excel. Task panes give users access to interface controls that run code to modify the Excel document or display data from a data source.



For more information about task panes, see [Task panes in Office Add-ins](#). For a sample that implements a task pane in Excel, see [Excel Add-in JS WoodGrove Expense Trends](#).

## Custom functions

Custom functions enable developers to add new functions to Excel by defining those functions in JavaScript as part of an add-in. Users within Excel can access custom functions just as they would any native function in Excel, such as `SUM()`.

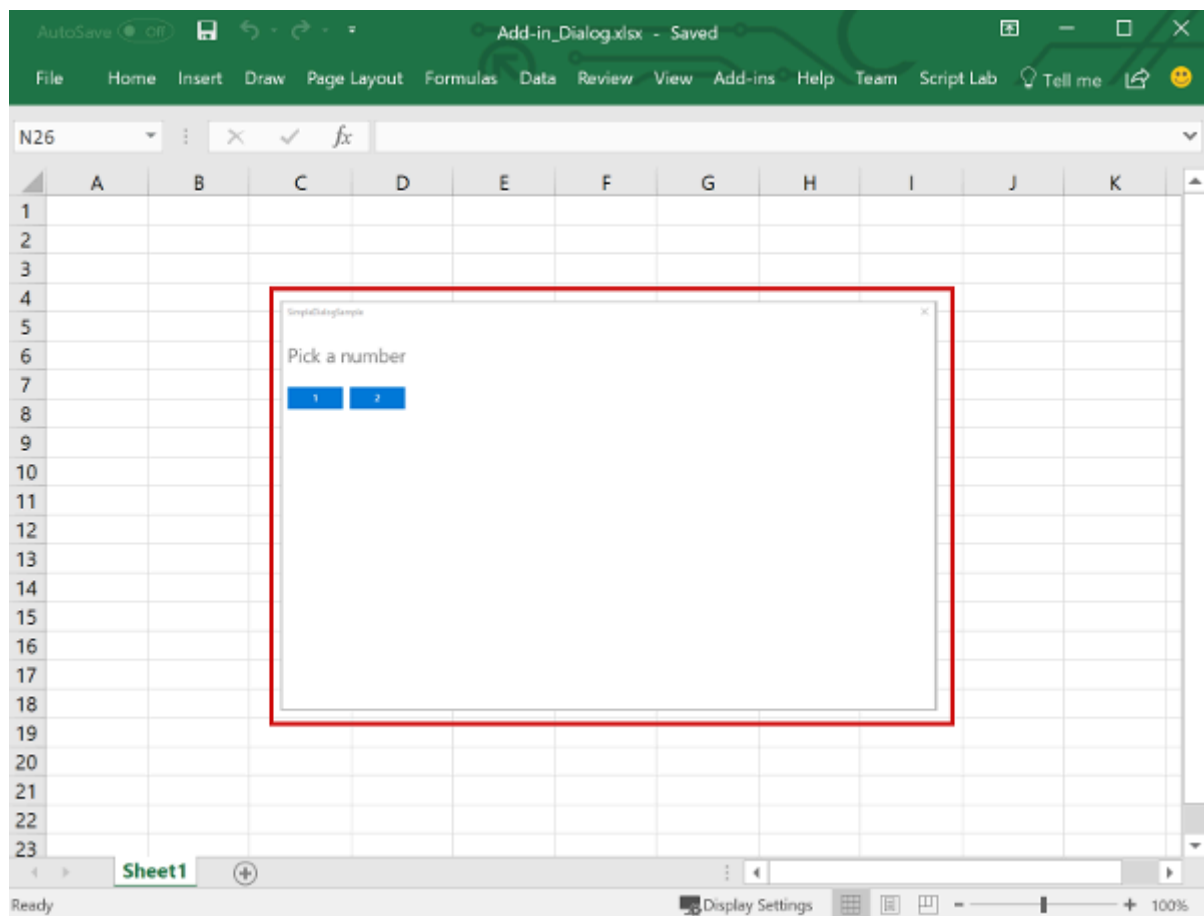


	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6								
7								
8								
9								

For more information about custom functions, see [Create custom functions in Excel](#).

## Dialog boxes

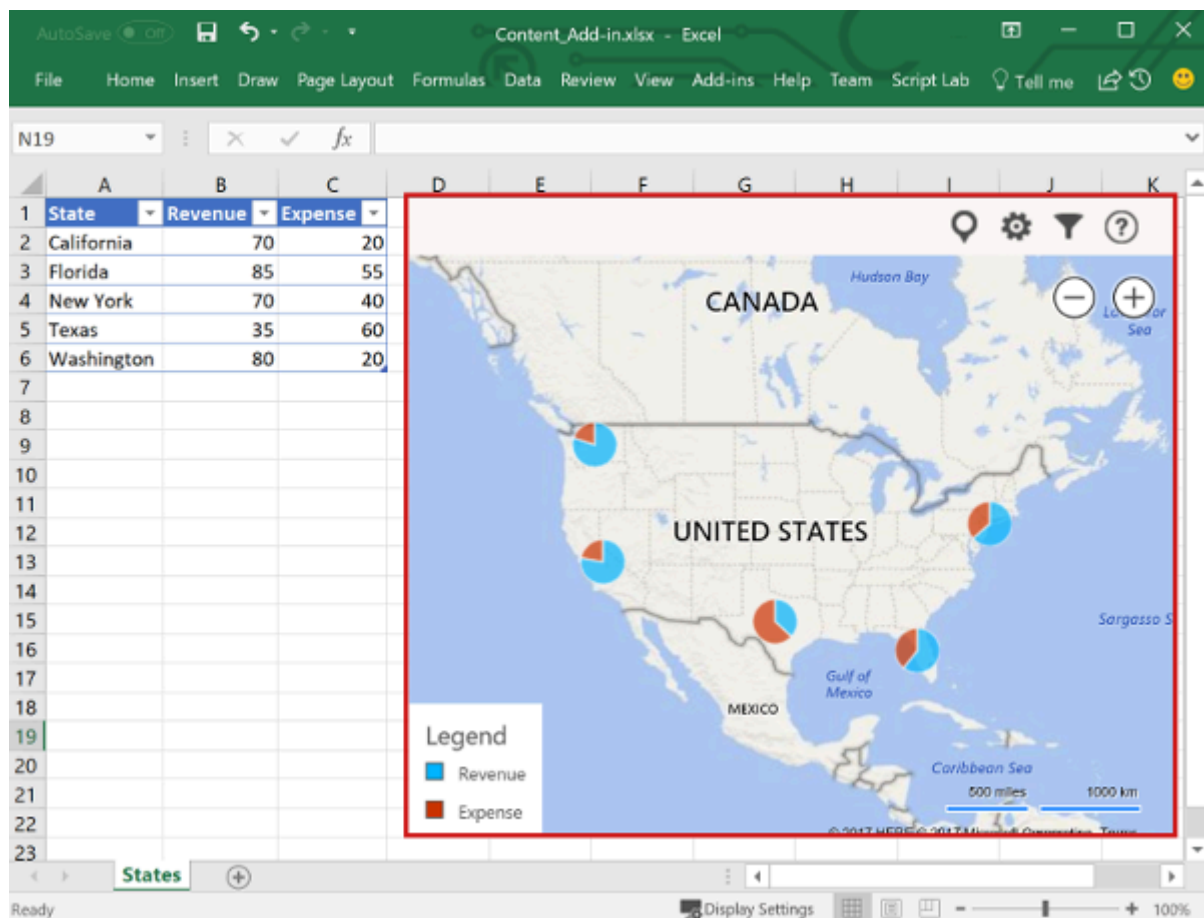
Dialog boxes are surfaces that float above the active Excel application window. You can use dialog boxes for tasks such as displaying sign-in pages that can't be opened directly in a task pane, requesting that the user confirm an action, or hosting videos that might be too small if confined to a task pane. To open dialog boxes in your Excel add-in, use the [Dialog API](#).



For more information about dialog boxes and the Dialog API, see [Use the Dialog API in your Office Add-ins](#).

## Content add-ins

Content add-ins are surfaces that you can embed directly into Excel documents. You can use content add-ins to embed rich, web-based objects such as charts, data visualizations, or media into a worksheet or to give users access to interface controls that run code to modify the Excel document or display data from a data source. Use content add-ins when you want to embed functionality directly into the document.



For more information about content add-ins, see [Content Office Add-ins](#). For a sample that implements a content add-in in Excel, see [Excel Content Add-in Humongous Insurance](#) in GitHub.

## JavaScript APIs to interact with workbook content

An Excel add-in interacts with objects in Excel by using the [Office JavaScript API](#), which includes two JavaScript object models:

- **Excel JavaScript API:** Introduced with Office 2016, the [Excel JavaScript API](#) provides strongly-typed Excel objects that you can use to access worksheets, ranges, tables, charts, and more.
- **Common API:** Introduced with Office 2013, the Common API enables you to access features such as UI, dialogs, and client settings that are common across multiple types of Office applications. The limited functionality for Excel interaction in the Common API has been replaced by the Excel JavaScript API.

## Next steps

Get started by [creating your first Excel add-in](#). Then, learn about the [core concepts](#) of building Excel add-ins.

## See also

- [Office Add-ins platform overview](#)
- [Learn about Microsoft 365 Developer Program](#)
- [Developing Office Add-ins](#)
- [Excel JavaScript object model in Office Add-ins](#)
- [Excel JavaScript API reference](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

### Office Add-ins feedback

Office Add-ins is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

# Build an Excel task pane add-in

05/07/2025

In this article, you'll walk through the process of building an Excel task pane add-in. You'll use either the Office Add-ins Development Kit or the Yeoman generator to create your Office Add-in. Select the tab for the one you'd like to use and then follow the instructions to create your add-in and test it locally. If you'd like to create the add-in project within Visual Studio Code, we recommend the Office Add-ins Development Kit.

Office Add-ins Development Kit

## Prerequisites

- Download and install [Visual Studio Code](#).
- Node.js (the latest LTS version). Visit the [Node.js site](#) to download and install the right version for your operating system. To verify if you've already installed these tools, run the commands `node -v` and `npm -v` in your terminal.
- Office connected to a Microsoft 365 subscription. You might qualify for a Microsoft 365 E5 developer subscription through the [Microsoft 365 Developer Program](#), see [FAQ](#) for details. Alternatively, you can [sign up for a 1-month free trial](#) or [purchase a Microsoft 365 plan](#).

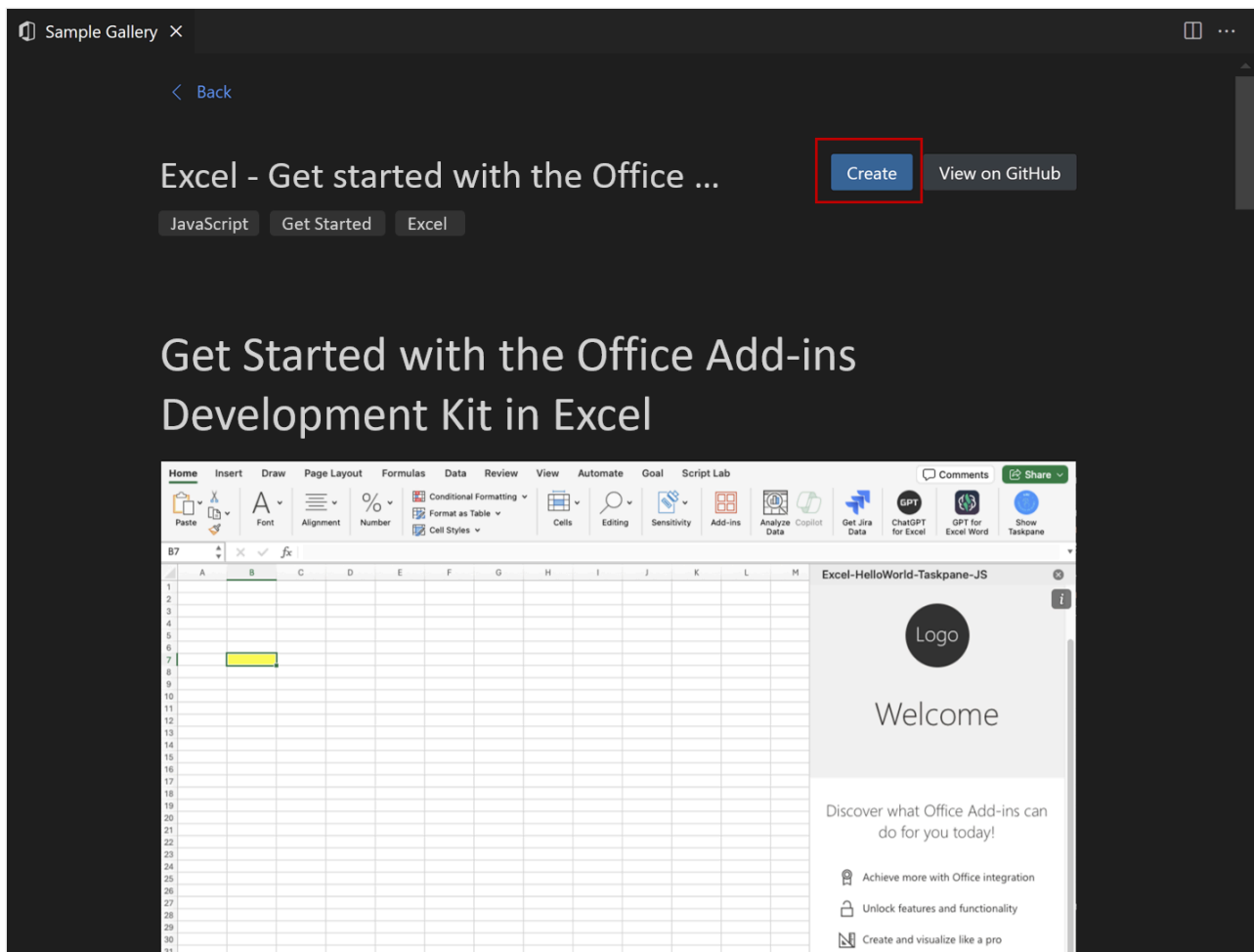
## Create the add-in project

Click the following button to create an add-in project using the Office Add-ins Development Kit for Visual Studio Code. You'll be prompted to install the extension if don't already have it. A page that contains the project description will open in Visual Studio Code.

Create an add-in in Visual Studio Code

In the prompted page, select **Create** to create the add-in project. In the **Workspace folder** dialog that opens, select the folder where you want to create the project.





The Office Add-ins Development Kit will create the project. It will then open the project in a *second* Visual Studio Code window. Close the original Visual Studio Code window.

#### ! Note

If you use VSCode Insiders, or you have problems opening the project page in VSCode, install the extension manually by following [these steps](#), and find the sample in the sample gallery.

## Explore the project

The add-in project that you've created with the Office Add-ins Development Kit contains sample code for a basic task pane add-in. If you'd like to explore the components of your add-in project, open the project in your code editor and review the files listed below. When you're ready to try out your add-in, proceed to the next section.

1. The `./manifest.xml` or `./manifest.json` file in the root directory of the project defines the settings and capabilities of the add-in.
2. The `./src/taskpane/taskpane.html` file contains the HTML markup for the task pane.

3. The `./src/taskpane/taskpane.css` file contains the CSS that's applied to content in the task pane.
4. The `./src/taskpane/taskpane.js` file contains the Office JavaScript API code that facilitates interaction between the task pane and the Office client application.

## Try it out

1. Open the extension by selecting the Office Add-ins Development Kit icon in the **Activity Bar**.
2. Select **Preview Your Office Add-in (F5)**
3. In the Quick Pick menu, select the option **{Office Application} Desktop (Edge Chromium)**, where '{Office Application}' is the appropriate application, such as "Excel" or "Word". This will launch the add-in and debug the code.

The development kit checks that the prerequisites are met before debugging starts. Check the terminal for detailed information if there are issues with your environment. After this process, the Office desktop application launches and sideloads the add-in. Please note that the first time you run a project, it may take a few minutes to install the dependencies. You'll need to install the certificate when prompted.

## Stop testing your Office Add-in

Once you are finished testing and debugging the add-in, *always* close the add-in by following these steps. (Closing the Office application or web server window doesn't reliably deregister the add-in.)

1. Open the extension by selecting the Office Add-ins Development Kit icon in the **Activity Bar**.
2. Select **Stop Previewing Your Office Add-in**. This closes the web server and removes the add-in from the registry and cache.
3. Close the Office application window.

## Troubleshooting

If you have problems running the add-in, take these steps.

- Close any open instances of Office.
- Close the previous web server started for the add-in with the **Stop Previewing Your Office Add-in** Office Add-ins Development Kit extension option.

The article [Troubleshoot development errors with Office Add-ins](#) contains solutions to common problems. If you're still having issues, [create a GitHub issue](#) and we'll help you.

For information on running the add-in on Office on the web, see [Sideload Office Add-ins to Office on the web](#).

For information on debugging on older versions of Office, see [Debug add-ins using developer tools in Microsoft Edge Legacy](#).

## Next steps

Congratulations, you've successfully created an Excel task pane add-in! Next, learn more about the capabilities of an Excel add-in and build a more complex add-in by following along with the [Excel add-in tutorial](#).

## Code samples

- [Excel "Hello world" add-in](#): Learn how to build a simple Office Add-in with only a manifest, HTML web page, and a logo.

## See also

- [Office Add-ins platform overview](#)
- [Develop Office Add-ins](#)
- [Excel JavaScript object model in Office Add-ins](#)
- [Excel add-in code samples](#)
- [Excel JavaScript API reference](#)
- [Using Visual Studio Code to publish](#)

# Use React to build an Excel task pane add-in

Article • 02/12/2025

In this article, you'll walk through the process of building an Excel task pane add-in using React and the Excel JavaScript API.

## Prerequisites

- Node.js (the latest LTS version). Visit the [Node.js site](#) to download and install the right version for your operating system.
- The latest version of Yeoman and the Yeoman generator for Office Add-ins. To install these tools globally, run the following command via the command prompt.

command line

```
npm install -g yo generator-office
```

### ⚠ Note

Even if you've previously installed the Yeoman generator, we recommend you update your package to the latest version from npm.

- Office connected to a Microsoft 365 subscription (including Office on the web).

### ⚠ Note

If you don't already have Office, you might qualify for a Microsoft 365 E5 developer subscription 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](#).

## Create the add-in project

Run the following command to create an add-in project using the Yeoman generator. A folder that contains the project will be added to the current directory.

command line

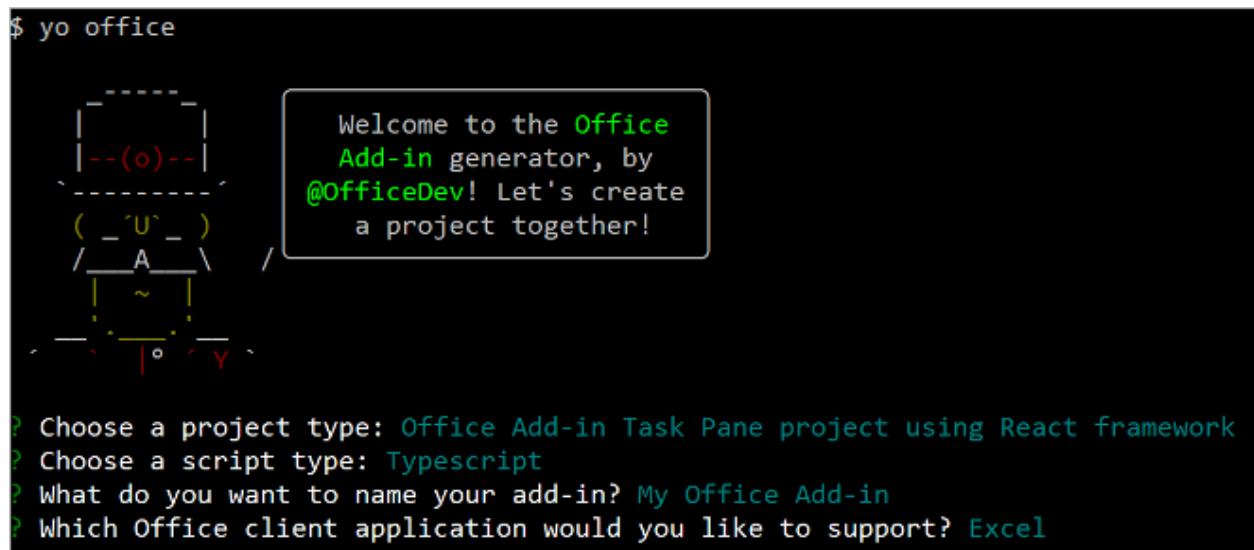
```
yo office
```

### ⓘ Note

When you run the `yo office` command, you may receive prompts about the data collection policies of Yeoman and the Office Add-in CLI tools. Use the information that's provided to respond to the prompts as you see fit.

When prompted, provide the following information to create your add-in project.

- **Choose a project type:** Office Add-in Task Pane project using React framework
- **Choose a script type:** TypeScript
- **What do you want to name your add-in?** My Office Add-in
- **Which Office client application would you like to support?** Excel



```
$ yo office

  ____
  |  (o)  |
  |_____|
  |  ^U^  |
  |   A   |
  |  ~    |
  |_____|
  |  o    |
  |_____|

Welcome to the Office
Add-in generator, by
@OfficeDev! Let's create
a project together!

? Choose a project type: Office Add-in Task Pane project using React framework
? Choose a script type: Typescript
? What do you want to name your add-in? My Office Add-in
? Which Office client application would you like to support? Excel
```

After you complete the wizard, the generator creates the project and installs supporting Node components.

## Explore the project

The add-in project that you've created with the Yeoman generator contains sample code for a basic task pane add-in. If you'd like to explore the key components of your add-in project, open the project in your code editor and review the files listed below. When you're ready to try out your add-in, proceed to the next section.

- The `./manifest.xml` or `manifest.json` file in the root directory of the project defines the settings and capabilities of the add-in.

- The `./src/taskpane/taskpane.html` file defines the HTML framework of the task pane, and the files within the `./src/taskpane/components` folder define the various parts of the task pane UI.
- The `./src/taskpane/taskpane.css` file contains the CSS that's applied to content in the task pane.
- The `./src/taskpane/components/App.tsx` file contains the Office JavaScript API code that facilitates interaction between the task pane and Excel.

## Try it out

1. Navigate to the root folder of the project.

command line

```
cd "My Office Add-in"
```

2. Complete the following steps to start the local web server and sideload your add-in.

### ⓘ Note

- Office Add-ins should use HTTPS, not HTTP, even while you're developing. If you're prompted to install a certificate after you run one of the following commands, accept the prompt to install the certificate that the Yeoman generator provides. You may also have to run your command prompt or terminal as an administrator for the changes to be made.
- If this is your first time developing an Office Add-in on your machine, you may be prompted in the command line to grant Microsoft Edge WebView a loopback exemption ("Allow localhost loopback for Microsoft Edge WebView?"). When prompted, enter **y** to allow the exemption. Note that you'll need administrator privileges to allow the exemption. Once allowed, you shouldn't be prompted for an exemption when you sideload Office Add-ins in the future (unless you remove the exemption from your machine). To learn more, see ["We can't open this add-in from localhost" when loading an Office Add-in or using Fiddler.](#)

```
> office-addin-taskpane-js@0.0.1 start
> office-addin-debugging start manifest.xml

Debugging is being started...
App type: desktop
? Allow localhost loopback for Microsoft Edge WebView? (Y/n) Y
```

### 💡 Tip

If you're testing your add-in on Mac, run the following command before proceeding. When you run this command, the local web server starts.

command line

```
npm run dev-server
```

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on a browser, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ⚠ Note

If you are developing on a Mac, enclose the {url} in single quotation marks. Do *not* do this on Windows.

command line

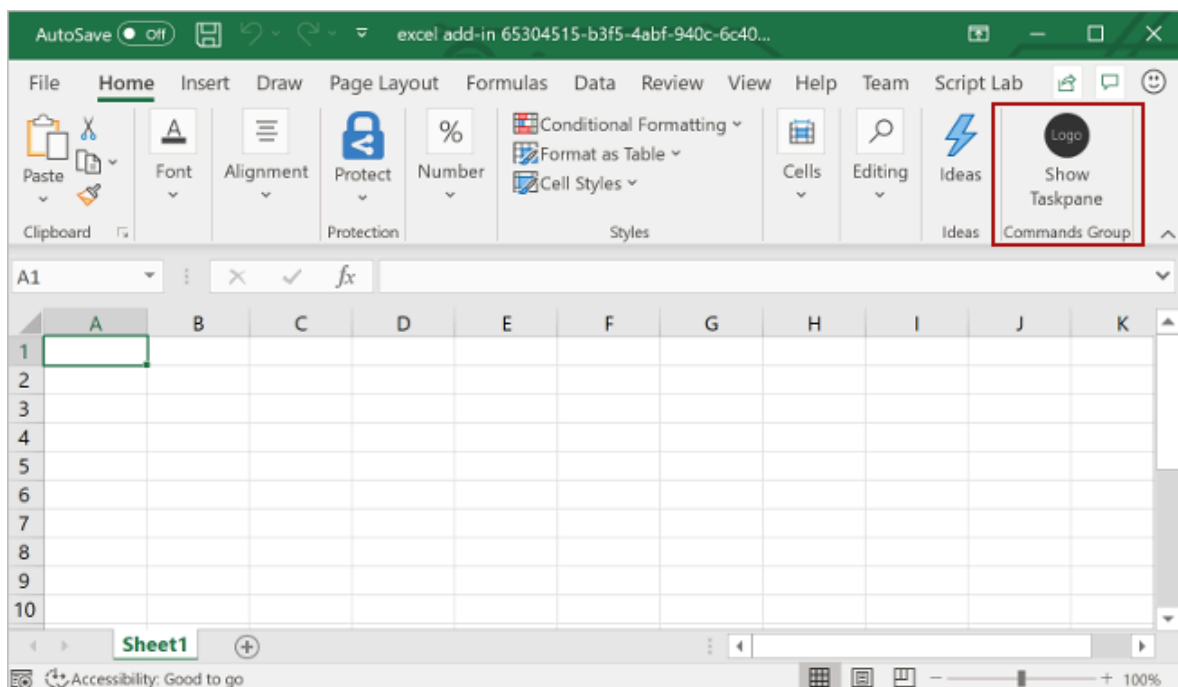
```
npm run start -- web --document {url}
```

The following are examples.

- `npm run start -- web --document https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkcH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-.df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii2lDr_oQ?e=RSccmNP`

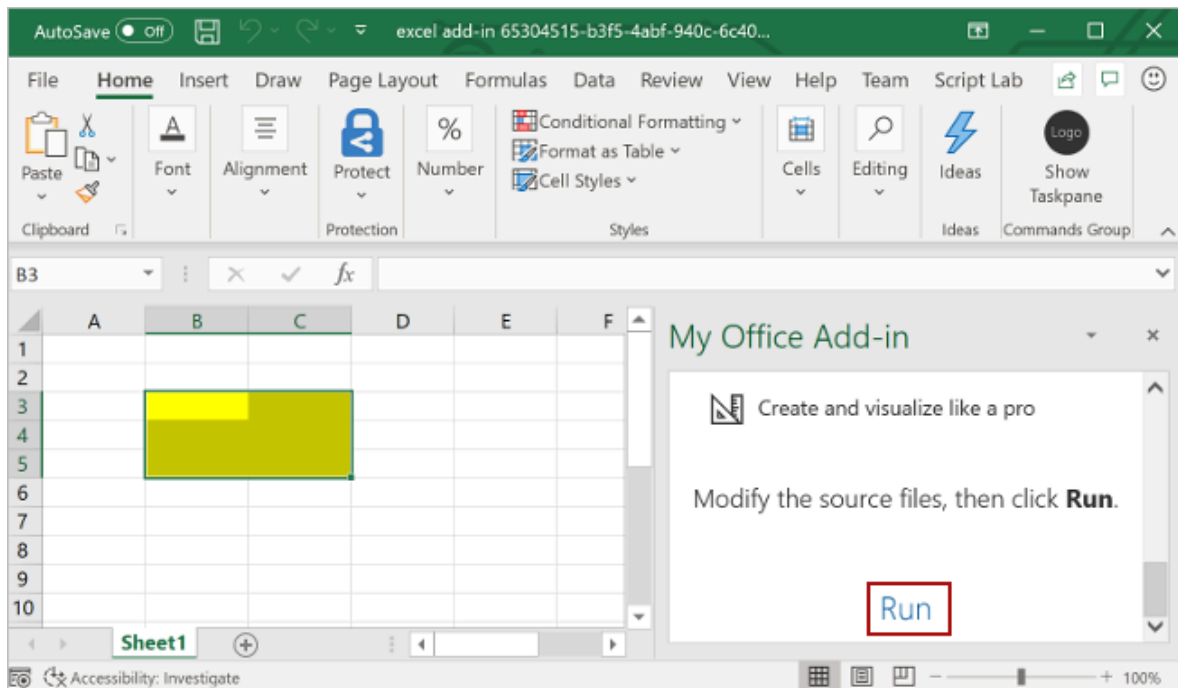
If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

3. In Excel, choose the **Home** tab, and then choose the **Show Taskpane** button on the ribbon to open the add-in task pane.



4. Select any range of cells in the worksheet.
5. At the bottom of the task pane, choose the **Run** link to set the color of the selected range to yellow.





6. When you want to stop the local web server and uninstall the add-in, follow the applicable instructions:

- To stop the server, run the following command. If you used `npm start`, the following command also uninstalls the add-in.

```
command line
```

```
npm stop
```

- If you manually sideloaded the add-in, see [Remove a sideloaded add-in](#).

## Next steps

Congratulations, you've successfully created an Excel task pane add-in using React! Next, learn more about the capabilities of an Excel add-in and build a more complex add-in by following along with the Excel add-in tutorial.

[Excel add-in tutorial](#)

## Troubleshooting

- Ensure your environment is ready for Office development by following the instructions in [Set up your development environment](#).
- Some of the sample code uses ES6 JavaScript. This isn't compatible with [older versions of Office that use the Trident \(Internet Explorer 11\) browser engine](#). For

information on how to support those platforms in your add-in, see [Support older Microsoft webviews and Office versions](#). If you don't already have a Microsoft 365 subscription to use for development, you might qualify for a Microsoft 365 E5 developer subscription 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](#) [↗](#).

- The automatic `npm install` step Yo Office performs may fail. If you see errors when trying to run `npm start`, navigate to the newly created project folder in a command prompt and manually run `npm install`. For more information about Yo Office, see [Create Office Add-in projects using the Yeoman Generator](#).

## See also

- [Excel add-in tutorial](#)
- [Excel JavaScript object model in Office Add-ins](#)
- [Excel add-in code samples](#)
- [Excel JavaScript API reference](#)
- [Using Visual Studio Code to publish](#)

# Build an Excel content add-in

Article • 08/27/2024

In this article, you'll walk through the process of building an Excel [content add-in](#) using Visual Studio.

## Prerequisites

- [Visual Studio 2019 or later](#) with the **Office/SharePoint development** workload installed.

### ⓘ Note

If you've previously installed Visual Studio, use the Visual Studio Installer to ensure that the **Office/SharePoint development** workload is installed.

- Office connected to a Microsoft 365 subscription (including Office on the web).

## Create the add-in project

1. In Visual Studio, choose **Create a new project**.
2. Using the search box, enter **add-in**. Choose **Excel Web Add-in**, then select **Next**.
3. Name your project **ExcelWebAddIn1** and select **Create**.
4. In the **Create Office Add-in** dialog window, choose the **Insert content into Excel spreadsheets** add-in type, then choose **Next**.
5. Choose the **Basic Add-in** or **Document Visualization Add-in** add-in template, and then choose **Finish** to create the project.
6. Visual Studio creates a solution and its two projects appear in **Solution Explorer**. The **Home.html** file opens in Visual Studio.

## Explore the Visual Studio solution

When you've completed the wizard, Visual Studio creates a solution that contains two projects.

Project	Description
Add-in project	Contains only an XML-formatted add-in only manifest file, which contains all the settings that describe your add-in. These settings help the Office application determine when your add-in should be activated and where the add-in should appear. Visual Studio generates the contents of this file for you so that you can run the project and use your add-in immediately. Change these settings any time by modifying the XML file.
Web application project	Contains the content pages of your add-in, including all the files and file references that you need to develop Office-aware HTML and JavaScript pages. While you develop your add-in, Visual Studio hosts the web application on your local IIS server. When you're ready to publish the add-in, you'll need to deploy this web application project to a web server.

## Update the manifest

1. In **Solution Explorer**, go to the **ExcelWebAddIn1** add-in project and open the **ExcelWebAddIn1Manifest** directory. This directory contains your manifest file, **ExcelWebAddIn1.xml**. The manifest file defines the add-in's settings and capabilities. See the preceding section [Explore the Visual Studio solution](#) for more information about the two projects created by your Visual Studio solution.
2. The `ProviderName` element has a placeholder value. Replace it with your name.
3. The `DefaultValue` attribute of the `DisplayName` element has a placeholder. Replace it with **My Office Add-in**.
4. The `DefaultValue` attribute of the `Description` element has a placeholder. Replace it with **A content add-in for Excel..**
5. Save the file. The updated lines should look like the following code sample.

XML

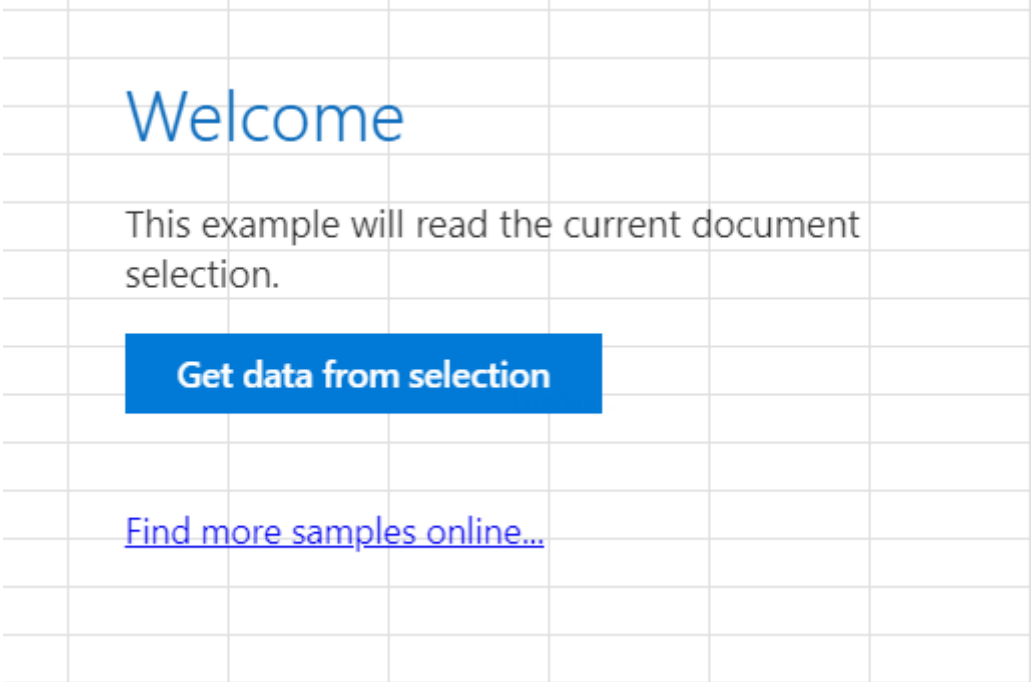
```
...
<ProviderName>John Doe</ProviderName>
<DefaultLocale>en-US</DefaultLocale>
<!-- The display name of your add-in. Used on the store and various
places of the Office UI such as the add-ins dialog. -->
<DisplayName DefaultValue="My Office Add-in" />
<Description DefaultValue="A content add-in for Excel."/>
...
```

# Try it out

1. Using Visual Studio, test the newly created Excel add-in by pressing **F5** or choosing the **Start** button to launch Excel with the content add-in displayed in the spreadsheet.
2. Ensure that there's text in the worksheet, then select any range of cells containing text in the worksheet.
3. Select the tab for the template you chose, then follow the instructions.

Basic Add-in

- In the content add-in, choose the **Get data from selection** button to get the text from the selected range.



The screenshot shows an Excel spreadsheet with a grid. In the center, the word "Welcome" is displayed in a large blue font. Below it, the text "This example will read the current document selection." is shown in a smaller black font. Underneath that text is a blue rectangular button with the white text "Get data from selection". At the bottom of the visible content, there is a blue underlined link that says "Find more samples online...".


## ⓘ Note

To see the `console.log` output, you'll need a separate set of developer tools for a JavaScript console. To learn more about F12 tools and the Microsoft Edge DevTools, visit [Debug add-ins using developer tools for Internet Explorer](#), [Debug add-ins using developer tools for Edge Legacy](#), or [Debug add-ins using developer tools in Microsoft Edge \(Chromium-based\)](#).

# Next steps

Congratulations, you've successfully created an Excel content add-in! Next, learn more about [developing Office Add-ins with Visual Studio](#).

## Troubleshooting

- Ensure your environment is ready for Office development by following the instructions in [Set up your development environment](#).
- Some of the sample code uses ES6 JavaScript. This isn't compatible with [older versions of Office that use the Trident \(Internet Explorer 11\) browser engine](#). For information on how to support those platforms in your add-in, see [Support older Microsoft webviews and Office versions](#). If you don't already have a Microsoft 365 subscription to use for development, you might qualify for a Microsoft 365 E5 developer subscription 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](#) [↗](#).
- If your add-in shows an error (for example, "This add-in could not be started. Close this dialog to ignore the problem or click "Restart" to try again.") when you press  or choose **Debug > Start Debugging** in Visual Studio, see [Debug Office Add-ins in Visual Studio](#) for other debugging options.

## Code samples

- [Excel Content Add-in Humongous Insurance](#) [↗](#)

## See also

- [Office Add-ins platform overview](#)
- [Develop Office Add-ins](#)
- [Excel JavaScript object model in Office Add-ins](#)
- [Excel add-in code samples](#)
- [Excel JavaScript API reference](#)
- [Using Visual Studio Code to publish](#)

# Tutorial: Create an Excel task pane add-in

Article • 02/12/2025

In this tutorial, you'll create an Excel task pane add-in that:

- ✓ Creates a table
- ✓ Filters and sorts a table
- ✓ Creates a chart
- ✓ Freezes a table header
- ✓ Protects a worksheet
- ✓ Opens a dialog

## 💡 Tip

If you've already completed the [Build an Excel task pane add-in](#) quick start using the Yeoman generator, and want to use that project as a starting point for this tutorial, go directly to the [Create a table](#) section to start this tutorial.

If you want a completed version of this tutorial, visit the [Office Add-ins samples repo on GitHub](#) [↗](#).

## Prerequisites

- Node.js (the latest LTS version). Visit the [Node.js site](#) [↗](#) to download and install the right version for your operating system.
- The latest version of Yeoman and the Yeoman generator for Office Add-ins. To install these tools globally, run the following command via the command prompt.

command line

```
npm install -g yo generator-office
```

## ⚠ Note

Even if you've previously installed the Yeoman generator, we recommend you update your package to the latest version from npm.

- Office connected to a Microsoft 365 subscription (including Office on the web).

#### ⓘ Note

If you don't already have Office, you might qualify for a Microsoft 365 E5 developer subscription through the [Microsoft 365 Developer Program](#) <sup>↗</sup>; for details, see the [FAQ](#). Alternatively, you can [sign up for a 1-month free trial](#) <sup>↗</sup> or [purchase a Microsoft 365 plan](#) <sup>↗</sup>.

## Create your add-in project

Run the following command to create an add-in project using the Yeoman generator. A folder that contains the project will be added to the current directory.

command line

```
yo office
```

#### ⓘ Note

When you run the `yo office` command, you may receive prompts about the data collection policies of Yeoman and the Office Add-in CLI tools. Use the information that's provided to respond to the prompts as you see fit.

When prompted, provide the following information to create your add-in project.

- Choose a project type: `Office Add-in Task Pane project`
- Choose a script type: `JavaScript`
- What do you want to name your add-in? `My Office Add-in`
- Which Office client application would you like to support? `Excel`





2. Open the file `./src/taskpane/taskpane.html`. This file contains the HTML markup for the task pane.
3. Locate the `<main>` element and delete all lines that appear after the opening `<main>` tag and before the closing `</main>` tag.
4. Add the following markup immediately after the opening `<main>` tag.

HTML

```
<button class="ms-Button" id="create-table">Create Table</button><br/>
<br/>
```

5. Open the file `./src/taskpane/taskpane.js`. This file contains the Office JavaScript API code that facilitates interaction between the task pane and the Office client application.
6. Remove all references to the `run` button and the `run()` function by doing the following:
  - Locate and delete the line `document.getElementById("run").onclick = run;`.
  - Locate and delete the entire `run()` function.
7. Within the `office.onReady` function call, locate the line `if (info.host === Office.HostType.Excel) {` and add the following code immediately after that line.  
Note:

- This code adds an event handler for the `create-table` button.
- The `createTable` function is wrapped in a call to `tryCatch` (both functions will be added next step). This allows any errors generated by the Office JavaScript layer to be handled separate from your service code.

JavaScript

```
// Assign event handlers and other initialization logic.
document.getElementById("create-table").onclick = () =>
tryCatch(createTable);
```

8. Add the following functions to the end of the file. Note:
  - Your Excel.js business logic will be added to the function that is passed to `Excel.run`. This logic does not execute immediately. Instead, it is added to a queue of pending commands.

- The `context.sync` method sends all queued commands to Excel for execution.
- The `tryCatch` function will be used by all the functions interacting with the workbook from the task pane. Catching Office JavaScript errors in this fashion is a convenient way to generically handle any uncaught errors.

#### ⚠ Note

The following code uses ES6 JavaScript, which isn't compatible with [older versions of Office that use the Trident \(Internet Explorer 11\) browser engine](#). For information on how to support those platforms in production, see [Support older Microsoft webviews and Office versions](#). You might qualify for a Microsoft 365 E5 developer subscription, which has the latest Office applications, to use for development through the [Microsoft 365 Developer Program](#) <sup>↗</sup>; for details, see the [FAQ](#). Alternatively, you can [sign up for a 1-month free trial](#) <sup>↗</sup> or [purchase a Microsoft 365 plan](#) <sup>↗</sup>.

#### JavaScript

```
async function createTable() {
    await Excel.run(async (context) => {

        // TODO01: Queue table creation logic here.

        // TODO02: Queue commands to populate the table with data.

        // TODO03: Queue commands to format the table.

        await context.sync();
    });
}

/** Default helper for invoking an action and handling errors. */
async function tryCatch(callback) {
    try {
        await callback();
    } catch (error) {
        // Note: In a production add-in, you'd want to notify the user
        // through your add-in's UI.
        console.error(error);
    }
}
```

9. Within the `createTable()` function, replace `TODO01` with the following code. Note:

- The code creates a table by using the `add` method of a worksheet's table collection, which always exists even if it is empty. This is the standard way that Excel.js objects are created. There are no class constructor APIs, and you never use a `new` operator to create an Excel object. Instead, you add to a parent collection object.
- The first parameter of the `add` method is the range of only the top row of the table, not the entire range the table will ultimately use. This is because when the add-in populates the data rows (in the next step), it will add new rows to the table instead of writing values to the cells of existing rows. This is a common pattern, because the number of rows a table will have is often unknown when the table is created.
- Table names must be unique across the entire workbook, not just the worksheet.

JavaScript

```
const currentWorksheet =
context.workbook.worksheets.getActiveWorksheet();
const expensesTable = currentWorksheet.tables.add("A1:D1", true
/*hasHeaders*/);
expensesTable.name = "ExpensesTable";
```

10. Within the `createTable()` function, replace `TOD02` with the following code. Note:

- The cell values of a range are set with an array of arrays.
- New rows are created in a table by calling the `add` method of the table's row collection. You can add multiple rows in a single call of `add` by including multiple cell value arrays in the parent array that is passed as the second parameter.

JavaScript

```
expensesTable.getHeaderRowRange().values =
[["Date", "Merchant", "Category", "Amount"]];

expensesTable.rows.add(null /*add at the end*/, [
["1/1/2017", "The Phone Company", "Communications", "120"],
["1/2/2017", "Northwind Electric Cars", "Transportation",
"142.33"],
["1/5/2017", "Best For You Organics Company", "Groceries", "27.9"],
["1/10/2017", "Coho Vineyard", "Restaurant", "33"],
["1/11/2017", "Bellows College", "Education", "350.1"],
["1/15/2017", "Trey Research", "Other", "135"],
["1/15/2017", "Best For You Organics Company", "Groceries",
```

```
"97.88"]
]);
```

11. Within the `createTable()` function, replace `TOD03` with the following code. Note:

- The code gets a reference to the **Amount** column by passing its zero-based index to the `getItemAt` method of the table's column collection.

#### ⓘ Note

Excel.js collection objects, such as `TableCollection`, `WorksheetCollection`, and `TableColumnCollection` have an `items` property that is an array of the child object types, such as `Table` or `Worksheet` or `TableColumn`; but a `*Collection` object is not itself an array.

- The code then formats the range of the **Amount** column as Euros to the second decimal. Learn more about the Excel number format syntax in the article [Number format codes](#) ↗ /
- Finally, it ensures that the width of the columns and height of the rows is big enough to fit the longest (or tallest) data item. Notice that the code must get `Range` objects to format. `TableColumn` and `TableRow` objects do not have format properties.

JavaScript

```
expensesTable.columns.getItemAt(3).getRange().numberFormat =  
[['\u20AC#,##0.00']];  
expensesTable.getRange().format.autofitColumns();  
expensesTable.getRange().format.autofitRows();
```

12. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. Complete the following steps to start the local web server and sideload your add-in.

#### ⓘ Note

- Office Add-ins should use HTTPS, not HTTP, even while you're developing. If you're prompted to install a certificate after you run one of the following commands, accept the prompt to install the certificate that the Yeoman generator provides. You may also have to run your command prompt or terminal as an administrator for the changes to be made.
- If this is your first time developing an Office Add-in on your machine, you may be prompted in the command line to grant Microsoft Edge WebView a loopback exemption ("Allow localhost loopback for Microsoft Edge WebView?"). When prompted, enter **Y** to allow the exemption. Note that you'll need administrator privileges to allow the exemption. Once allowed, you shouldn't be prompted for an exemption when you sideload Office Add-ins in the future (unless you remove the exemption from your machine). To learn more, see ["We can't open this add-in from localhost" when loading an Office Add-in or using Fiddler.](#)

```
> office-addin-taskpane-js@0.0.1 start
> office-addin-debugging start manifest.xml

Debugging is being started...
App type: desktop
? Allow localhost loopback for Microsoft Edge WebView? (Y/n) Y
```

### Tip

If you're testing your add-in on Mac, run the following command in the root directory of your project before proceeding. When you run this command, the local web server starts.

command line

```
npm run dev-server
```

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

#### ⓘ Note

If you are developing on a Mac, enclose the {url} in single quotation marks. Do *not* do this on Windows.

command line

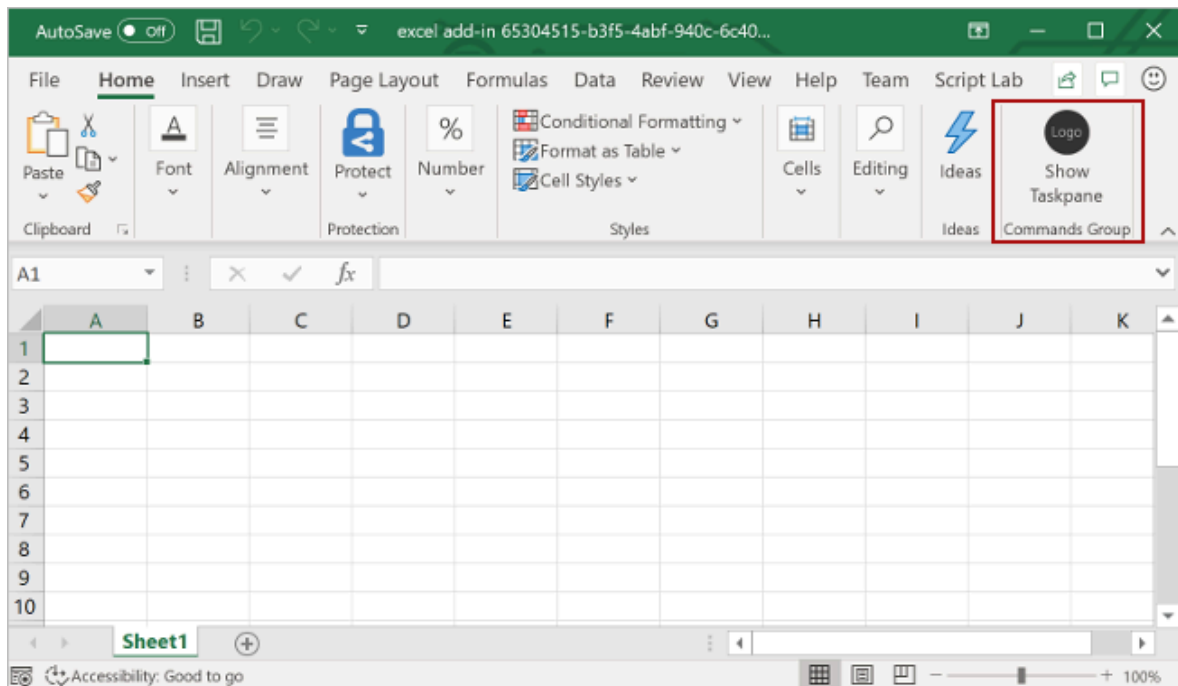
```
npm run start -- web --document {url}
```

The following are examples.

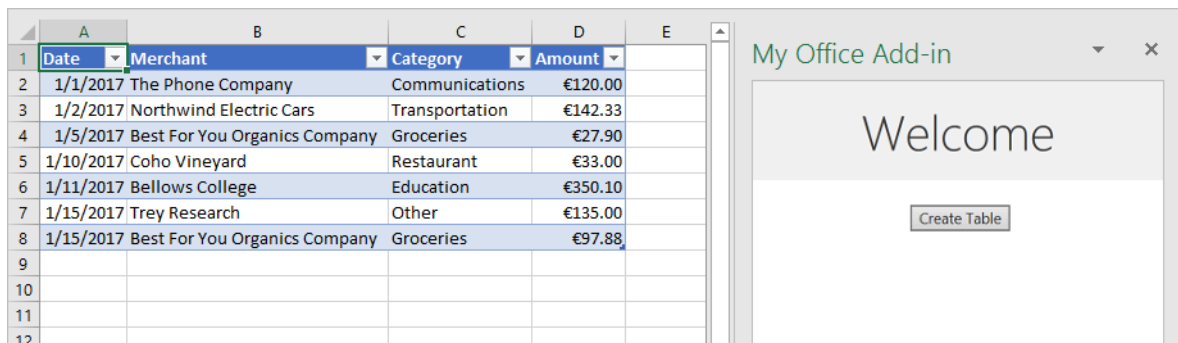
- `npm run start -- web --document https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkCH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii2lDr_oQ?e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

2. In Excel, choose the **Home** tab, and then choose the **Show Taskpane** button on the ribbon to open the add-in task pane.



3. In the task pane, choose the **Create Table** button.



4. When you want to stop the local web server and uninstall the add-in, follow the applicable instructions:

- To stop the server, run the following command. If you used `npm start`, the following command also uninstalls the add-in.

command line

```
npm stop
```

- If you manually sideloaded the add-in, see [Remove a sideloaded add-in](#).

## Filter and sort a table

In this step of the tutorial, you'll filter and sort the table that you created previously.

### Filter the table



1. Open the file `./src/taskpane/taskpane.html`.
2. Locate the `<button>` element for the `create-table` button, and add the following markup after that line.

HTML

```
<button class="ms-Button" id="filter-table">Filter Table</button><br/>
<br/>
```

3. Open the file `./src/taskpane/taskpane.js`.
4. Within the `Office.onReady` function call, locate the line that assigns a click handler to the `create-table` button, and add the following code after that line.

JavaScript

```
document.getElementById("filter-table").onclick = () =>
  tryCatch(filterTable);
```

5. Add the following function to the end of the file.

JavaScript

```
async function filterTable() {
  await Excel.run(async (context) => {

    // TODO01: Queue commands to filter out all expense categories
    except
    //      Groceries and Education.

    await context.sync();
  });
}
```

6. Within the `filterTable()` function, replace `TODO01` with the following code. Note:

- The code first gets a reference to the column that needs filtering by passing the column name to the `getItem` method, instead of passing its index to the `getItemAt` method as the `createTable` method does. Since users can move table columns, the column at a given index might change after the table is created. Hence, it is safer to use the column name to get a reference to the column. We used `getItemAt` safely in the preceding tutorial, because we used it in the very same method that creates the table, so there is no chance that a user has moved the column.

- The `applyValuesFilter` method is one of several filtering methods on the `Filter` object.

JavaScript

```
const currentWorksheet =
context.workbook.worksheets.getActiveWorksheet();
const expensesTable = currentWorksheet.tables.getItem('ExpensesTable');
const categoryFilter =
expensesTable.columns.getItem('Category').filter;
categoryFilter.applyValuesFilter(['Education', 'Groceries']);
```

## Sort the table

1. Open the file `./src/taskpane/taskpane.html`.
2. Locate the `<button>` element for the `filter-table` button, and add the following markup after that line.

HTML

```
<button class="ms-Button" id="sort-table">Sort Table</button><br/><br/>
```

3. Open the file `./src/taskpane/taskpane.js`.
4. Within the `Office.onReady` function call, locate the line that assigns a click handler to the `filter-table` button, and add the following code after that line.

JavaScript

```
document.getElementById("sort-table").onclick = () =>
tryCatch(sortTable);
```

5. Add the following function to the end of the file.

JavaScript

```
async function sortTable() {
    await Excel.run(async (context) => {

        // TODO01: Queue commands to sort the table by Merchant name.

        await context.sync();
    });
}
```

6. Within the `sortTable()` function, replace `TOD01` with the following code. Note:

- The code creates an array of `SortField` objects, which has just one member since the add-in only sorts on the Merchant column.
- The `key` property of a `SortField` object is the zero-based index of the column used for sorting. The rows of the table are sorted based on the values in the referenced column.
- The `sort` member of a `Table` is a `TableSort` object, not a method. The `SortFields` are passed to the `TableSort` object's `apply` method.

JavaScript

```
const currentWorksheet =
context.workbook.worksheets.getActiveWorksheet();
const expensesTable = currentWorksheet.tables.getItem('ExpensesTable');
const sortFields = [
  {
    key: 1,           // Merchant column
    ascending: false,
  }
];

expensesTable.sort.apply(sortFields);
```

7. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. If the local web server is already running and your add-in is already loaded in Excel, proceed to step 2. Otherwise, start the local web server and sideload your add-in:

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ⓘ Note

If you are developing on a Mac, enclose the `{url}` in single quotation marks. Do *not* do this on Windows.

command line

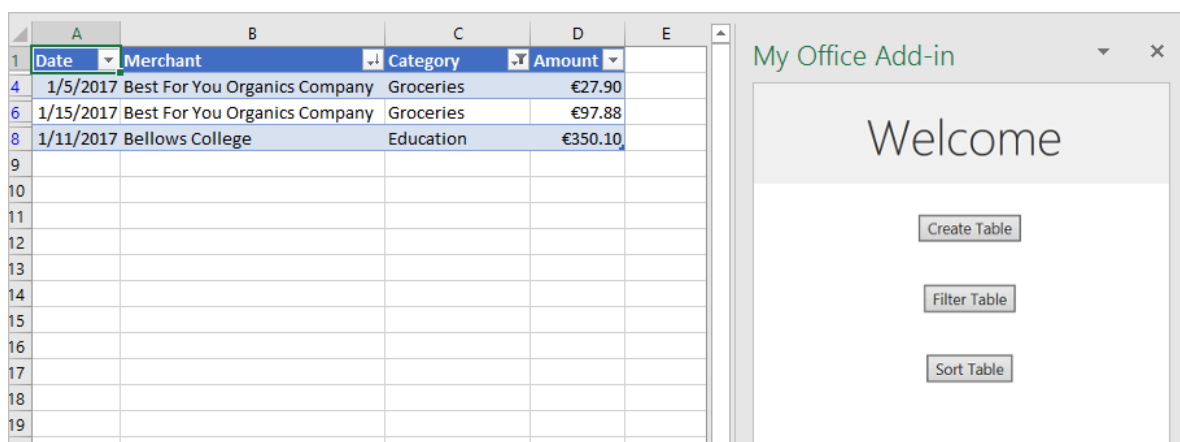
```
npm run start -- web --document {url}
```

The following are examples.

- `npm run start -- web --document https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkCH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii2lDr_oQ?e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

2. If the add-in task pane isn't already open in Excel, go to the **Home** tab and choose the **Show Taskpane** button on the ribbon to open it.
3. If the table you added previously in this tutorial is not present in the open worksheet, choose the **Create Table** button in the task pane.
4. Choose the **Filter Table** button and the **Sort Table** button, in either order.



## Create a chart

In this step of the tutorial, you'll create a chart using data from the table that you created previously, and then format the chart.

## Chart a chart using table data

1. Open the file `./src/taskpane/taskpane.html`.
2. Locate the `<button>` element for the `sort-table` button, and add the following markup after that line.

HTML

```
<button class="ms-Button" id="create-chart">Create Chart</button><br/>
<br/>
```

3. Open the file `./src/taskpane/taskpane.js`.
4. Within the `Office.onReady` function call, locate the line that assigns a click handler to the `sort-table` button, and add the following code after that line.

JavaScript

```
document.getElementById("create-chart").onclick = () =>
  tryCatch(createChart);
```

5. Add the following function to the end of the file.

JavaScript

```
async function createChart() {
  await Excel.run(async (context) => {

    // TODO01: Queue commands to get the range of data to be
    charted.

    // TODO02: Queue command to create the chart and define its
    type.

    // TODO03: Queue commands to position and format the chart.

    await context.sync();
  });
}
```

6. Within the `createChart()` function, replace `TODO01` with the following code. Note that in order to exclude the header row, the code uses the `Table.getDataBodyRange`

method to get the range of data you want to chart instead of the `getRange` method.

JavaScript

```
const currentWorksheet =  
context.workbook.worksheets.getActiveWorksheet();  
const expensesTable = currentWorksheet.tables.getItem('ExpensesTable');  
const dataRange = expensesTable.getDataBodyRange();
```

7. Within the `createChart()` function, replace `TOD02` with the following code. Note the following parameters.

- The first parameter to the `add` method specifies the type of chart. There are several dozen types.
- The second parameter specifies the range of data to include in the chart.
- The third parameter determines whether a series of data points from the table should be charted row-wise or column-wise. The option `auto` tells Excel to decide the best method.

JavaScript

```
const chart = currentWorksheet.charts.add('ColumnClustered', dataRange,  
'Auto');
```

8. Within the `createChart()` function, replace `TOD03` with the following code. Most of this code is self-explanatory. Note:

- The parameters to the `setPosition` method specify the upper left and lower right cells of the worksheet area that should contain the chart. Excel can adjust things like line width to make the chart look good in the space it has been given.
- A "series" is a set of data points from a column of the table. Since there is only one non-string column in the table, Excel infers that the column is the only column of data points to chart. It interprets the other columns as chart labels. So there will be just one series in the chart and it will have index 0. This is the one to label with "Value in €".

JavaScript

```
chart.setPosition("A15", "F30");  
chart.title.text = "Expenses";
```

```
chart.legend.position = "Right";
chart.legend.format.fill.setSolidColor("white");
chart.dataLabels.format.fontSize = 15;
chart.dataLabels.format.font.color = "black";
chart.series.getItemAt(0).name = 'Value in \u20AC';
```

9. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. If the local web server is already running and your add-in is already loaded in Excel, proceed to step 2. Otherwise, start the local web server and sideload your add-in:

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ⓘ Note

If you are developing on a Mac, enclose the {url} in single quotation marks. Do *not* do this on Windows.

command line

```
npm run start -- web --document {url}
```

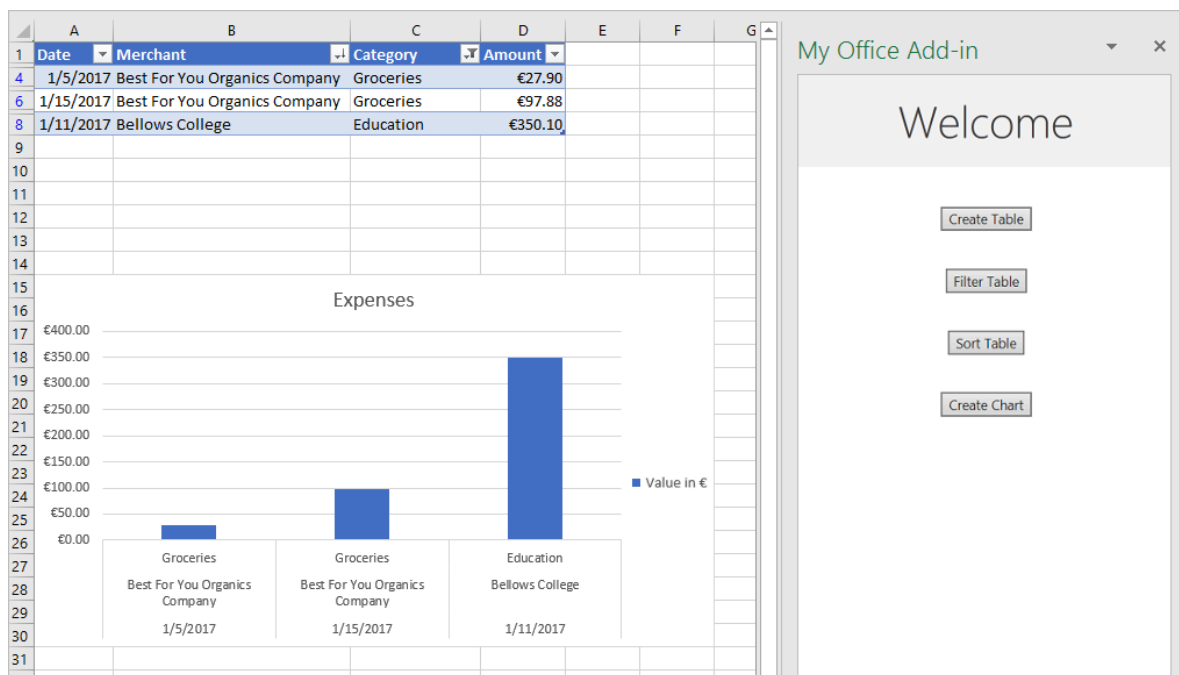
The following are examples.

- `npm run start -- web --document https://contoso.sharepoint.com/:t/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkCH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`

- o `npm run start -- web --document https://contoso-my.sharepoint-  
df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii21Dr_oQ?  
e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

2. If the add-in task pane isn't already open in Excel, go to the **Home** tab and choose the **Show Taskpane** button on the ribbon to open it.
3. If the table you added previously in this tutorial is not present in the open worksheet, choose the **Create Table** button, and then the **Filter Table** button and the **Sort Table** button, in either order.
4. Choose the **Create Chart** button. A chart is created and only the data from the rows that have been filtered are included. The labels on the data points across the bottom are in the sort order of the chart; that is, merchant names in reverse alphabetical order.



## Freeze a table header

When a table is long enough that a user must scroll to see some rows, the header row can scroll out of sight. In this step of the tutorial, you'll freeze the header row of the table that you created previously, so that it remains visible even as the user scrolls down the worksheet.

## Freeze the table's header row



1. Open the file `./src/taskpane/taskpane.html`.
2. Locate the `<button>` element for the `create-chart` button, and add the following markup after that line.

HTML

```
<button class="ms-Button" id="freeze-header">Freeze Header</button>
<br/><br/>
```

3. Open the file `./src/taskpane/taskpane.js`.
4. Within the `Office.onReady` function call, locate the line that assigns a click handler to the `create-chart` button, and add the following code after that line.

JavaScript

```
document.getElementById("freeze-header").onclick = () =>
  tryCatch(freezeHeader);
```

5. Add the following function to the end of the file.

JavaScript

```
async function freezeHeader() {
  await Excel.run(async (context) => {

    // TODO01: Queue commands to keep the header visible when the
    user scrolls.

    await context.sync();
  });
}
```

6. Within the `freezeHeader()` function, replace `TODO01` with the following code. Note:
- The `Worksheet.freezePanes` collection is a set of panes in the worksheet that are pinned, or frozen, in place when the worksheet is scrolled.
  - The `freezeRows` method takes as a parameter the number of rows, from the top, that are to be pinned in place. We pass `1` to pin the first row in place.

JavaScript

```
const currentWorksheet =
  context.workbook.worksheets.getActiveWorksheet();
```

```
currentWorksheet.freezePanes.freezeRows(1);
```

7. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. If the local web server is already running and your add-in is already loaded in Excel, proceed to step 2. Otherwise, start the local web server and sideload your add-in:

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ⓘ Note

If you are developing on a Mac, enclose the {url} in single quotation marks. Do *not* do this on Windows.

command line

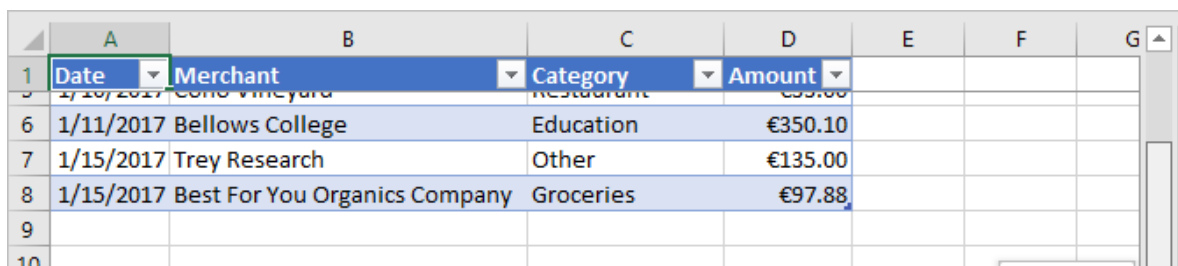
```
npm run start -- web --document {url}
```

The following are examples.

- `npm run start -- web --document https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkCH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-.df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii2lDr_oQ?e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

2. If the add-in task pane isn't already open in Excel, go to the **Home** tab and choose the **Show Taskpane** button on the ribbon to open it.
3. If the table you added previously in this tutorial is present in the worksheet, delete it.
4. In the task pane, choose the **Create Table** button.
5. In the task pane, choose the **Freeze Header** button.
6. Scroll down the worksheet far enough to see that the table header remains visible at the top even when the higher rows scroll out of sight.



1	Date	Merchant	Category	Amount			
2	1/10/2017	Cono Vineyard	Restaurant	€33.00			
6	1/11/2017	Bellows College	Education	€350.10			
7	1/15/2017	Trey Research	Other	€135.00			
8	1/15/2017	Best For You Organics Company	Groceries	€97.88			
9							
10							

## Protect a worksheet

In this step of the tutorial, you'll add a button to the ribbon that toggles worksheet protection on and off.

## Configure the manifest to add a second ribbon button

The steps vary depending on the type of manifest.

Unified manifest for Microsoft 365 (preview)

### ⓘ Note

Using the unified manifest for Microsoft 365 with Excel add-ins is in public developer preview. The unified manifest for Microsoft 365 shouldn't be used in production Excel add-ins. We invite you to try it out in test or development environments. For more information, see the [Public developer preview app manifest schema](#).

## Configure the runtime for the ribbon button

1. Open the manifest file `./manifest.json`.
2. Find the `"extensions.runtimes"` array and add the following commands runtime object.

JSON

```
"runtimes": [  
  {  
    "id": "CommandsRuntime",  
    "type": "general",  
    "code": {  
      "page": "https://localhost:3000/commands.html"  
    },  
    "lifetime": "short",  
    "actions": [  
      {  
        "id": <!--TODO1: Set the action ID -->,  
        "type": "executeFunction",  
      }  
    ]  
  }  
]
```

3. Find `TODO1` and replace it with `"toggleProtection"`. This matches the `id` for the JavaScript function you create in a later step.

### Tip

The value of `"actions.id"` must match the first parameter of the call to `Office.actions.associate` in your `commands.js` file.

4. Ensure that the `"requirements.capabilities"` array contains an object that specifies the `"AddinCommands"` requirement set with a `"minVersion"` of `"1.1"`.

JSON

```
"requirements": {  
  "capabilities": [  
    {  
      "name": "AddinCommands",  
      "minVersion": "1.1"  
    }  
  ]  
},
```

## Configure the UI for the ribbon button

1. After the "extensions.runtimes" array, add the following "ribbons" array.

JSON

```
"ribbons": [
  {
    "contexts": [
      "default"
    ],
    "tabs": [
      {
        "builtInTabID": <!--TODO1: Set the tab ID -->,
        "groups": [
          {
            "id": "worksheetProtectionGroup",
            "label": "Contoso Add-in",
            "controls": [
              {
                "id": "toggleProtectionButton",
                "type": "button",
                "label": <!--TODO2: Label the
button -->,
                "icons": [
                  {
                    "size": 16,
                    "url":
"https://localhost:3000/assets/icon-16.png"
                  },
                  {
                    "size": 32,
                    "url":
"https://localhost:3000/assets/icon-32.png"
                  },
                  {
                    "size": 80,
                    "url":
"https://localhost:3000/assets/icon-80.png"
                  }
                ],
                "supertip": {
                  "title": "Toggle worksheet
protection",
                  "description": "Enables or
disables worksheet protection."
                },
                "actionId": <!--TODO3: Set the
action ID -->
              }
            ]
          }
        ]
      }
    ]
  }
]
```

```
    }  
  ]  
}
```

2. Find `TOD01` and replace it with **"TabHome"**. This ensures that the new button displays in the Home tab in Excel. For other available tab IDs, see [Find the IDs of built-in Office ribbon tabs](#).
3. Find `TOD02` and replace it with **"Toggle worksheet protection"**. This is the label for your button in the Excel ribbon.
4. Find `TOD03` and replace it with **"toggleProtection"**. This value must match the **"`runtimes.actions.id`"** value.
5. Save the file.

## Create the function that protects the sheet

1. Open the file `.\commands\commands.js`.
2. Add the following function immediately after the `action` function. Note that we specify an `args` parameter to the function and the very last line of the function calls `args.completed`. This is a requirement for all add-in commands of type **ExecuteFunction**. It signals the Office client application that the function has finished and the UI can become responsive again.

JavaScript

```
async function toggleProtection(args) {  
  try {  
    await Excel.run(async (context) => {  
  
      // TOD01: Queue commands to reverse the protection status  
      of the current worksheet.  
  
      await context.sync();  
    });  
  } catch (error) {  
    // Note: In a production add-in, you'd want to notify the user  
    through your add-in's UI.  
    console.error(error);  
  }  
  
  args.completed();  
}
```

3. Add the following line immediately after the function to register it.

JavaScript

```
Office.actions.associate("toggleProtection", toggleProtection);
```

4. Within the `toggleProtection` function, replace `TOD01` with the following code. This code uses the worksheet object's `protection` property in a standard toggle pattern. The `TOD02` will be explained in the next section.

JavaScript

```
const sheet = context.workbook.worksheets.getActiveWorksheet();

// TOD02: Queue command to load the sheet's "protection.protected"
// property from
// the document and re-synchronize the document and task pane.

if (sheet.protection.protected) {
    sheet.protection.unprotect();
} else {
    sheet.protection.protect();
}
```

## Add code to fetch document properties into the task pane's script objects

In each function that you've created in this tutorial until now, you queued commands to *write* to the Office document. Each function ended with a call to the `context.sync()` method, which sends the queued commands to the document to be executed. However, the code you added in the last step calls the `sheet.protection.protected` property. This is a significant difference from the earlier functions you wrote, because the `sheet` object is only a proxy object that exists in your task pane's script. The proxy object doesn't know the actual protection state of the document, so its `protection.protected` property can't have a real value. To avoid an exception error, you must first fetch the protection status from the document and use it to set the value of `sheet.protection.protected`. This fetching process has three steps.

1. Queue a command to load (that is, fetch) the properties that your code needs to read.
2. Call the context object's `sync` method to send the queued command to the document for execution and return the requested information.

3. Because the `sync` method is asynchronous, ensure that it has completed before your code calls the properties that were fetched.

These steps must be completed whenever your code needs to *read* information from the Office document.

1. Within the `toggleProtection` function, replace `TOD02` with the following code.

Note:

- Every Excel object has a `load` method. You specify the properties of the object that you want to read in the parameter as a string of comma-delimited names. In this case, the property you need to read is a subproperty of the `protection` property. You reference the subproperty almost exactly as you would anywhere else in your code, with the exception that you use a forward slash (/) character instead of a "." character.
- To ensure that the toggle logic, which reads `sheet.protection.protected`, doesn't run until after the `sync` is complete and the `sheet.protection.protected` has been assigned the correct value that is fetched from the document, it must come after the `await` operator ensures `sync` has completed.

JavaScript

```
sheet.load('protection/protected');  
await context.sync();
```

When you are done, the entire function should look like the following:

JavaScript

```
async function toggleProtection(args) {  
  try {  
    await Excel.run(async (context) => {  
      const sheet =  
context.workbook.worksheets.getActiveWorksheet();  
  
      sheet.load('protection/protected');  
      await context.sync();  
  
      if (sheet.protection.protected) {  
        sheet.protection.unprotect();  
      } else {  
        sheet.protection.protect();  
      }  
  
      await context.sync();  
    });  
  }  
}
```



```

    });
  } catch (error) {
    // Note: In a production add-in, you'd want to notify the user
    through your add-in's UI.
    console.error(error);
  }

  args.completed();
}

```

2. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. Close all Office applications, including Excel (or close the browser tab if you're using Excel on the web).
2. Clear the Office cache. This is necessary to completely clear the old version of the add-in from the client application. Instructions for this process are in the article [Clear the Office cache](#).
3. If the local web server is already running, stop it by entering the following command in the command prompt. This should close the node command window.

command line

```
npm stop
```

4. Because your manifest file has been updated, you must sideload your add-in again, using the updated manifest file. Start the local web server and sideload your add-in.

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ! Note

If you are developing on a Mac, enclose the `{url}` in single quotation marks. Do *not* do this on Windows.

command line

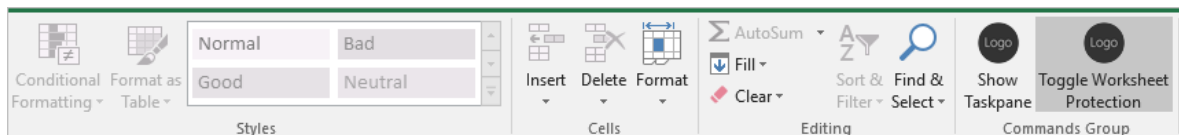
```
npm run start -- web --document {url}
```

The following are examples.

- `npm run start -- web --document https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCMfF1WZQj3VYhYQ?e=F4QM1R`
- `npm run start -- web --document https://1drv.ms/x/s!jkcH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-.df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0Bw1rzetbXvnaRYii2lDr_oQ?e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

5. On the **Home** tab in Excel, choose the **Toggle Worksheet Protection** button. Note that most of the controls on the ribbon are disabled (and visually grayed-out) as seen in the following screenshot.



6. Select a cell and try to edit its content. Excel displays an error message indicating that the worksheet is protected.
7. Choose the **Toggle Worksheet Protection** button again, and the controls are reenabled, and you can change cell values again.

## Open a dialog

In this final step of the tutorial, you'll open a dialog in your add-in, pass a message from the dialog process to the task pane process, and close the dialog. Office Add-in dialogs

are *nonmodal*: a user can continue to interact with both the document in the Office application and with the host page in the task pane.

## Create the dialog page

1. In the `./src` folder that's located at the root of the project, create a new folder named **dialogs**.
2. In the `./src/dialogs` folder, create new file named **popup.html**.
3. Add the following markup to **popup.html**. Note:
  - The page has an `<input>` field where the user will enter their name, and a button that will send this name to the task pane where it will display.
  - The markup loads a script named **popup.js** that you will create in a later step.
  - It also loads the Office.js library because it will be used in **popup.js**.

HTML

```
<!DOCTYPE html>
<html>
  <head lang="en">
    <title>Dialog for My Office Add-in</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">

    <!-- For more information on Fluent UI, visit
https://developer.microsoft.com/fluentui. -->
    <link rel="stylesheet" href="https://res-
1.cdn.office.net/files/fabric-cdn-prod_20230815.002/office-ui-fabric-
core/11.0.0/css/fabric.min.css"/>

    <script type="text/javascript"
src="https://appsforoffice.microsoft.com/lib/1/hosted/office.js">
</script>
    <script type="text/javascript" src="popup.js"></script>

  </head>
  <body style="display:flex;flex-direction:column;align-
items:center;justify-content:center">
    <p class="ms-font-xl">ENTER YOUR NAME</p>
    <input id="name-box" type="text"/><br/><br/>
    <button id="ok-button" class="ms-Button">OK</button>
  </body>
</html>
```

4. In the `./src/dialogs` folder, create new file named `popup.js`.
5. Add the following code to `popup.js`. Note the following about this code.
  - *Every page that calls APIs in the Office.js library must first ensure that the library is fully initialized.* The best way to do that is to call the `Office.onReady()` function. The call of `Office.onReady()` must run before any calls to Office.js; hence the assignment is in a script file that is loaded by the page, as it is in this case.

JavaScript

```
Office.onReady((info) => {  
    // TODO1: Assign handler to the OK button.  
});  
  
// TODO2: Create the OK button handler.
```

6. Replace `TODO1` with the following code. You'll create the `sendStringToParentPage` function in the next step.

JavaScript

```
document.getElementById("ok-button").onclick = () =>  
    tryCatch(sendStringToParentPage);
```

7. Replace `TODO2` with the following code. The `messageParent` method passes its parameter to the parent page, in this case, the page in the task pane. The parameter must be a string, which includes anything that can be serialized as a string, such as XML or JSON, or any type that can be cast to a string. This also adds the same `tryCatch` method used in `taskpane.js` for error handling.

JavaScript

```
function sendStringToParentPage() {  
    const userName = document.getElementById("name-box").value;  
    Office.context.ui.messageParent(userName);  
}  
  
/** Default helper for invoking an action and handling errors. */  
async function tryCatch(callback) {  
    try {  
        await callback();  
    } catch (error) {  
        // Note: In a production add-in, you'd want to notify the user  
        // through your add-in's UI.  
        console.error(error);  
    }  
}
```

```
}  
}
```

### ⓘ Note

The **popup.html** file, and the **popup.js** file that it loads, run in an entirely separate browser runtime process from the add-in's task pane. If **popup.js** was transpiled into the same **bundle.js** file as the **app.js** file, then the add-in would have to load two copies of the **bundle.js** file, which defeats the purpose of bundling. Therefore, this add-in does not transpile the **popup.js** file at all.

## Update webpack config settings

Open the file **webpack.config.js** in the root directory of the project and complete the following steps.

1. Locate the **entry** object within the **config** object and add a new entry for **popup**.

JavaScript

```
popup: "./src/dialogs/popup.js"
```

After you've done this, the new **entry** object will look like this.

JavaScript

```
entry: {  
  polyfill: "@babel/polyfill",  
  taskpane: "./src/taskpane/taskpane.js",  
  commands: "./src/commands/commands.js",  
  popup: "./src/dialogs/popup.js"  
},
```

2. Locate the **plugins** array within the **config** object and add the following object to the end of that array.

JavaScript

```
new HtmlWebpackPlugin({  
  filename: "popup.html",  
  template: "./src/dialogs/popup.html",  
  chunks: ["polyfill", "popup"]  
})
```

After you've done this, the new `plugins` array will look like this.

JavaScript

```
plugins: [  
  new CleanWebpackPlugin(),  
  new HtmlWebpackPlugin({  
    filename: "taskpane.html",  
    template: "./src/taskpane/taskpane.html",  
    chunks: ['polyfill', 'taskpane']  
  }),  
  new CopyWebpackPlugin([  
    {  
      to: "taskpane.css",  
      from: "./src/taskpane/taskpane.css"  
    }  
  ]),  
  new HtmlWebpackPlugin({  
    filename: "commands.html",  
    template: "./src/commands/commands.html",  
    chunks: ["polyfill", "commands"]  
  }),  
  new HtmlWebpackPlugin({  
    filename: "popup.html",  
    template: "./src/dialogs/popup.html",  
    chunks: ["polyfill", "popup"]  
  })  
],
```

3. If the local web server is running, stop it by entering the following command in the command prompt. This should close the node command window.

command line

```
npm stop
```

4. Run the following command to rebuild the project.

command line

```
npm run build
```

## Open the dialog from the task pane

1. Open the file `./src/taskpane/taskpane.html`.

2. Locate the `<button>` element for the `freeze-header` button, and add the following markup after that line.

HTML

```
<button class="ms-Button" id="open-dialog">Open Dialog</button><br/>
<br/>
```

3. The dialog will prompt the user to enter a name and pass the user's name to the task pane. The task pane will display it in a label. Immediately after the `button` that you just added, add the following markup.

HTML

```
<label id="user-name"></label><br/><br/>
```

4. Open the file `./src/taskpane/taskpane.js`.
5. Within the `Office.onReady` function call, locate the line that assigns a click handler to the `freeze-header` button, and add the following code after that line. You'll create the `openDialog` method in a later step.

JavaScript

```
document.getElementById("open-dialog").onclick = openDialog;
```

6. Add the following declaration to the end of the file. This variable is used to hold an object in the parent page's execution context that acts as an intermediary to the dialog page's execution context.

JavaScript

```
let dialog = null;
```

7. Add the following function to the end of the file (after the declaration of `dialog`). The important thing to notice about this code is what is *not* there: there is no call of `Excel.run`. This is because the API to open a dialog is shared among all Office applications, so it is part of the Office JavaScript Common API, not the Excel-specific API.

JavaScript

```
function openDialog() {  
    // TODO01: Call the Office Common API that opens a dialog.  
}
```

8. Replace `TODO01` with the following code. Note:

- The `displayDialogAsync` method opens a dialog in the center of the screen.
- The first parameter is the URL of the page to open.
- The second parameter passes options. `height` and `width` are percentages of the size of the Office application's window.

JavaScript

```
Office.context.ui.displayDialogAsync(  
    'https://localhost:3000/popup.html',  
    {height: 45, width: 55},  
  
    // TODO02: Add callback parameter.  
);
```

## Process the message from the dialog and close the dialog

1. Within the `openDialog` function in the file `./src/taskpane/taskpane.js`, replace `TODO02` with the following code. Note:

- The callback is executed immediately after the dialog successfully opens and before the user has taken any action in the dialog.
- The `result.value` is the object that acts as an intermediary between the execution contexts of the parent and dialog pages.
- The `processMessage` function will be created in a later step. This handler will process any values that are sent from the dialog page with calls of the `messageParent` function.

JavaScript

```
function (result) {  
    dialog = result.value;  
    dialog.addEventHandler(Office.EventType.DialogMessageReceived,  
        processMessage);  
}
```



2. Add the following function after the `openDialog` function.

JavaScript

```
function processMessage(arg) {  
    document.getElementById("user-name").innerHTML = arg.message;  
    dialog.close();  
}
```

3. Verify that you've saved all of the changes you've made to the project.

## Test the add-in

1. If the local web server is already running and your add-in is already loaded in Excel, proceed to step 2. Otherwise, start the local web server and sideload your add-in:

- To test your add-in in Excel, run the following command in the root directory of your project. This starts the local web server (if it's not already running) and opens Excel with your add-in loaded.

command line

```
npm start
```

- To test your add-in in Excel on the web, run the following command in the root directory of your project. When you run this command, the local web server starts. Replace "{url}" with the URL of an Excel document on your OneDrive or a SharePoint library to which you have permissions.

### ⓘ Note

If you are developing on a Mac, enclose the `{url}` in single quotation marks. Do *not* do this on Windows.

command line

```
npm run start -- web --document {url}
```

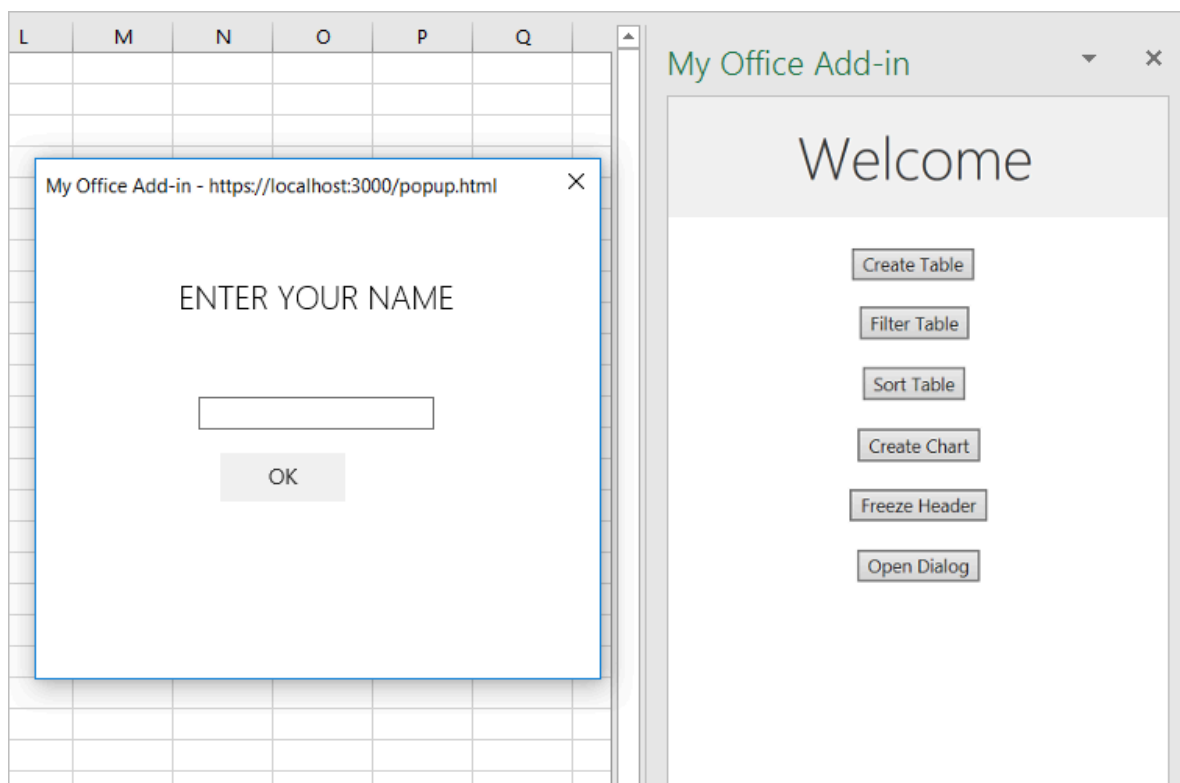
The following are examples.

- `npm run start -- web --document`  
`https://contoso.sharepoint.com/:t:/g/EZGxP7ksiE5DuxvY638G798BpuhwluxCM`  
`fF1WZQj3VYhYQ?e=F4QM1R`

- `npm run start -- web --document https://1drv.ms/x/s!jkCH7spkM4EGgcZUgqthk4IK3N0ypVw?e=Z6G1qp`
- `npm run start -- web --document https://contoso-my.sharepoint-df.com/:t:/p/user/EQda453DNTpFn11bFPhOVR0BwlrzetbXvnaRYii2lDr_oQ?e=RSccmNP`

If your add-in doesn't sideload in the document, manually sideload it by following the instructions in [Manually sideload add-ins to Office on the web](#).

2. If the add-in task pane isn't already open in Excel, go to the **Home** tab and choose the **Show Taskpane** button on the ribbon to open it.
3. Choose the **Open Dialog** button in the task pane.
4. While the dialog is open, drag it and resize it. Note that you can interact with the worksheet and press other buttons on the task pane, but you cannot launch a second dialog from the same task pane page.
5. In the dialog, enter a name and choose the **OK** button. The name appears on the task pane and the dialog closes.
6. Optionally, in the `./src/taskpane/taskpane.js` file, comment out the line `dialog.close();` in the `processMessage` function. Then repeat the steps of this section. The dialog stays open and you can change the name. You can close it manually by pressing the **X** button in the upper right corner.



## Next steps


In this tutorial, you've created an Excel task pane add-in that interacts with tables, charts, worksheets, and dialogs in an Excel workbook. To learn more about building Excel add-ins, continue to the following article.

[Excel add-ins overview](#)

## Code samples

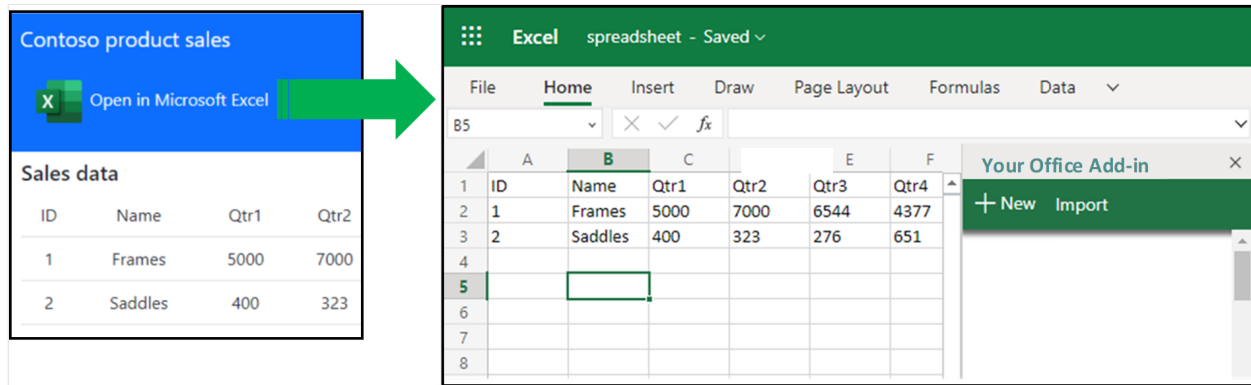
- [Completed Excel add-in tutorial](#) : The result of completing this tutorial.

## See also

- [Office Add-ins platform overview](#)
- [Develop Office Add-ins](#)
- [Excel JavaScript object model in Office Add-ins](#)
- [Office Add-ins code samples](#) 

# Create an Excel spreadsheet from your web page, populate it with data, and embed your Office Add-in

Article • 03/09/2023



Microsoft partners with SaaS web applications know that their customers often want to open their data from a web page in an Excel spreadsheet. They use Excel to do analysis on the data, or other types of number crunching. Then they upload the data back to the web site.

Instead of multiple steps to export the data from the web site to a .csv file, import the .csv file into Excel, work with the data, then export it from Excel, and upload it back to the web site, we can simplify this process to one button click.

This article shows how to add an Excel button to your web site. When a customer chooses the button, it automatically creates a new spreadsheet with the requested data, uploads it to the customer's OneDrive, and opens it in Excel on a new browser tab. With one click the requested data is opened in Excel and formatted correctly. Additionally the pattern embeds your own Office Add-in inside the spreadsheet so that customers can still access your services from the context of Excel.

Microsoft partners who implemented this pattern have seen increased customer satisfaction. They've also seen a significant increase in engagement with their add-ins by embedding them in the Excel spreadsheet. We recommend that if you have a web site for customers to work with data, that you consider implementing this pattern in your own solution.

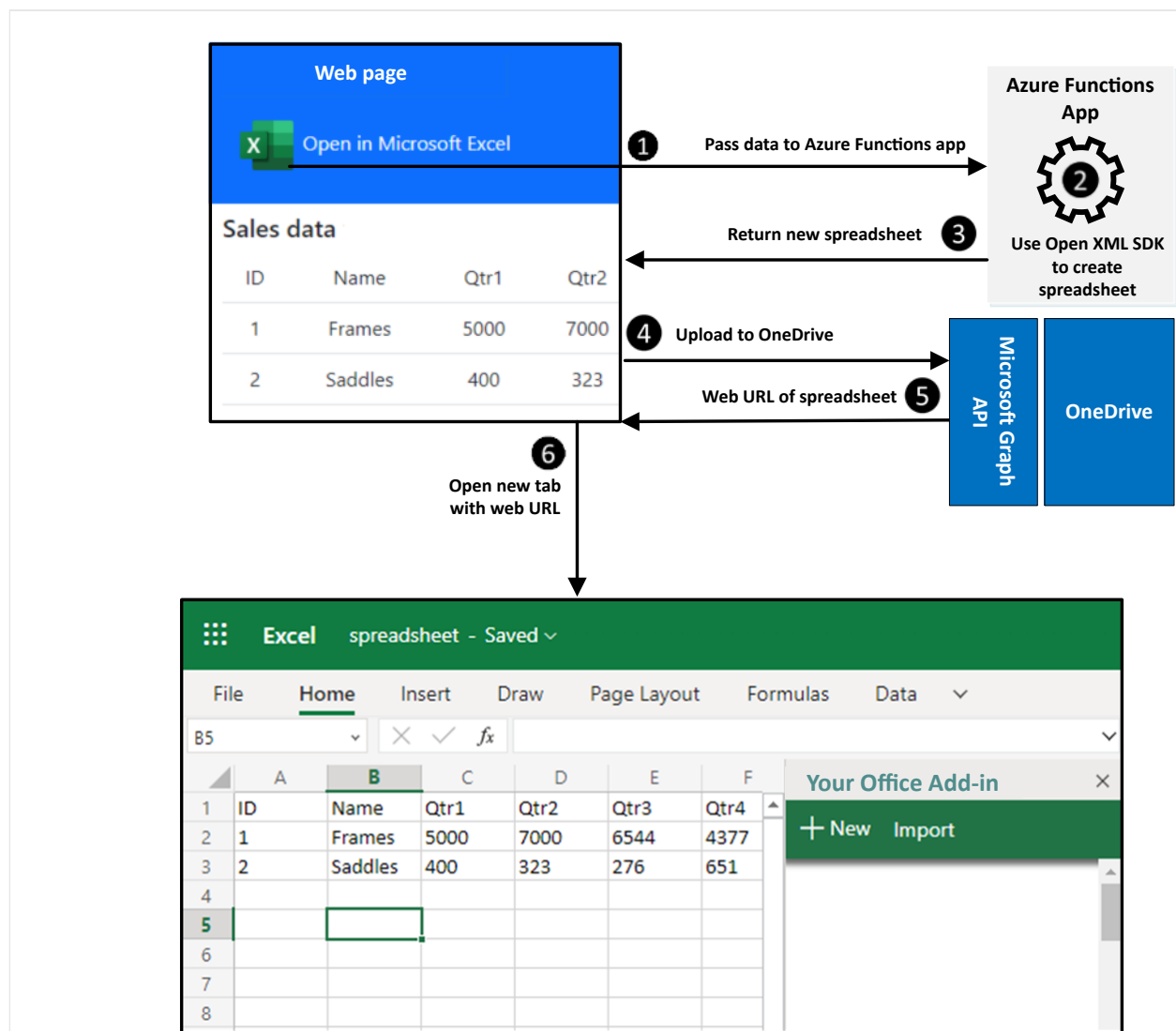
## Prerequisites

- [Visual Studio 2022 or later](#) . Add the Office/SharePoint development workload when configuring Visual Studio.
- [Visual Studio Code](#) .
- Microsoft 365. You can get a free developer sandbox that provides a renewable 90-day Microsoft 365 E5 developer subscription. The developer sandbox includes a Microsoft Azure subscription that you can use for app registrations in later steps in this article. If you prefer, you can use a separate Microsoft Azure subscription for app registrations. Get a trial subscription at Microsoft Azure.
- One or more files and folders on OneDrive in the Microsoft 365 account.

## Run the sample code

The sample code for this article is named [Create a spreadsheet from your web site, populate it with data, and embed your Excel add-in](#) . To run the sample, follow the instructions in the [readme](#) .

## Solution architecture



The solution described in this article adds an **Open in Microsoft Excel** button to the web site and interacts with Azure Functions, and the Microsoft Graph API. The following sequence of events occurs when the user wants to open their data in a new Excel spreadsheet.

1. The user chooses the **Open in Microsoft Excel** button. The web page passes the data to a function in an Azure Functions app.
2. The function uses the Open XML SDK to create a new Excel spreadsheet in memory. It populates the spreadsheet with the data, and embeds your Office Add-in.
3. The function returns the spreadsheet as a Base64 encoded string to the web page.
4. The web page calls the Microsoft Graph API to upload the spreadsheet to the user's OneDrive.
5. Microsoft Graph returns the web url location of the new spreadsheet file.
6. The web page opens a new browser tab to open the spreadsheet at the web url. The spreadsheet contains the data, and your add-in.

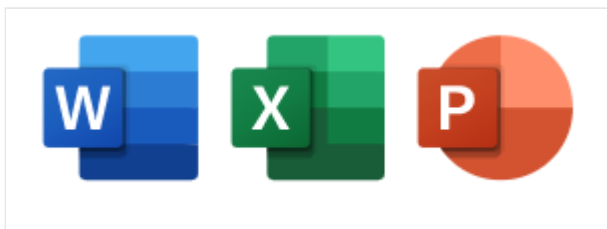
## Key parts of the solution

The solution has two projects that you build:

- An Azure Functions app containing a `FunctionCreateSpreadsheet` function.
- A Node.js web application project.

The following sections describe important concepts and implementation details for constructing the solution. A full reference implementation can be found in the [sample code](#) [↗](#) for additional implementation details.

## Excel button and Fluent UI



You need a button on the web site that creates the Excel spreadsheet. A best practice is to use the Fluent UI to help your users transition between Microsoft products. You should always use an Office icon to indicate which Office application will be launched from your web page. For more information, see [Office Brand Icons](#) on the Fluent UI developer portal.

## Sign in the user

The sample code is built from the Microsoft identity sample named [Vanilla JavaScript single-page application using MSAL.js to authenticate users to call Microsoft Graph](#) <sup>↗</sup>. All authentication code and UI is from this sample. Please refer to this sample for more information about writing code for authentication and authorization. For a full list of identity samples for a wide range of platforms, see [Microsoft identity platform code samples](#).

## Create the spreadsheet with the Open XML SDK

The sample code uses the [Open XML SDK](#) to create the spreadsheet. Because the Open XML SDK uses .NET it is encapsulated in an Azure Functions app named `FunctionCreateSpreadsheet`. You can call this function from your Node.js web application. `FunctionCreateSpreadsheet` uses the `SpreadsheetBuilder` helper class to create a new spreadsheet in memory. The code is based on [Create a spreadsheet document by providing a file name \(Open XML SDK\)](#).

## Populate the spreadsheet with data

The `FunctionCreateSpreadsheet` function accepts a JSON body containing the row and column data. This is passed to the `SpreadsheetBuilder.InsertData` method which iterates through all rows and columns and adds them to the worksheet.

Much of the `SpreadsheetBuilder` class contains code that was generated by the Open XML 2.5 SDK Productivity Tools. These are available at the link for the [Open XML 2.5 SDK](#).

## Embed your Office Add-in inside the spreadsheet

The `SpreadsheetBuilder` class also embeds the Script Lab add-in inside the spreadsheet and configures to display when the document is opened.

The `SpreadsheetBuilder.GenerateWebExtensionPart1Content` method in the `SpreadsheetBuilder.cs` file sets the reference to the ID of Script Lab in Microsoft AppSource:

C#

```
We.WebExtensionStoreReference webExtensionStoreReference1 = new  
We.WebExtensionStoreReference() { Id = "wa104380862", Version = "1.1.0.0",
```

```
Store = "en-US", StoreType = "OMEX" };
```

- The **StoreType** value is "OMEX", an alias for Microsoft AppSource.
- The **Store** value is "en-US" found in the Microsoft AppSource culture section for Script Lab.
- The **Id** value is the Microsoft AppSource asset ID for Script Lab.

You can change these values to embed your own Office Add-in. This makes it discoverable to the user and increases engagement with your add-in and web services. If your add-in is deployed through central deployment, use the following values instead.

C#

```
We.WebExtensionStoreReference webExtensionStoreReference1 = new
We.WebExtensionStoreReference() { Id = "<Your add-in GUID>", Version = "
<Your version>", Store = "excatalog", StoreType = "excatalog" };
We.WebExtensionStoreReference webExtensionStoreReference2 = new
We.WebExtensionStoreReference() { Id = "<Your add-in GUID>", Version = "
<Your version>", Store = "omex", StoreType = "omex" };
webExtensionReferenceList1.Append(webExtensionStoreReference2);
```

For more information about alternative values for these attributes, see [Automatically open a task pane with a document](#) and [\[MS-OWEXML\]: CT\\_OsfWebExtensionReference](#)

## Upload the spreadsheet to OneDrive

When the spreadsheet is fully constructed the `FunctionCreateSpreadsheet` function returns a Base64 encoded string version of the spreadsheet to the web application. Then the web application uses the Microsoft Graph API to upload the spreadsheet to the user's OneDrive. The web application creates the file at `\openinexcel\spreadsheet.xlsx`, but you can modify the code to use any folder and filename you prefer.

## Additional considerations for your solution

Everyone's solution is different in terms of technologies and approaches. The following considerations will help you plan how to modify your solution to open documents and embed your Office Add-in.

### Read custom properties when your add-in starts

When you embed your add-in inside the spreadsheet, you can include custom properties. The `SpreadsheetBuilder.cs` file includes commented code that shows how to



insert a user name if you have a `userName` variable.

C#

```
// CUSTOM MODIFICATION BEGIN
// Uncomment the following code to add your own custom name/value pair
properties for the add-in.
// We.WebExtensionProperty webExtensionProperty2 = new
We.WebExtensionProperty() { Name = "userName", Value = userName };
// webExtensionPropertyBag1.Append(webExtensionProperty2);
// CUSTOM MODIFICATION END
```

Uncomment the code and change it to add any customer properties you need. In your add-in, use the [Office Settings get method](#) to retrieve a custom property. The following sample shows how to get the user name property from the spreadsheet.

JavaScript

```
let userName = Office.context.document.settings.get('userName');
```

### ⊗ Caution

Don't store sensitive information in custom properties such as auth tokens or connection strings. Properties in the spreadsheet are not encrypted or protected.

See [Persist add-in state and settings](#) for complete details on how to read custom properties when your add-in starts.

## Use single sign-on

To simplify authentication, we recommend your add-in implements single sign-on. This ensure the user does not need to sign in a second time to access your add-in. For more information, see [Enable single sign-on for Office Add-ins](#)

## See also

- [Welcome to the Open XML SDK 2.5 for Office](#)
- [Automatically open a task pane with a document](#)
- [Persisting add-in state and settings](#)
- [Create a spreadsheet document by providing a file name](#)

# Excel JavaScript API overview

Article • 05/02/2023

An Excel add-in interacts with objects in Excel by using the Office JavaScript API, which includes two JavaScript object models:

- **Excel JavaScript API:** These are the [application-specific APIs](#) for Excel. Introduced with Office 2016, the [Excel JavaScript API](#) provides strongly-typed objects that you can use to access worksheets, ranges, tables, charts, and more.
- **Common APIs:** Introduced with Office 2013, the [Common API](#) can be used to access features such as UI, dialogs, and client settings that are common across multiple types of Office applications.

This section of the documentation focuses on the Excel JavaScript API, which you'll use to develop the majority of functionality in add-ins that target Excel on the web or Excel 2016 or later. For information about the Common API, see [Common JavaScript API object model](#).

## Learn object model concepts

See [Excel JavaScript object model in Office Add-ins](#) for information about important object model concepts.

For hands-on experience using the Excel JavaScript API to access objects in Excel, complete the [Excel add-in tutorial](#).

## Learn API capabilities

Each major Excel API feature has an article or set of articles exploring what that feature can do and the relevant object model.

- [Charts](#)
- [Comments](#)
- [Conditional formatting](#)
- [Custom functions](#)
- [Data validation](#)
- [Data types](#)
- [Events](#)
- [PivotTables](#)
- [Ranges](#) and [Cells](#)

- [RangeAreas \(Multiple ranges\)](#)
- [Shapes](#)
- [Tables](#)
- [Workbooks and Application-level APIs](#)
- [Worksheets](#)

For detailed information about the Excel JavaScript API object model, see the [Excel JavaScript API reference documentation](#).

## Try out code samples in Script Lab

Use [Script Lab](#) to get started quickly with a collection of built-in samples that show how to complete tasks with the API. You can run the samples in Script Lab to instantly see the result in the task pane or worksheet, examine the samples to learn how the API works, and even use samples to prototype your own add-in.

## See also

- [Excel add-ins documentation](#)
- [Excel add-ins overview](#)
- [Excel JavaScript API reference](#)
- [Office client application and platform availability for Office Add-ins](#)
- [Using the application-specific API model](#)