**Office Add-ins platform overview**

You can use the Office Add-ins platform to build solutions that extend Office applications and interact with content in Office documents. With Office Add-ins, you can use familiar web technologies such as HTML, CSS, and JavaScript to extend and interact with Word, Excel, PowerPoint, OneNote, Project, and Outlook. Your solution can run in Office across multiple platforms, including Office for Windows, Office Online, Office for the Mac, and Office for the iPad.

Office Add-ins can do almost anything a webpage can do inside a browser. Use the Office Add-ins platform to:

* **Add new functionality to Office clients** - Bring external data into Office, automate Office documents, expose third-party functionality in Office clients, and more.  For example, use Microsoft Graph API to connect to data that drives productivity.
* **Create new rich, interactive objects that can be embedded in Office documents** - Embed maps, charts, and interactive visualizations that users can add to their own Excel spreadsheets and PowerPoint presentations.

**How are Office Add-ins different than COM and VSTO add-ins?**

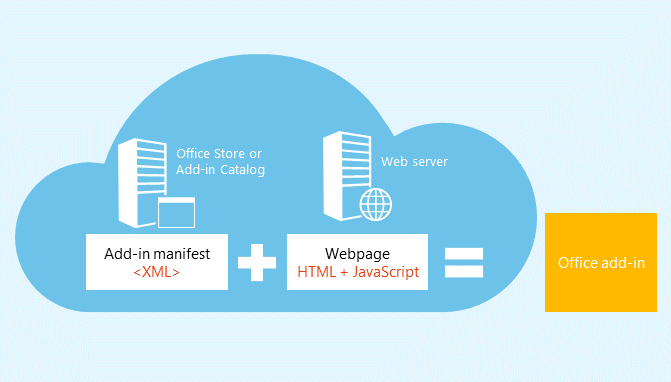
COM or VSTO add-ins are earlier Office integration solutions that run only on Office for Windows. Unlike COM add-ins, Office Add-ins don't involve code that runs on the user's device or in the Office client. For an Office Add-in, the host application, for example Excel, reads the add-in manifest and hooks up the add-in’s custom ribbon buttons and menu commands in the UI. Then, when needed, it loads the add-in's JavaScript and HTML code which executes in the context of a browser in a sandbox.

Office Add-ins provide the following advantages over add-ins built using VBA, COM or VSTO:

* Cross-platform support. Office Add-ins run in Office for Windows, Mac, iOS, and Office Online.
* Single sign on (SSO). Office Add-ins integrate easily with users' Office 365 accounts.
* Centralized deployment and distribution. Admins can deploy Office Add-ins centrally across an organization.
* Easy access via the Office Store. You can make your solution available to a broad audience by submitting it to the Office Store.
* Based on standard web technology. You can use any library you like to build Office Add-ins.

**Components of an Office Add-in**

An Office Add-in includes two basic components: an XML manifest file, and your own web application. The manifest defines various settings, including how your add-in integrates with Office clients. Your web application needs to be hosted on a web server, or web hosting service, such as Microsoft Azure.

​**Manifest + webpage = an Office Add-in** [](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview01.png)

**Manifest**

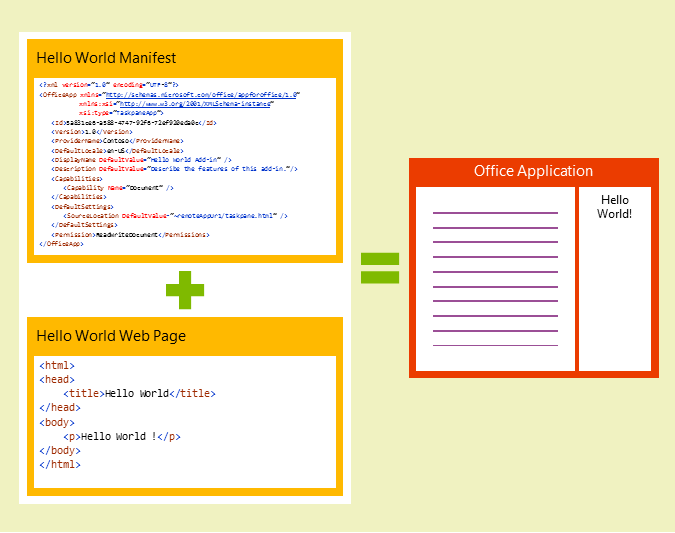
The manifest is an XML file that specifies settings and capabilities of the add-in, such as:

* The add-in's display name, description, ID, version, and default locale.
* How the add-in integrates with Office.
* The permission level and data access requirements for the add-in.

**Web app**

The most basic Office Add-in consists of a static HTML page that is displayed inside an Office application, but doesn't interact with either the Office document or any other Internet resource. However, to create an experience that interacts with Office documents or allows the user to interact with online resources from an Office host application, you can use any technologies, both client and server side, that your hosting provider supports (such as ASP.NET, PHP, or Node.js). To interact with Office clients and documents, you use the Office.js JavaScript APIs.

**Components of a Hello World Office Add-in**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview07.png)

**Extending and interacting with Office clients**

Office Add-ins can do the following within an Office host application:

* Extend functionality (any Office application)
* Create new objects (Excel or PowerPoint)

**Extend Office functionality**

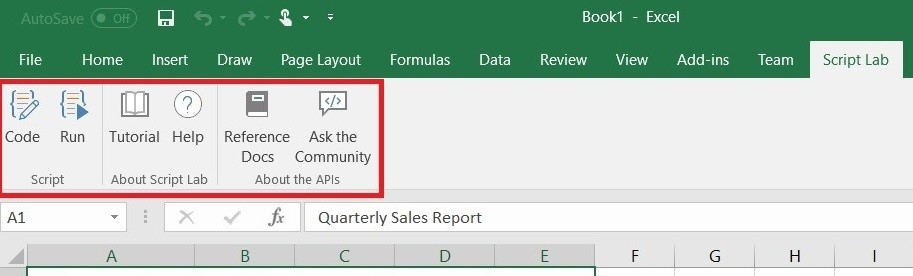
You can add new functionality to Office applications via the following:

* Custom ribbon buttons and menu commands (collectively called “add-in commands”)
* Insertable task panes

Custom UI and task panes are specified in the add-in manifest.

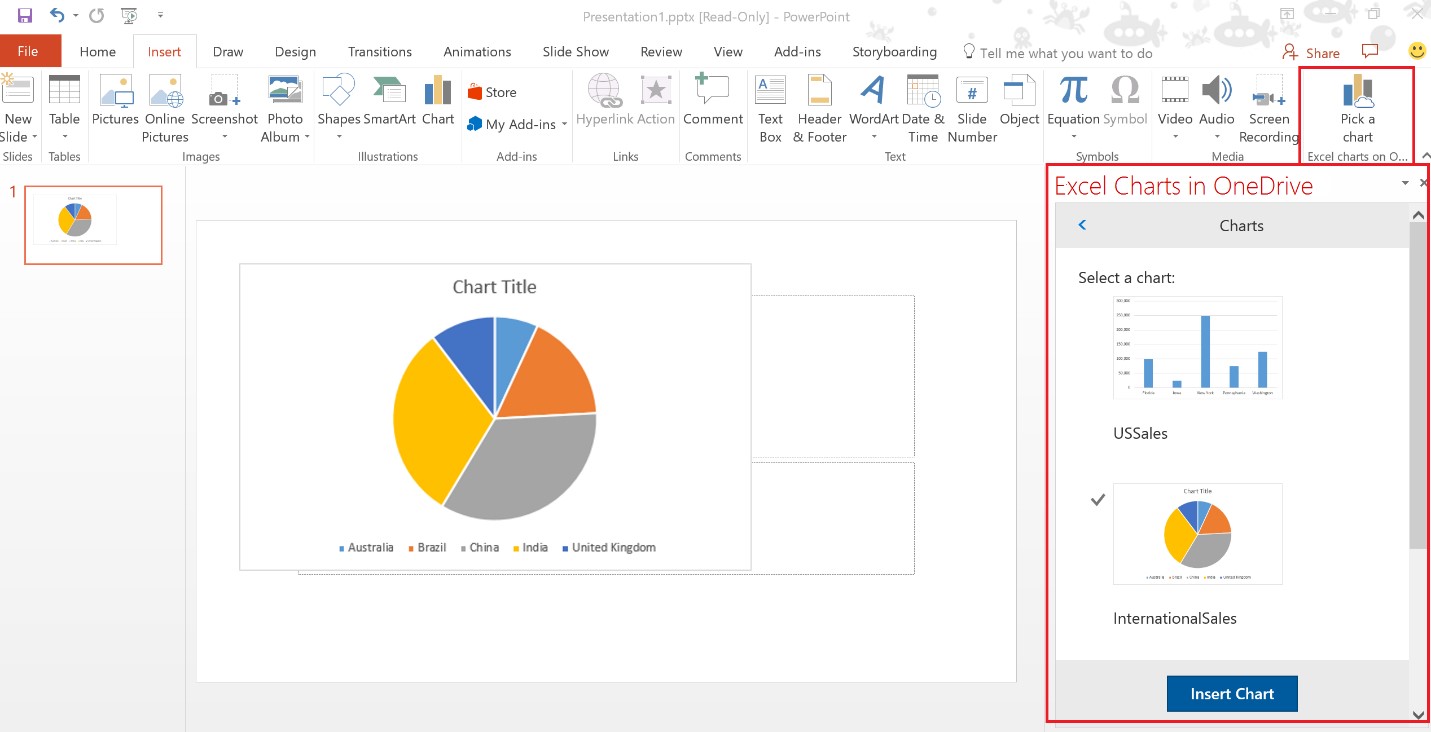
**Custom buttons and menu commands**

You can add custom ribbon buttons and menu items to the ribbon in Office for Windows Desktop and Office Online. This makes it easy for users to access your add-in directly from their Office application. Command buttons can launch different actions such as showing a task pane with custom HTML or executing a JavaScript function.

**Add-in commands running in Excel Desktop** [](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-commands-overview..png)

**Task panes**

You can use task panes in addition to add-in commands to enable users to interact with your solution. Clients that do not support add-in commands (Office 2013 and Office for iPad) will run your add-in as a task pane. Users launch task pane add-ins via the **My Add-ins** button on the Insert tab.

**Task pane** [](https://github.com/OfficeDev/office-js-docs/blob/master/images/task-pane-overview.jpg)

**Extend Outlook functionality**

Outlook add-ins can extend the Office ribbon and also display contextually next to an Outlook item when you're viewing or composing it. They can work with an email message, meeting request, meeting response, meeting cancellation, or appointment when a user is viewing a received item or replying or creating a new item.

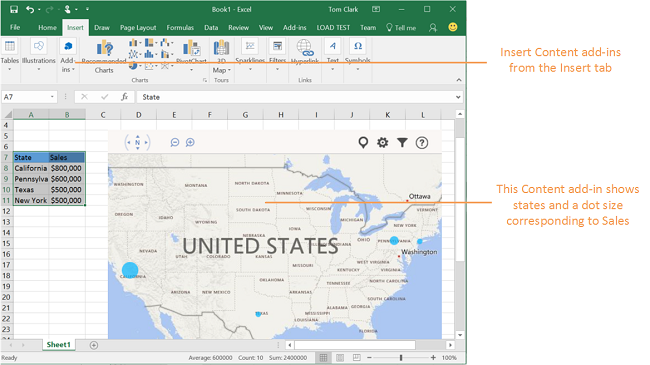
Outlook add-ins can access contextual information from the item, such as address or tracking ID, and then use that data to access additional information on the server and from web services to create compelling user experiences. In most cases, an Outlook add-in runs without modification on the various supporting host applications, including Outlook, Outlook for Mac, Outlook Web App, and Outlook Web App for devices, to provide a seamless experience on the desktop, web, and tablet and mobile devices.

For an overview of Outlook add-ins, see [Outlook add-ins overview](https://docs.microsoft.com/en-us/outlook/add-ins/).

**Create new objects in Office documents**

You can embed web-based objects within Excel and PowerPoint documents – called content add-ins. With content add-ins, you can integrate rich, web-based data visualizations, media (such as a YouTube video player or a picture gallery), and other external content.

**Content add-in**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview05.png)

**Office JavaScript APIs**

The Office JavaScript APIs contain objects and members for building add-ins and interacting with Office content and web services. The Excel, Outlook, Word, PowerPoint, OneNote and Project JavaScript APIs provide host-specific object models that you can use in an Office Add-in. These APIs provide access to well-known objects such as paragraphs and workbooks, which makes it easier to create an add-in for a specific host.

**Next steps**

To learn more about how to start building your Office Add-in, try out [Get started](https://dev.office.com/getting-started/addins) experience. You can start building add-ins right away using [Visual Studio](https://github.com/OfficeDev/office-js-docs/blob/master/docs/get-started/create-and-debug-office-add-ins-in-visual-studio.md) or [your own tools](https://github.com/OfficeDev/office-js-docs/blob/master/docs/get-started/create-an-office-add-in-using-any-editor).

To start planning solutions that create effective and compelling user experiences, get familiar with the [design guidelines](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-design) and [best practices](https://github.com/OfficeDev/office-js-docs/blob/master/docs/add-ins/overview/add-in-development-best-practices) for Office Add-ins.

**Requirements for running Office Add-ins**

This article describes the software and device requirements for running Office Add-ins.

**Note:** When you build your add-in, if you plan to [publish](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md) your add-in to the Office Store, make sure that you conform to the [Office Store validation policies](https://msdn.microsoft.com/en-us/library/jj220035.aspx). For example, to pass validation, your add-in must work across all platforms that support the methods that you define (for more information, see [section 4.12](https://msdn.microsoft.com/en-us/library/jj220035.aspx#Anchor_3) and the [Office Add-in host and availability page](https://dev.office.com/add-in-availability)).

For a high-level view of where Office Add-ins are currently supported, see the [Office Add-in host and platform availability](http://dev.office.com/add-in-availability)page.

**Server requirements**

To be able to install and run any Office Add-in, you first need to deploy the manifest and webpage files for the UI and code of your add-in to the appropriate server locations.

For all types of add-ins (content, Outlook, and task pane add-ins and add-in commands), you need to deploy your add-in's webpage files to a web server, or web hosting service, such as [Microsoft Azure](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/host-an-office-add-in-on-microsoft-azure.md).

**Note:** When you develop and debug an add-in in Visual Studio, Visual Studio deploys and runs your add-in's webpage files locally with IIS Express, and doesn't require an additional web server.

For content and task pane add-ins, in the supported Office host applications - Access web apps, Word, Excel, PowerPoint, or Project - you also need an [add-in catalog](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish-task-pane-and-content-add-ins-to-an-add-in-catalog.md) on SharePoint to upload the add-in's XML manifest file.

To test and run an Outlook add-in, the user's Outlook email account must reside on Exchange 2013 or later, which is available through Office 365, Exchange Online, or through an on-premises installation. The user or administrator installs manifest files for Outlook add-ins on that server.

**Note:** POP and IMAP email accounts in Outlook don't support Office Add-ins.

**Client requirements: Windows desktop and tablet**

The following software is required for developing an Office Add-in for the supported Office desktop clients or web clients that run on Windows-based desktop, laptop, or tablet devices:

* For Windows x86 and x64 desktops, and tablets such as Surface Pro:
  + The 32- or 64-bit version of Office 2013 or a later version, running on Windows 7 or a later version.
  + Excel 2013, Outlook 2013, PowerPoint 2013, Project Professional 2013, Project 2013 SP1, Word 2013, or a later version of the Office client, if you are testing or running an Office Add-in specifically for one of these Office desktop clients. Office desktop clients can be installed on premises or via Click-to-Run on the client computer.

If you have a valid Office 365 subscription and you do not have access to Office 2013, you can download it via one the following CDN links:

* + - Office 2013 for Business: <https://c2rsetup.officeapps.live.com/c2r/download.aspx?productReleaseID=O365BusinessRetail&platform=X86&language=en-us&version=O15GA&source=O15OLSO365>
    - Office 2013 for Home: <https://c2rsetup.officeapps.live.com/c2r/download.aspx?productReleaseID=O365HomePremRetail&platform=X86&language=en-us&version=O15GA&source=O15OLSO365>
* Internet Explorer 11 or later, which must be installed but doesn't have to be the default browser. To support Office Add-ins, the Office client that acts as host uses browser components that are part of Internet Explorer 11 or later.
* One of the following as the default browser: Internet Explorer 11 or later, or the latest version of Microsoft Edge, Chrome, Firefox, or Safari (Mac OS).
* An HTML and JavaScript editor such as Notepad, [Visual Studio and the Microsoft Developer Tools](https://www.visualstudio.com/features/office-tools-vs), or a third-party web development tool.

**Client requirements: OS X desktop**

Outlook for Mac, which is distributed as part of Office 365, supports Outlook add-ins. Running Outlook add-ins on Outlook for Mac has the same requirements as Outlook for Mac itself: the operating system must be at least OS X v10.10 "Yosemite". Because Outlook for Mac uses WebKit as a layout engine to render the add-in pages, there is no additional browser dependency.

The following are the minimum client versions of Office for Mac that support Office Add-ins:

* Word for Mac version 15.18 (160109)
* Excel for Mac version 15.19 (160206)
* PowerPoint for Mac version 15.24 (160614)

**Client requirements: Browser support for Office Online web clients and SharePoint**

Any browser that supports ECMAScript 5.1, HTML5, and CSS3, such as Internet Explorer 11 or later, or the latest version of Microsoft Edge, Chrome, Firefox, or Safari (Mac OS).

**Client requirements: non-Windows smartphone and tablet**

Specifically for OWA for Devices, and Outlook Web App running in a browser on smartphones and non-Windows tablet devices, the following software is required for testing and running Outlook add-ins.

| **Host application** | **Device** | **Operating system** | **Exchange account** | **Mobile browser** |
| --- | --- | --- | --- | --- |
| OWA for Android | Android smartphones. Technically, those devices considered as "small" or "normal" by [Android OS](https://developer.android.com/guide/practices/screens_support.html). | Android 4.4 KitKat or later | On the latest update of Office 365 for business or Exchange Online | Native add-in for Android, browser not applicable |
| OWA for iPad | iPad 2 or later | iOS 6 or later | On the latest update of Office 365 for business or Exchange Online | Native add-in for iOS, browser not applicable |
| OWA for iPhone | iPhone 4S or later | iOS 6 or later | On the latest update of Office 365 for business or Exchange Online | Native add-in for iOS, browser not applicable |
| Outlook Web App | iPhone 4 or later, iPad 2 or later, iPod Touch 4 or later | iOS 5 or later | On Office 365, Exchange Online, or on premise on Exchange Server 2013 or later | Safari |

**Best practices for developing Office Add-ins**

Effective add-ins offer unique and compelling functionality that extends Office applications in a visually appealing way. To create a great add-in, provide an engaging first-time experience for your users, design a first-class UI experience, and optimize your add-in's performance. Apply the best practices described in this article to create add-ins that help your users complete their tasks quickly and efficiently.

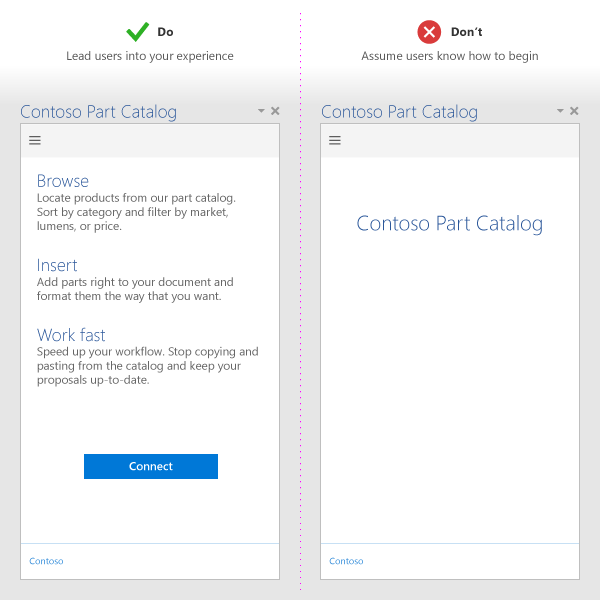
**Note:** When you build your add-in, if you plan to [publish](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md) your add-in to the Office Store, make sure that you conform to the [Office Store validation policies](https://msdn.microsoft.com/en-us/library/jj220035.aspx). For example, to pass validation, your add-in must work across all platforms that support the methods that you define (for more information, see [section 4.12](https://msdn.microsoft.com/en-us/library/jj220035.aspx#Anchor_3) and the [Office Add-in host and availability page](https://dev.office.com/add-in-availability)).

**Provide clear value**

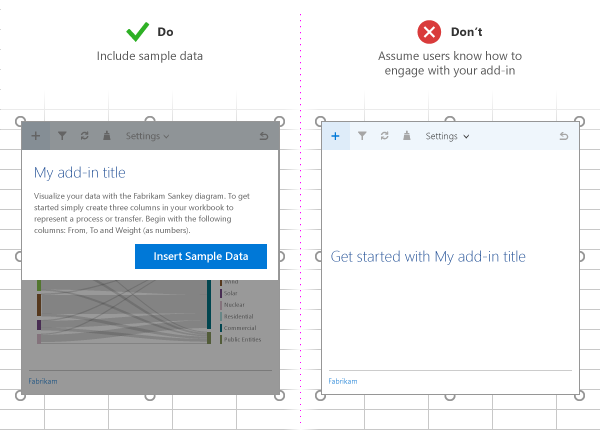
* Create add-ins that help users complete tasks quickly and efficiently. Focus on scenarios that make sense for Office applications. For example:
* Make core authoring tasks faster and easier, with fewer interruptions.
* Enable new scenarios within Office.
* Embed complementary services within Office hosts.
* Improve the Office experience to enhance productivity.
* Make sure that the value of your add-in is clear to users right away by [creating an engaging first run experience](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-development-best-practices.md#create-an-engaging-first-run-experience).
* Create an [effective Office Store listing](http://msdn.microsoft.com/library/c66a6e6b-2e96-458f-8f8c-2a499fe942c9%28Office.15%29.aspx). Make the benefits of your add-in clear in your title and description. Don't rely on your brand to communicate what your add-in does.

**Create an engaging first-run experience**

* Engage new users with a highly usable and intuitive first experience. Note that users are still deciding whether to use or abandon an add-in after they download it from the store.
* Make the steps that the user needs to take to engage with your add-in clear. Use videos, placemats, paging panels, or other resources to entice users.
* Reinforce the value proposition of your add-in on launch, rather than just asking users to sign in.
* Provide teaching UI to guide users and make your UI personal.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/586202ad-333b-417c-ad31-cc6eb952b239.png)

* If your content add-in binds to data in the user's document, include sample data or a template to show users the data format to use.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/7de2215f-ccef-4f82-aa9d-babcbddae0c6.png)

* Offer [free trials](https://msdn.microsoft.com/en-us/library/dn456317.aspx#Anchor_1). If your add-in requires a subscription, make some functionality available without a subscription.
* Make signup simple. Prefill information (email, display name) and skip email verifications.
* Avoid pop ups. If you have to use them, guide the user to enable your pop up.
* Use [single sign-on (SSO) authentication](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/authenticate-a-user-with-an-identity-token.md).

For templates that illustrate patterns that you can apply as you develop your first-run experience, see [UX design patterns for Office Add-ins](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code).

**Use add-in commands**

* Provide relevant UI entry points for your add-in by using add-in commands. For details, including design best practices, see [add-in commands](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-commands.md).

**Apply UX design principles**

* Ensure that the look and feel and functionality of your add-in complements the Office experience. Use [Office UI Fabric](https://dev.office.com/fabric).
* Favor content over chrome. Avoid superfluous UI elements that don't add value to the user experience.
* Keep users in control. Ensure that users understand important decisions, and can easily reverse actions the add-in performs.
* Use branding to inspire trust and orient users. Do not use branding to overwhelm or advertise to users.
* Avoid scrolling. Optimize for 1366 x 768 resolution.
* Do not include unlicensed images.
* Use [clear and simple language](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/voice-guidelines.md) in your add-in.
* Account for [accessibility](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/accessibility-guidelines.md) - make your add-in easy for all users to interact with, and accommodate assistive technologies such as screen readers.
* Design for all platforms and input methods, including mouse/keyboard and [touch](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-development-best-practices.md#optimize-for-touch). Ensure that your UI is responsive to different form factors.

For templates that apply design principles that you can use and customize as you develop your add-in, see [UX design patterns for Office Add-ins](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code).

**Optimize for touch**

* Use the [Context.touchEnabled](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.touchenabled.md) property to detect whether the host application your add-in runs on is touch enabled.

**Note** This property is not supported in Outlook.

* Ensure that all controls are appropriately sized for touch interaction. For example, buttons have adequate touch targets, and input boxes are large enough for users to enter input.
* Do not rely on non-touch input methods like hover or right-click.
* Ensure that your add-in works in both portrait and landscape modes. Be aware that on touch devices, part of your add-in might be hidden by the soft keyboard.
* Test your add-in on a real device by using [sideloading](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/sideload-an-office-add-in-on-ipad-and-mac.md).

**Note** If you're using [Office UI Fabric](https://github.com/OfficeDev/Office-UI-Fabric) for your design elements, many of these elements are taken care of.

**Optimize and monitor add-in performance**

* Create the perception of fast UI responses. Your add-in should load in 500 ms or less.
* Ensure that all user interactions respond in under one second.
* Provide loading indicators for long-running operations.
* Use a CDN to host images, resources, and common libraries. Load as much as you can from one place.
* Follow standard web practices to optimize your web page. In production, use only minified versions of libraries. Only load resources that you need, and optimize how resources are loaded.
* If operations take time to execute, provide feedback to users. Note the thresholds listed in the following table. See also [Resource limits and performance optimization for Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/resource-limits-and-performance-optimization.md)

| **Interaction class** | **Target** | **Upper bound** | **Human perception** |
| --- | --- | --- | --- |
| Instant | <=50 ms | 100 ms | No noticeable delay. |
| Fast | 50-100 ms | 200 ms | Minimally noticeable delay. No feedback necessary. |
| Typical | 100-300 ms | 500 ms | Quick, but too slow to be described as fast. No feedback necessary. |
| Responsive | 300-500 ms | 1 second | Not fast, but still feels responsive. No feedback necessary. |
| Continuous | >500 ms | 5 seconds | Medium wait, no longer feels responsive. Might need feedback. |
| Captive | >500 ms | 10 seconds | Long, but not long enough to do something else. Might need feedback. |
| Extended | >500 ms | >10 seconds | Long enough to do something else while waiting. Might need feedback. |
| Long running | >5 ms | >1 minute | Users will certainly do something else. |

* Monitor your service health, and use telemetry to monitor user success.

**Market your add-in**

* Publish your add-in to the [Office Store](http://msdn.microsoft.com/library/ff075782-1303-4517-91cc-b3d730e9b9ae%28Office.15%29.aspx) and [promote it](http://msdn.microsoft.com/library/b19e21f8-76f5-44e1-9971-bef79cad4c71%28Office.15%29.aspx) from your website. Create an [effective Office Store listing](http://msdn.microsoft.com/library/c66a6e6b-2e96-458f-8f8c-2a499fe942c9%28Office.15%29.aspx).
* Use succinct and descriptive add-in titles. Include no more than 128 characters.
* Write short, compelling descriptions of your add-in. Answer the question "What problem does this add-in solve?".
* Convey the value proposition of your add-in in your title and description. Don't rely on your brand.
* Create a website to help users find and use your add-in.
* **Changelog for Office Add-ins**
* This changelog covers what's changed on the Office Add-in platform, including updates to the APIs as well as new features and tools available to Office developers.
* **April 2017**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Office Add-in tooling | Released the [Office Add-in Validator](https://github.com/OfficeDev/office-addin-validator) to help developers validate the add-in manifests. | [Validating your Office Add-in manifest just got easier - Introducing the Office Add-in Validator](https://dev.office.com/blogs/creating-add-ins-gets-easier-announcing-updates-to-yo-officehttps:/dev.office.com/blogs/introducing-the-office-add-in-validator) |
| Office Add-in publishing | Released centralized deployment for deploying add-ins to organizations via the Office 365 admin center. | [Use centralized deployment to publish Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/publish/centralized-deployment) |

* **March 2017**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | ExcelApi 1.5 beta and 1.6 beta are now available for Office Online. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |

* **February 2017**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Office Add-in Tooling | Updated the [Office Yeoman Generator](https://github.com/OfficeDev/generator-office) to make it easier for developers to get started with Office Add-ins. | [Creating Add-ins Gets Easier - Announcing Updates to Yo Office](https://dev.office.com/blogs/creating-add-ins-gets-easier-announcing-updates-to-yo-office) |
| Excel APIs | ExcelApi 1.5 beta and 1.6 beta are now available for Windows. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |

* **January 2017**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | ExcelApi 1.4 is now available for Windows and Office Online. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |

* **September 2016**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | ExcelApi 1.3 is now available for Office Online. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |
| Word APIs | WordApi 1.1 and 1.2 is now available for Office Online. | [Word JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/word-api-requirement-sets) |
| OneNote APIs | Introduced OneNoteApi 1.1 for OneNote Online. | [OneNote JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/onenote-api-requirement-sets) |

* **August 2016**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | ExcelApi 1.3 is now available for Windows. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |

* **February 2016**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Dialog APIs | Introduced DialogApi 1.1 for Office 2016 for Windows. | [Dialog API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/dialog-api-requirement-sets) |

* **January 2016**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | Introduced ExcelApi 1.1 and 1.2 for Office Online.  Released ExcelApi 1.2 for Office 2016 for Windows. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |
| Word APIs | Released WordApi 1.2 for Office 2016 for Windows.  Introduced WordApi 1.1 and 1.2 for Office 2016 for iPad and Office 2016 for Mac. | [Word JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/word-api-requirement-sets) |

* **September 2015**

| **Category** | **Description** | **Link** |
| --- | --- | --- |
| Excel APIs | Introduced ExcelApi 1.1 for Excel 2016 for Windows. | [Excel JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets) |
| Word APIs | Introduced WordApi 1.1 for Word 2016 for Windows. | [Word JavaScript API requirement sets](https://dev.office.com/reference/add-ins/requirement-sets/word-api-requirement-sets) |

# Create and debug Office Add-ins in Visual Studio

This article describes how to use Visual Studio to create your first Office Add-in. The steps in this article based on Visual Studio 2015. If you're using another version of Visual Studio, the procedures might vary slightly.

**Note:** To get started with an add-in for OneNote, see [Build your first OneNote add-in](https://github.com/OfficeDev/office-js-docs/blob/master/docs/onenote/onenote-add-ins-getting-started.md).

## Create an Office Add-in project in Visual Studio

To get started, make sure you have the [Office Developer Tools](https://www.visualstudio.com/features/office-tools-vs.aspx) installed, and a version of Microsoft Office. You can join the [Office 365 Developer program](https://dev.office.com/devprogram), or follow these instructions to get the [latest version](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/install-latest-office-version.md).

1. On the Visual Studio menu bar, choose **File** > **New** > **Project**.
2. In the list of project types under **Visual C#** or **Visual Basic**, expand **Office/SharePoint**, choose **Web Add-ins**, and then select one of the Add-in projects.
3. Name the project, and then choose **OK** to create the project.
4. Visual Studio creates the project, and its files appear in **Solution Explorer**. The default Home.html page opens in Visual Studio.

In Visual Studio 2015, some of the add-in project templates have been updated to reflect additional functionality:

* Content add-ins can appear in the body of Access and PowerPoint documents, in addition to Excel spreadsheets. You can also choose the Basic Project option to create a basic content add-in project with minimal starter code, or the Document Visualization Project option (for Access and Excel only) to create a more full-featured content add-in that includes starter code to visualize and bind to data.
* Outlook add-ins include options not just for including your add-in in email messages or appointments, but also for specifying whether the add-in is available when an email message or appointment is being composed as well as read.

**Note** In Visual Studio most options are understandable from their descriptions except for the **Email Message** checkbox. Use that checkbox if you want to create an Outlook add-in that appears not just with mail items, but also with meeting requests, responses, and cancellations.

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

| **Project** | **Description** |
| --- | --- |
| Add-in project | Contains only an XML manifest file, which contains all the settings that describe your add-in. These settings help the Office host 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. You change these settings any time by using the Manifest editor. |
| 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, you'll have to find a server to host this project.To learn more about ASP.NET web application projects, see [ASP.NET Web Projects](http://msdn.microsoft.com/en-us/library/cdcd712f-96b0-4165-8b5d-9d0566650a28%28Office.15%29.aspx). |

## Modify your add-in settings

To modify the settings of your add-in, edit the XML manifest file of the project. In **Solution Explorer**, expand the add-in project node, expand the folder that contains the XML manifest, and choose the XML manifest. You can point to any element in the file to view a tooltip that describes the purpose of the element. For more information about the manfiest file, see [Office Add-ins XML manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-manifests.md).

## Develop the contents of your add-in

While the add-in project lets you modify the settings that describe your add-in, the web application provides the content that appears in the add-in.

The web application project contains a default HTML page and JavaScript file that you can use to get started. The project also contains a JavaScript file that is common to all pages that you add to your project. These files are convenient because they contain references to other JavaScript libraries including the JavaScript API for Office.

As your add-in becomes more sophisticated, you can add more HTML and JavaScript files. You can use the contents of the default HTML and JavaScript files as examples of the types of references you might want to add to other pages in your project to make them work with your add-in. The following table describes default HTML and JavaScript files.

| **File** | **Description** |
| --- | --- |
| **Home.html** | Located in the **Home** folder of the project, this is default HTML page of the add-in. This page appears as the first page inside of the add-in when it is activated in a document, email message or appointment item. This file is convenient because it contains all of the file references that you need to get started. When you are ready to create your first add-in, just add your HTML code to this file. |
| **Home.js** | Located in the **Home** folder of the project, this is the JavaScript file associated with the Home.js page. You can place any code that is specific to the behavior of the Home.html page in the Home.js file. The Home.js file contains some example code to get you started. |
| **App.js** | Located in the **Add-in** folder of the project, this is the default JavaScript file of the entire add-in. You can place code that is common to the behavior of multiple pages of your add-in in the App.js file. The App.js file contains some example code to get you started. |

**Note** You don't have to use these files. Feel free to add other files to the project and use those instead. If you want another HTML file to appear as the initial page of the add-in, open the manifest editor, and then point the**SourceLocation** property to the name of the file.

## Debug your add-in

When you are ready to start your add-in, review build and debug related properties, and then start the solution.

### Review the build and debug properties

Before you start the solution, verify that Visual Studio will open the host application that you want. That information appears in the property pages of the project along with several other properties that relate to building and debugging the add-in.

### To open the property pages of a project

1. In **Solution Explorer**, choose the project name.
2. On the menu bar, choose **View**, **Properties Window**.

The following table describes the properties of the project.

| **Property** | **Description** |
| --- | --- |
| **Start Action** | Specifies whether to debug your add-in in an Office desktop client or in an Office Online client in the specified browser. |
| **Start Document** (Content and task pane add-ins only) | Specifies what document to open when you start the project. |
| **Web Project** | Specifies the name of the web project associated with the add-in. |
| **Email Address** (Outlook add-ins only) | Specifies the email address of the user account in Exchange Server or Exchange Online that you want to test your Outlook add-in with. |
| **EWS Url** (Outlook add-ins only) | Exchange Web service URL (For example: <https://www.contoso.com/ews/exchange.aspx>). |
| **OWA Url** (Outlook add-ins only) | Outlook Web App URL (For example: <https://www.contoso.com/owa>). |
| **User name** (Outlook add-ins only) | Specifies the name of your user account in Exchange Server or Exchange Online. |
| **Project File** | Specifies the name of the file containing build, configuration, and other information about the project. |
| **Project Folder** | The location of the project file. |

### Use an existing document to debug the add-in (content and task pane add-ins only)

You can add documents to the add-in project. If you have a document that contains test data that you want to use with your add-in, Visual Studio opens that document for you when you start the project.

### To use an existing document to debug the add-in

1. In **Solution Explorer**, choose the add-in project folder.

**Note** Choose the add-in project and not the web application project.

1. On the **Project** menu, choose **Add Existing Item**.
2. In the **Add Existing Item** dialog box, locate and select the document that you want to add.
3. Choose the **Add** button to add the document to your project.
4. In **Solution Explorer**, open the shortcut menu for the project, and then choose **Properties**.

The property pages for the project appear.

1. In the **Start Document** list, choose the document that you added to the project, and then choose the **OK** button to close the property pages.

### Start the solution

Visual Studio will automatically build the solution when you start it. You can start the solution from the **Menu** bar by choosing **Debug**, **Start**.

**Note** If script debugging isn't enabled in Internet Explorer, you won't be able to start the debugger in Visual Studio. You can enable script debugging by opening the **Internet Options** dialog box, choosing the **Advanced** tab, and then clearing the **Disable Script Debugging (Internet Explorer)** and **Disable Script Debugging (Other)** check boxes.

Visual Studio builds the project and does the following:

1. Creates a copy of the XML manifest file and adds it to ProjectName\Output directory. The host application consumes this copy when you start Visual Studio and debug the add-in.
2. Creates a set of registry entries on your computer that enable the add-in to appear in the host application.
3. Builds the web application project, and then deploys it to the local IIS web server ([http://localhost](http://localhost/)).

Next, Visual Studio does the following:

1. Modifies the [SourceLocation](http://msdn.microsoft.com/en-us/library/e6ea8cd4-7c8b-1da7-d8f8-8d3c80a088bc%28Office.15%29.aspx) element of the XML manifest file by replacing the ~remoteAppUrl token with the fully qualified address of the start page (for example, <http://localhost/MyAgave.html>).
2. Starts the web application project in IIS Express.
3. Opens the host application.

Visual Studio doesn't show validation errors in the **OUTPUT** window when you build the project. Visual Studio reports errors and warnings in the **ERRORLIST** window as they occur. Visual Studio also reports validation errors by showing wavy underlines (known as squiggles) of different colors in the code and text editor. These marks notify you of problems that Visual Studio detected in your code. For more information, see [Code and Text Editor](http://go.microsoft.com/fwlink/?LinkID=128497). For more information about how to enable or disable validation, see:

* [Options, Text Editor, JavaScript, IntelliSense](http://go.microsoft.com/fwlink/?LinkID=238779)
* [How to: Set Validation Options for HTML Editing in Visual Web Developer](http://msdn.microsoft.com/en-us/library/vstudio/0byxkfet%28v=vs.100%29.aspx)
* [CSS, see Validation, CSS, Text Editor, Options Dialog Box](http://go.microsoft.com/fwlink/?LinkID=238780)

To review the validation rules of the XML manifest file in your project, see [Office Add-ins XML manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-manifests.md).

### Show an add-in in Excel, Word, or Project and step through your code

If you set the **Start Document** property of the add-in project to Excel or Word, Visual Studio creates a new document and the add-in appears. If you set the **Start Document** property of the add-in project to use an existing document, Visual Studio opens the document, but you have to insert the add-in manually. If you set the **Start Document** to **Microsoft Project**, you also have to insert the add-in manually.

### To show an Office Add-in in Excel or Word

1. In Excel or Word, on the **Insert** tab, choose **Office Add-ins**.
2. In the list that appears, choose your add-in.

### To show an Office Add-in in Project

1. In Project, on the **Project** tab, choose **Office Add-ins**.
2. In the list that appears, choose your add-in.

In Visual Studio, you can then set break-points. Then, as you interact with your add-in and step through the code in your HTML, JavaScript, and C# or VB code files.

### Show the Outlook add-in in Outlook and step through your code

To view the add-in in Outlook, open an email message or appointment item.

Outlook activates the add-in for the item as long as the activation criteria are met. The add-in bar appears at the top of the Inspector window or Reading Pane, and your Outlook add-in appears as a button in the add-in bar. If your add-in has an add-in command, a button will appear in the ribbon, either in the default tab or a specified custom tab, and the add-in will not appear in the add-in bar.

To view your Outlook add-in, choose the button for your Outlook add-in.

In Visual Studio, you can set break-points. Then, as you interact with your Outlook add-in and step through the code in your HTML, JavaScript, and C# or VB code files.

You can also change your code and review the effects of those changes in your Outlook add-in without having to close the Office Add-in and start the project again. In Outlook, just open the shortcut menu for the Outlook add-in, and then choose**Reload**.

### Modify code and continue to debug the add-in without having to start the project again

You can change your code and review the effects of those changes in your add-in without having to close the host application and start the project again. After you change your code, open the shortcut menu for the add-in, and then choose **Reload**. When you reload the add-in it becomes disconnected with the Visual Studio debugger. Therefore, you can view the effects of your change, but you cannot step through your code again until you attach the Visual Studio debugger to all of the available Iexplore.exe processes.

### To attach the Visual Studio debugger to all of the available Iexplore.exe processes

1. In Visual Studio, choose **DEBUG**, **Attach to Process**.
2. In the **Attach to Process** dialog box, choose all of the available **Iexplore.exe** processes, and then choose the **Attach**button.

# Get JavaScript IntelliSense in Visual Studio 2017

When you use Visual Studio 2017 to develop Office Add-ins, you can use JSDoc to enable IntelliSense for your JavaScript variables, objects, parameters, and return values. This article provides an overview of JSDoc and how you can use it to create IntellSense in Visual Studio. For more details, see [JavaScript IntelliSense](https://docs.microsoft.com/en-us/visualstudio/ide/javascript-intellisense) and [JSDoc support in JavaScript](https://github.com/Microsoft/TypeScript/wiki/JsDoc-support-in-JavaScript).

## Office.js type definitions

You need to provide the definitions of the types in Office.js to Visual Studio. To do this, you can:

* Have a local copy of the Office.js files in a folder in your solution named \Office\1\. The Office Add-in project templates in Visual Studio add this local copy when you create an add-in project.
* Use an online version of Office.js by adding a tsconfig.json file to the root of the web application project in the add-in solution. The file should include the following content.
* {
* "compilerOptions": {
* "allowJs": true, // These settings apply to JavaScript files also.
* "noEmit": true // Do not compile the JS (or TS) files in this project.
* },
* "exclude": [
* "node\_modules", // Don't include any JavaScript found under "node\_modules".
* "Scripts/Office/1" // Suppress loading all the JavaScript files from the Office NuGet package.
* ],
* "typeAcquisition": {
* "enable": true, // Enable automatic fetching of type definitions for detected JavaScript libraries.
* "include": [ "office-js" ] // Ensure that the "Office-js" type definition is fetched.
* }

}

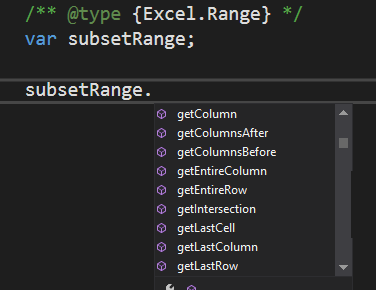
## JSDoc syntax

The basic technique is to precede the variable (or parameter, and so on) with a comment that identifies its data type. This allows IntelliSense in Visual Studio to infer its members. The following are examples.

### Variable

/\*\* @type {Excel.Range} \*/

var subsetRange;

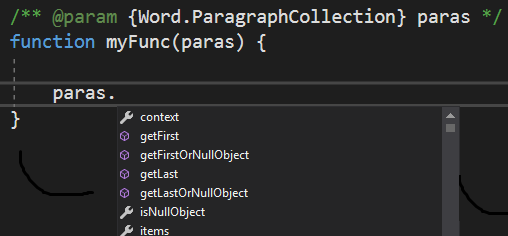
[](https://github.com/OfficeDev/office-js-docs/blob/master/images/IntellisenseVS17Var.png)

### Parameter

/\*\* @param {Word.ParagraphCollection} paragraphs \*/

function myFunc(paragraphs){

}

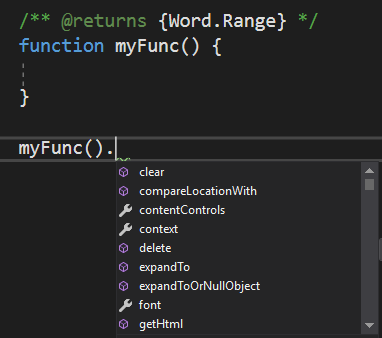
[](https://github.com/OfficeDev/office-js-docs/blob/master/images/IntellisenseVS17Param.png)

### Return value

/\*\* @returns {Word.Range} \*/

function myFunc() {

}

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/IntellisenseVS17Return.png)

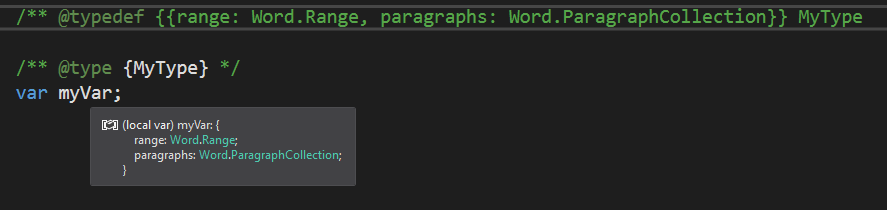
### Complex types

/\*\* @typedef {{range: Word.Range, paragraphs: Word.ParagraphCollection}} MyType

/\*\* @returns {MyType} \*/

function myFunc() {

}



# Create an Office Add-in using any editor

You can use the Yeoman generator to create your Office Add-in. The Yeoman generator provides the project scaffolding and build management. The manifest.xml file tells the Office application where your add-in is located and how you want it to appear. The Office application takes care of hosting it within Office.

**Note:** These instructions use Terminal on a Mac, but you can also use other shell environments.

## Prerequisites for the Yeoman generator

To install the Yeoman Office generator, you must have [git](https://git-scm.com/downloads) and node.js installed on your computer. If you're on a Mac, we recommend that you use [Node Version Manager](https://github.com/creationix/nvm) to install node.js with the right permissions. If you're on Windows, you can install node.js from [nodejs.org](https://nodejs.org/en/).

**Note:** If you're on Windows, use the default values when you install git, with the following exceptions:

* Use git from the Windows command prompt
* Use the Windows default console window

After you install node.js, open a Terminal and install the generator globally.

npm install -g yo generator-office

## Create the default files for your add-in

The Yeoman generator runs in the directory where you want to scaffold the project. Before you develop an Office Add-in, you should first create a folder for your project.

In Terminal, move to the parent folder where you want to create your project. Then use the following commands create a new folder named myHelloWorldaddin and shift the current directory to it:

mkdir myHelloWorldaddin

cd myHelloWorldaddin

Use the Yeoman generator to create the add-in of your choice. The steps in this article create a simple task pane add-in. To run the generator, enter the following command:

yo office

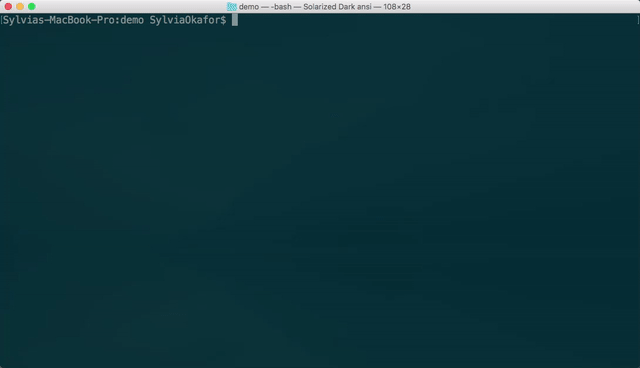
**Yeoman generator input for an add-in**

The generator will prompt you for the following:

* New subfolder -- use N
* Add-in name -- use myHelloWorldaddin
* The supported Office application - you can choose any application
* Create new add-in -- use Yes, I want a new add-in.
* Add [TypeScript](https://www.typescriptlang.org/) -- use N
* Choose framework -- use Jquery

**Note:** If you want to create an Office Add-in that uses Office UI Fabric React, enter the following:

* Add [TypeScript](https://www.typescriptlang.org/) -- use Y
* Choose framework -- use React

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/gettingstarted-fast.gif)

This creates the structure and basic files for your add-in.

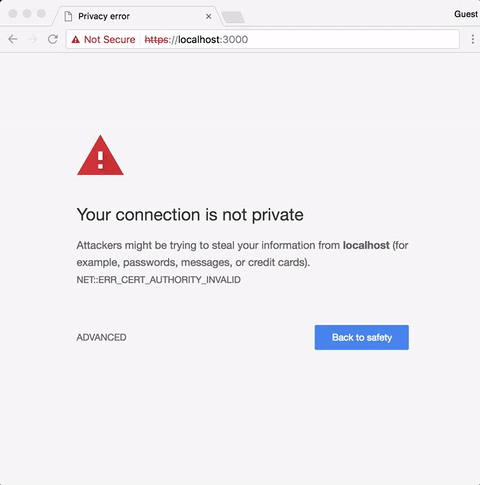
## Hosting your Office Add-in

Office Add-ins must be hosted, even in development, via HTTPS. Yo Office creates a bsconfig.json, which uses Browsersync to make it faster for you to tweak and test your add-in by synchronizing file changes across multiple devices.

Launch the local HTTPS site on [https://localhost:3000](https://localhost:3000/) by typing the following command in your console:

npm start

Browsersync will start a HTTPS server, and launch the index.html file in your project. You will see an error that states "There is a problem with this website's security certificate."

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/ssl-chrome-bypass.gif)

This error occurs because Browsersync includes a self-signed SSL certificate that your development environment must trust. For information about how to resolve this error, see [adding self-signed certificates](https://github.com/OfficeDev/generator-office/blob/master/src/docs/ssl.md).

## Sideload the add-in into Office

You can use sideloading to install your add-in for testing within the Office clients:

* [Sideload Office Add-ins for testing](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/create-a-network-shared-folder-catalog-for-task-pane-and-content-add-ins.md)
* [Sideload Office Add-ins on iPad and Mac for testing](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/sideload-an-office-add-in-on-ipad-and-mac.md)
* [Sideload Outlook add-ins for testing](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/testing-and-tips.md)

## Develop your Office Add-in

You can use any text editor to develop the files for your custom Office Add-in.

**Important:** The manifest-myHelloWorldaddin.xml file tells the Office client applications how to interact with your add-in. The value in the <id> tag is a GUID that Yo Office creates when it generates the project. Do not change the GUID for your add-in. If the host is Azure, the SourceLocation value will be a URL that is similar to https:// [name-of-your-web-app].azurewebsites.net/[path-to-add-in]. If you are using the self-hosted option, as in this example, it will be [*https://localhost:3000/[path-to-add-in]*](https://localhost:3000/%5Bpath-to-add-in%5D).

## Debug your Office Add-in

You can debug your add-in in several ways:

* Attach a debugger from the task pane (Office 2016 for Windows).
* Use your browser's developer tools.
* Use F12 developer tools in Windows 10.

### Attach debugger from the task pane

In Office 2016 for Windows, Build 77xx.xxxx or later, you can attach the debugger from the task pane. The attach debugger feature will directly attach the debugger to the correct Internet Explorer process for you. You can attach a debugger regardless of whether you are using Yeoman Generator, Visual Studio Code, node.js, Angular, or another tool.

For more information, see [Attach debugger from the task pane](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/attach-debugger-from-task-pane.md).

### Browser developer tools

You can use the Office web clients and open the browser's developer tools to debug your add-in the way you debug any other client-side JavaScript application.

### F12 developer tools on Windows 10

If you're using the Office desktop client on Windows 10, you can [Debug add-ins using F12 developer tools on Windows 10](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/debug-add-ins-using-f12-developer-tools-on-windows-10.md).

# Design your Office Add-ins

Office Add-ins extend the Office experience by providing contextual functionality that users can access within Office clients. Add-ins empower users to get more done by enabling them to access third-party functionality within Office, without costly context switches.

Your add-in UX design must integrate seamlessly with Office to provide an efficient, natural interaction for your users. Take advantage of [add-in commands](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-commands.md) to provide access to your add-in and apply the best practices that we recommend when you create custom HTML-based UI.

Office applications follow a general set of interaction guidelines. The apps share content and have elements that look and behave similarly. This commonality is built on a set of design principles. The principles help the Office team create interfaces that support customers’ tasks. Understanding and following them will help you support your customers’ goals inside of Office.

Follow the Office design principles to create positive add-in experiences:

* **Design explicitly for Office.** The functionality, look and feel of an add-in must harmoniously complement the Office experience. Add-ins should feel native. They should fit seamlessly into Word on an iPad or PowerPoint Online. A well-designed add-in will be an appropriate blend of your experience, the platform and the Office application. Consider using Office UI Fabric as your design language. Apply document and UI theming where appropriate.
* **Focus on a few key tasks; do them well.** Help customers get one job done without getting in the way of other jobs. Provide real value to customers. Focus on common use cases, pick carefully those that benefit users most when interacting with Office documents.
* **Favor content over chrome.** Allow customers’ page, slide or spreadsheet to remain the focus of the experience. An add-in is an auxiliary interface. No accessory chrome should interfere with the add-in’s content and functionality. Brand your experience wisely. We know it is important to provide users with a unique, recognizable experience but avoid distraction. Strive to keep the focus on content and task completion, not brand attention.
* **Make it enjoyable and keep users in control.** People enjoy using products that are both functional and visual appealing. Craft your experience carefully. Get the details right by considering every interaction and visual detail. Allow users to control their experience. The necessary steps to complete a task must be clear and relevant. Important decisions should be easy to understand. Actions should be easily reversible. An add-in is not a destination – it’s an enhancement to Office functionality.
* **Design for all platforms and input methods**. Add-ins are designed to work on all the platforms that Office supports, and your add-in UX should be optimized to work across platforms and form factors. Support mouse/keyboard and touch input devices, and ensure that your custom HTML UI is responsive to adapt to different form factors. For more information, see [Touch](https://msdn.microsoft.com/EN-US/library/mt590883.aspx#bk_Touch).

# Office Add-in design guidelines

Enhance the user experience in your Office Add-in by developing UI that matches the Office voice, and apply accessibility guidelines to ensure that your add-in is accessible to all users.

If you plan to make your add-in [available in the Office Store](https://dev.office.com/officestore/docs/submit-to-the-office-store), make sure that your language and content complies with the [Validation policies](https://dev.office.com/officestore/docs/validation-policies).

## Voice guidelines

As you design your Office Add-ins, consider the voice that you use in your UI text and elements. Strive to match the voice and tone of the Office UI, which is conversational, engaging, and accessible to users.

To align your text with the principles of the Office voice:

* **Use a natural style.** Write the way that you speak. Avoid jargon and overly technical words and phrases. Use terms that are familiar to your users.
* **Use simple, direct language.** Use short words and sentences, and active voice in your text.
* **Be consistent.** Use the same words for the same concepts throughout.
* **Engage the user.** Address the user as "you". Avoid using third person. Use imperatives for user tasks.
* **Be helpful and empathetic.** Make your text positive, polite, supportive, and encouraging. Emphasize what users can accomplish ―- not what they can't.
* **Know your customers.** Be mindful of cultural considerations and globalization when you use idioms or colloquialisms.

## Accessibility guidelines

As you design and develop your Office Add-ins, you'll want to ensure that all potential users and customers are able to use your add-in successfully. Apply the following guidelines to ensure that your solution is accessible to all audiences.

### Design for multiple input methods

* Ensure that users can perform operations by using only the keyboard. Users should be able to move to all actionable elements on the page by using a combination of the Tab and arrow keys.
* On a mobile device, when users operate a control by touch, the device should provide useful audio feedback.
* Provide helpful labels for all interactive controls.

### Make your add-in easy to use

* Don't rely on a single attribute, such as color, size, shape, location, orientation, or sound, to convey meaning in your UI.
* Avoid unexpected changes of context, such as moving the focus to a different UI element without user action.
* Provide a way to verify, confirm, or reverse all binding actions.
* Provide a way to pause or stop media, such as audio and video.
* Do not impose a time limit for user action.

### Make your add-in easy to see

* Avoid unexpected color changes.
* Provide meaningful and timely information to describe UI elements, titles and headings, inputs, and errors. Ensure that names of controls adequately describe the intent of the control.
* Follow [standard guidelines](http://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html) for color contrast.

### Account for assistive technologies

* Avoid using features that interfere with assistive technologies, including visual, audio, or other interactions.
* Do not provide text in an image format. Screen readers cannot read text within images.
* Provide a way for users to adjust or mute all audio sources.
* Provide a way for users to turn on captions or audio description with audio sources.
* Provide alternatives to sound as a means to alert users, such as visual cues or vibrations.

### Accessibility resources

* [Web Content Accessibility Guidelines (WCAG) 2.0](http://www.w3.org/TR/wcag2ict/#REF-WCAG20)
* [Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT)](http://www.w3.org/TR/wcag2ict/)
* [European Standard on accessibility requirements for Information and Communication Technologies (ICT)](http://www.etsi.org/deliver/etsi_en/301500_301599/301549/01.00.00_20/en_301549v010000c.pdf)

# Office Add-in design language

The Office design language is a clean and simple visual system that ensures consistency across experiences. It contains a set of visual elements that define Office interfaces, including:

* A standard typeface
* A common color palette
* A set of typographic sizes and weights
* Icon guidelines
* Shared icon assets
* Animation definitions
* Common components

[Office UI Fabric](https://dev.office.com/fabric) is the official front-end framework for building with the Office design language. Using Fabric is optional, but it is the fastest way to ensure that your add-ins feel like a natural extension of Office. Take advantage of Fabric to design and build add-ins that complement Office.

Many Office Add-ins are associated with a preexisting brand. You can retain a strong brand and its visual or component language in your add-in. Look for opportunities to retain your own visual language while integrating with Office. Consider ways to swap out Office colors, typography, icons, or other stylistic elements with elements of your own brand. Consider ways to follow common add-in layouts or UX design patterns while inserting controls and components that are familiar to your customers.

Inserting a heavily branded HTML-based UI inside of Office can create dissonance for customers. Find a balance that fits seamlessly in Office but also clearly aligns with your service or parent brand. When an add-in does not fit with Office, it's often because stylistic elements conflict. For example, typography is too large and off grid, colors are contrasting or particularly loud, or animations are superfluous and behave differently than Office. The appearance and behavior of controls or components veer too far from Office standards.

## Typography

Segoe is the standard typeface for Office. Use it in your add-in to align with Office task panes, dialog boxes, and content objects. Office UI Fabric gives you access to Segoe. It provides a full type ramp of Segoe with many variations - across font weight and size - in convenient CSS classes. Not all Office UI Fabric sizes and weights will look great in an Office Add-in. To fit harmoniously or avoid conflicts, consider using a subset of the Fabric type ramp. Here's a list of Fabric's base classes that we recommend for use in Office Add-ins.

| **Sample** | **Class** | **Size** | **Weight** | **Recommended Usage** |
| --- | --- | --- | --- | --- |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-hero.png) | .ms-font-xxl | 28 px | Segoe Light | * This class is larger than all other typographic elements in Office. Use it sparingly to avoid unseating visual hierarchy. * Avoid use on long strings in constrained spaces. * Provide ample whitespace around text using this class. * Commonly used for first run messages, hero elements, or other calls to action. |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-title.png) | .ms-font-xl | 21 px | Segoe Light | * This class matches the task pane title of Office applications. * Use it sparingly to avoid a flat typographic hierarchy. * Commonly used as the top-level element such as dialog box, page, or content titles. |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-subtitle.png) | .ms-font-l | 17 px | Segoe Semilight | * This class is the first stop below titles. * Commonly used as a subtitle, navigation element, or group header. |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-body.png) | .ms-font-m | 14 px | Segoe Regular | \*Commonly used as body text within add-ins. |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-caption.png) | .ms-font-xs | 11 px | Segoe Regular | \*Commonly used for secondary or tertiary text such as timestamps, by lines, captions, or field labels. |
| [Hero Text Image](https://github.com/OfficeDev/office-js-docs/blob/master/images/add-in-typeramp-annotation.png) | .ms-font-mi | 10 px | Segoe Semibold | \*The smallest step in the type ramp should be used rarely. It's available for circumstances where legibility is not required. |

Text color is not included in these base classes. Use Fabric's "neutral primary" for most text on white backgrounds.

## Color

Color is often used to emphasize brand and reinforce visual hierarchy. It helps identify an interface as well as guide customers through an experience. Inside Office, color is used for the same goals but it is applied purposefully and minimally. At no point does it overwhelm customer content. Even when each Office app is branded with its own dominant color, it is used to sparingly.

Office UI Fabric includes a set of default theme colors. When Fabric is applied to an Office Add-in as components or in layouts, the same goals apply. Color should communicate hierarchy, purposefully guiding customers to action without interfering with content. Fabric theme colors can introduce a new accent color to the overall interface. This new accent can conflict with Office app branding and interfere with hierarchy. In other words, Fabric can introduce a new accent color to the overall interface when used inside an add-in. This new accent color can distract and interfere with the overall hierarchy. Consider ways to avoid conflicts and interference. Use neutral accents or overwrite Fabric theme colors to match Office app branding or your own brand colors.

Office applications allow customers to personalize their interfaces by applying an Office UI theme. Customers can choose between four UI themes to vary styling of backgrounds and buttons in Word, PowerPoint, Excel and other apps in the Office suite. To make your add-ins feel like a natural part of Office and respond to personalization, use our Themeing APIs. For example, task pane background colors switch to a dark gray in some themes. Our theming APIs allow you to follow suit and adjust foreground text to ensure [accessibility](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-design-guidelines.md#accessibility-guidelines).

For mail and task pane add-ins, use the [Context.officeTheme](https://dev.office.com/docs/reference/shared/office.context.officetheme.htm) property to match the theme of the Office applications. This API is currently only available in Office 2016.

For PowerPoint content add-ins, see [Use Office themes in your PowerPoint add-ins](https://dev.office.com/docs/add-ins/powerpoint/use-document-themes-in-your-powerpoint-add-ins.htm).

Apply the following general guidelines for color:

* Use color sparingly to communicate hierarchy and reinforce brand.
* Overuse of a single accent color applied to both interactive and non-interactive elements can lead to confusion. For example, avoid using the same color for selected and unselected items in a navigation menu.
* Avoid unnecessary conflicts with Office branded app colors.
* Use your own brand colors to build association with your service or company.
* Ensure that all text is accessible. Be sure that there is a 4.5:1 constrast ratio between foreground text and background.
* Be aware of color blindness. Use more than just color to indicate interactivity and hierarchy.
* Refer to [icon guidelines](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/design-icons.md) to learn more about designing add-in command icons with the Office icon color pallet.

## Layout

Each HTML container embedded in Office will have a layout. These layouts are the main screens of your add-in. In them you will create experiences that enable customers to initiate actions, modify settings, view, scroll, or navigate content. Design your add-in with a consistent layouts across screens to guarantee continuity of experience. If you have an existing website that your customers are familiar with using, consider reusing layouts from your existing web pages. Adapt them to fit harmoniously within Office HTML containers.

For guidelines on layout, see [Task pane](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/task-pane-add-ins.md), [Content](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/content-add-ins.md), and [Dialog box](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/dialog-boxes.md). For more information about how to assemble Office UI Fabric components into common layouts and user experience flows, see [UX design patterns templates](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/ux-design-patterns.md).

Apply the following general guidelines for layouts:

* Avoid narrow or wide margins on your HTML containers. 20 pixels is a great default.
* Align elements intentionally. Extra indents and new points of alignment should aid visual hierarchy.
* Office interfaces are on a 4px grid. Aim to keep your padding between elements at multiples of 4.
* Overcrowding your interface can lead to confusion and inhibit ease of use with touch interactions.
* Keep layouts consistent across screens. Unexpected layout changes look like visual bugs that contribute to a lack of confidence and trust with your solution.
* Follow common layout patterns. Conventions help users understand how to use an interface.
* Avoid redundant elements like branding or commands.
* Consolidate controls and views to avoid requiring too much mouse movement.
* Create responsive experiences that adapt to HTML container widths and heights.

## Component language

Screens and layouts are composed of content and components. Components are controls that help your customers interact with elements of your software or service. Buttons, navigation, badges, alerts, and dropdowns are all examples of common components that often have consistent styles and behaviors.

Office UI Fabric renders components that look and behave like a part of Office. Take advantage of Fabric to integrate seamlessly with Office. If your add-in has its own preexisting component language, you don't need to discard it in favor of Fabric. Look for opportunities to retain it while integrating with Office. Consider ways to swap out stylistic elements, remove conflicts, or adopt styles and behaviors that remove user confusion.

Apply the following general guidelines for components:

* Don’t replicate the Office ribbon inside your add-in
* Avoid creating menus, buttons, or other components that behave differently from Office components.
* Use the [Office UI Fabric](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/office-ui-fabric.md) components the we recommend for add-ins.
* Use the [UX design patterns templates](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/ux-design-patterns.md) for common Office UI components.

## Icons

Icons are the visual representation of a behavior or concept. They are often used to add meaning to controls and commands. Visuals, either realistic or symbolic, enable the user to navigate the UI the same way signs help users navigate their environment. They should be simple and clear, and contain only the necessary details, to enable customers to quickly parse what action will occur when they choose a control.

Office ribbon interfaces have a standard visual style. If you are designing an add-in command for the Office ribbon, follow our [icon guidelines](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/design-icons). This ensures consistency and familiarity across Office apps. The guidelines will help you design a set of PNG assets for your solution that fit in as a natural part of Office.

Many HTML containers contain controls with iconography. Use Office UI Fabric’s custom font to render Office styled icons in your add-in. Fabric’s icon font contains many glyphs for common Office metaphors that you can scale, color, and style to suit your needs. If you have an existing visual language with your own set of icons, feel free to use it in your HTML canvases. Building continuity with your own brand with a standard set of icons is an important part of any design language. Be careful to avoid creating confusion for customers by conflicting with Office metaphors.

Apply the following general guidelines for icons:

* Don’t repurpose Office UI Fabric glyphs for add-in commands in the Office ribbon or contextual menus. Fabric icons are stylistically different and will not match.
* Use the Office icon language to represent behaviors or concepts.
* Reuse common Office visual metaphors such as paintbrush for format or magnifying glass for find.
* Don’t misuse metaphors for unrelated actions. Using the same visual for a different behavior or concept can cause confusion for users.

## Animation

UI elements, controls, and components often have interactive behaviors that require transitions, motion, or animation. Common characteristics of motion across UI elements define the animation aspects of a design language. Because Office is focused on productivity, the Office animation language supports the goal of helping customers get things done. It strikes a balance between performant response, reliable choreography, and detailed delight.

Office UI fabric includes an animation library to control motion in your HTML containers. Use it to fit seamlessly in Office. It will help you create experiences that are more felt than observed. The animation CSS classes provide directionality, enter/exit, and duration specifics that reinforce Office mental models and provide opportunities for customers to learn how to interact with your add-in.

If your add-in has its own animation language, use it. Look for opportunities to retain your branded animation while integrating with Office. Be careful not to interfere with or conflict with common motion patterns in Office. Avoid creating experiences that are embellishments that only distract your customers.

Apply the following general guidelines for animations:

* Animations should be felt, and experienced subconsciously, to avoid hindering task completion.
* Avoid anticipations, bounces, rubberband, or other effects that emulate natural world physics.
* Choreograph elements to reinforce hierarchy and mental models.
* Use motion to guide the user and provide compositional focus on key elements for task completion.
* Consider the origin of your triggering element. Use motion to create a link between the action and the resulting UI.
* Consider tone and purpose of your content when choosing animations. Handle critical messages differently than exploratory navigations.

# UX design pattern templates for Office Add-ins

The [UX design patterns for Office Add-ins project](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code) includes HTML, JavaScript, and CSS files that you can use to create the UX for your add-in.

Use the UX design patterns project to:

* Apply solutions to common customer scenarios.
* Apply design best practices.
* Incorporate [Office UI Fabric](https://dev.office.com/fabric#/get-started) components and styles.
* Build add-ins that visually integrate with the default Office UI.

## Using the UX design patterns

You can use the [UX designer specifications](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/Source%20Files) as a guide when you create your own UX design, or you can add the [source code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates)directly to your project.

To use the specifications to build a mock-up of your own add-in UI:

1. Download the [UX Design Patterns](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns).
2. Use the [Word](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_Word_Desktop_reference.pdf), [Excel](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_Excel_Desktop_reference.pdf), or [PowerPoint](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_PowerPoint_Desktop_reference.pdf) templates to mock up your extension points for desktop or tablet.
3. Copy and modify [ux design patterns](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/tree/master/Patterns) to meet your specific needs.
4. Refer to templates to browse an [Office UI Fabric icon master sheet](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/OfficeUIFabric_icon_mastersheet.pdf), view extension points for Word, Excel and PowerPoint (for example, [Excel iPad](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_Excel_iPad_reference.pdf) and [Excel Windows Desktop](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_Excel_Desktop_reference.pdf)), or [finalize icons for add-in commands](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Helpful%20Templates/Icon_production.pdf).

To add the source code:

1. Clone the [UX design patterns for Office Add-ins project repo](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code).
2. Copy the [assets folder](https://github.com/OfficeDev/Office-Add-in-UX-Design-Pattern-Code/tree/master/assets), and the code folder for the individual pattern you choose to your add-in project.
3. Incorporate the individual pattern into your add-in. For example:
   * Edit the source location or add-in command URL in the manifest.
   * Use the UX design pattern as a template for other pages.
   * Link to or from the UX design pattern.

## Types of UX design patterns

### Generic pages

Generic page templates can be applied to any page in your add-in and don't have a special purpose. An example of a special purpose page, would be any of the first-run patterns. The following list describes the generic pages available:

* **Landing page** - A standard add-in page, for example the page a user lands on after a first-run experience or sign-in process.
  + [Landing page specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Helpful%20Templates/AddIn_Template_Standard_Layout.pdf)
  + [Landing page code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/generic/landing-page)
* **Brand image in brand bar** - The landing page with an image in the footer that represents your brand.
  + [Brand bar specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Brand_Bar.md)
  + [Brand bar code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/generic/brand-bar)

|  |  |
| --- | --- |
| **Landing** | **Brand Bar** |
| [landing page](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/generic/landing-page) | [brand bar](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/generic/brand-bar) |

### First-Run experience

A first-run experience is the experience a user has when they open your add-in for the first time. The following first-run design pattern templates are available:

* **Steps to start** - Provides users with an ordered list of steps to perform to get started using your add-in.
  + [Steps to start specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/FirstRun_StepsToStart.md)
  + [Steps to start code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/instruction-step)
* **Value** - Communicates your add-in's value proposition.
  + [Value specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/FirstRun_ValuePlacemat.md)
  + [Value code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/value-placemat)
* **Video** - Shows users a video before they start using your add-in.
  + [Video specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/FirstRun_VideoPlacemat.md)
  + [Video code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/video-placemat)
* **Walkthrough** - Takes users through a series of features or information before they start using the add-in.
  + [Walkthrough specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/FirstRun_PagingPanel.md)
  + [Walkthrough code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/walkthrough)

The [Office Store](https://msdn.microsoft.com/en-us/library/office/jj220033.aspx) has a system that manages trial versions of an add-in, but if you want to control the UI of the trial experience for your add-in, use the following patterns:

* **Trial** - Shows users how to get started with a trial version of your add-in.
  + [Trial specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/FirstRun_TrialVersion.md)
  + [Trial code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/trial-placemat)
* **Trial feature** - Advises users that the feature they are trying to use is not available in the trial version of the add-in. Alternatively, if your add-in is free but it includes a feature that requires a subscription, consider using this pattern. You might also use this pattern to provide a downgraded experience after a trial has ended.
  + [Trial feature specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/FirstRun_TrialFeature.md)
  + [Trial feature code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/trial-placemat-feature)

**Important:** If you decide to manage your own trial, and not use the Office Store to manage the trial, make sure to include the **Additional purchase may be required** tag in the testing notes in the seller dashboard.

Consider whether showing users the first-run experience once or many times is important to your scenario. For example, if users use your add-in periodically, they might forget how to use it, and it might be helpful to see the first-run experience more than once.

|  |  |  |
| --- | --- | --- |
| **Steps to Start** | **Value** | **Video** |
| [instruction steps](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/instruction-step) | [value placemat](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/value-placemat) | [video placemat](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/video-placemat) |

|  |  |  |
| --- | --- | --- |
| **Walkthrough first page** | **Trial** | **Trial feature** |
| [walkthrough 1](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/walkthrough) | [trial placemat](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/trial-placemat) | [trial placemat feature](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/trial-placemat-feature) |

### Navigation

Users need to navigate between the different pages of your add-in. The following navigation templates show different options you can use to organize pages and commands in your add-in.

* **Back Button and Next Page** - Shows a task pane with Back and Next Page buttons. Use this pattern to ensure users follow an ordered series of steps.
  + [Back Button and Next Page specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Back_Button.md)
  + [Back Button and Next Page code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/back-button)
* **Navigation** - Shows a menu, commonly referred to as the hamburger menu, with page menu items in a task pane.
  + [Navigation specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Navigation.md)
  + [Navigation code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/navigation)
* **Navigation with commands** - Shows the hamburger menu with command (or action) buttons in a task pane. Use this pattern when you want to provide navigation and command options together.
  + [Navigation with commands specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Navigation_%26_Commands.md)
  + [Navigation with commands code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/navigation-commands)
* **Pivot** - Shows Pivot navigation inside of a task pane. Use pivot navigation to allow users to navigate between different content.
  + [Pivot specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Pivot.md)
  + [Pivot code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/pivot)
* **Tab bar** - Shows navigation using buttons with vertically stacked text and icons. Use the tab bar to provide navigation using tabs with short and descriptive titles.
  + [Tab bar specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Tab_Bar.md)
  + [Tab bar code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/tab-bar)

|  |  |  |
| --- | --- | --- |
| **Back button** | **Navigation** | **Navigation with commands** |
| [back button](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/back-button) | [navigation](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/navigation) | [navigation with commands](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/navigation-commands) |

|  |  |
| --- | --- |
| **Pivot** | **Tab bar** |
| [pivot navigation](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/pivot) | [tab bar](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/navigation/tab-bar) |

### Notifications

Your add-in can notify users of events, such as errors, or of progress in a variety of ways. The following notification templates are available:

* **Embedded dialog box** - Shows a dialog box inside the task pane that provides information and, optionally, an interactive experience, using buttons or other controls. Consider using one to prompt a user to confirm an action. Use the Embedded dialog pattern when you want to keep the user experience in the task pane.
  + [Embedded dialog box specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/Embedded_Dialog.md)
  + [Embedded dialog box code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/embedded-dialog)
* **Inline message** - Indicates error, success, or information, and can appear at a specified location in the task pane. For example, if a user enters an incorrectly formatted email address in a text box, an error message appears just below the text box.
  + [Inline message specification](https://github.com/OfficeDev/Office-Add-in-Design-Patterns/blob/master/Patterns/Notification_Inline_Message.md)
  + [Inline message code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/inline-message)
* **Message banner** - Provides information and, optionally, a simple call to action, in a banner that can be collapsed to a single line, expanded to multiple lines, or dismissed. Use message banners to report a service update or a helpful tip when the add-in starts.
  + [Message banner specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Notification_MessageBanner.md)
  + [Message banner code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/message-banner)
* **Progress bar** - Indicates the progress of a long-running, synchronous process, such as a configuration task that must complete before the user can take any further action. It is a separate interstitial page that also reinforces the add-in brand. Use a progress bar when the process can send periodic measures of how far along it is back to the add-in.
  + [Progress bar specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Notification_Progress.md)
  + [Progress bar code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/progress-bar)
* **Spinner** - Indicates that a long-running, synchronous process is underway, but provides no indication of how far along it is. It is a separate interstitial page that also reinforces the add-in brand. Use a spinner when the add-in cannot know reliably how far along a process is.
  + [Spinner specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Notification_Progress.md)
  + [Spinner code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/spinner)
* **Toast** - Provides a brief message that fades away after a few seconds. Because the user might not see the message, use toast only for nonessential information. It is a good choice for notifying users of an event in a remote system, such as the receipt of an email.
  + [Toast specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Notification_Toast.md)
  + [Toast code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/toast)

|  |  |  |
| --- | --- | --- |
| **Embedded dialog** | **Inline message** | **Message banner** |
| [embedded dialog](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/embedded-dialog) | [inline message](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/inline-message) | [message banner](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/message-banner) |

|  |  |  |
| --- | --- | --- |
| **Progress bar** | **Spinner** | **Toast** |
| [progress bar](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/progress-bar) | [spinner](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/spinner) | [toast](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/notifications/toast) |

### General components

The following are general components that you can use in your add-ins in a variety of scenarios.

#### Client dialog boxes

Client dialog boxes provide another way for users to work with your add-in outside of a task pane. The following dialog box templates are available:

* **Typeramp dialog box** - Shows a dialog box with textual content. Use the typeramp dialog to display elaborative information to users.
  + [Typeramp dialog box specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Client_Dialog.md)
  + [Typeramp dialog box code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/typeramp)
* **Alert dialog box** - Shows an alert box with important information, such as errors or notifications, to users.
  + [Alert dialog box specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Client_Dialog.md)
  + [Alert dialog box code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/alert)
* **Navigation dialog box** - Shows a dialog box with navigation. Use the navigation dialog box to allow users to navigate between different content.
  + [Navigation dialog box specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Client_Dialog.md)
  + [Navigation dialog box code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/navigation)

|  |  |
| --- | --- |
| **Typeramp dialog** | **Alert dialog** |
| [typeramp dialog](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/typeramp) | [alert dialog](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/alert) |

|  |
| --- |
| **Navigation dialog** |
| [navigation dialog](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/navigation) |

#### Feedback and Ratings

To improve the visibility and adoption of your add-in, it is helpful to provide users with the ability to rate and review your add-in in the Office Store. This pattern shows two methods for presenting feedback and ratings from within the add-in:

* User-initiated feedback - A user chooses to send feedback by using either the navigation menu (for example, using the **Send Feedback** link) or an icon on the footer.
* System-initiated feedback - After the add-in runs three times, the user is prompted to provide feedback via a Message Banner.

Either method opens a dialog box that contains the Office Store page for the add-in.

* [Feedback and ratings specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Notification_Feedback.md)
* [Feedback and ratings code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/feedback/office-store)

**Important:** This pattern currently points to the Office Store home page. Be sure to update this URL to the URL of your add-in's page in the Office Store.

|  |
| --- |
| **Feedback and ratings** |
| [Feedback and Ratings](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/feedback/office-store) |

#### Settings and Privacy

Add-ins may need a Settings page that allows users to configure settings that control the behavior of the add-in. Also, you may want to provide users with the privacy policies your add-in adheres to.

* **Settings** - Shows a task pane with configuration components that controls the behavior of the add-in. A settings page provides options for the user to choose.
  + [Settings specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Settings.md)
  + [Settings code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/settings)
* **Privacy Policy** - Shows task pane with important information about privacy policies.
  + [Privacy Policy specification](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns/blob/master/Patterns/Settings.md)
  + [Privacy Policy code](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/settings)

|  |  |
| --- | --- |
| **Settings** | **Privacy Policy** |
| [settings](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/settings) | [privacy](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/settings) |

# Usability testing for Office Add-ins

A great add-in design takes user behaviors into account. Because your own preconceptions influence your design decisions, it’s important to test designs with real users to make sure that your add-ins work well for your customers.

You can run usability tests in different ways. For many add-in developers, remote, unmoderated usability studies are the most time and cost effective. Several popular testing services make this easy; the following are some examples:

* [UserTesting.com](https://www.usertesting.com/)
* [Optimalworkshop.com](https://www.optimalworkshop.com/)
* [Userzoom.com](https://www.userzoom.com/)

These testing services help you to streamline test plan creation and remove the need to seek out participants or moderate the tests.

You need only five participants to uncover most usability issues in your design. Incorporate small tests regularly throughout your development cycle to ensure that your product is user-centered.

**Note:** We recommend that you test the usability of your add-in across multiple platforms. To [publish your add-in to the Office Store](https://msdn.microsoft.com/en-us/library/office/jj220037.aspx), it must work on all [platforms that support the methods that you define](https://dev.office.com/add-in-availability).

## 1. Sign up for a testing service

For more information, see [Selecting an Online Tool for Unmoderated Remote User Testing.](https://www.nngroup.com/articles/unmoderated-user-testing-tools/)

## 2. Develop your research questions

Research questions define the objectives of your research and guide your test plan. Your questions will help you identify participants to recruit and the tasks they will perform. Make your research questions as specific as you can. You can also seek to answer broader questions.

The following are some examples of research questions:

**Specific**

* Do users notice the "free trial" link on the landing page?
* When users insert content from the add-in to their document, do they understand where in the document it is inserted?

**Broad**

* What are the biggest pain points for the user in our add-in?
* Do users understand the meaning of the icons in our command bar, before they click on them?
* Can users easily find the settings menu?

It’s important to get data on the entire user journey – from discovering your add-in, to installing and using it. Consider research questions that address the following aspects of the add-in user experience:

* Finding your add-in in the Store
* Choosing to install your add-in
* First run experience
* Ribbon commands
* Add-in UI
* How the add-in interacts with the document space of the Office application
* How much control the user has over any content insertion flows

For more information, see [Writing Effective Questions.](http://help.usertesting.com/customer/en/portal/articles/2077663-writing-effective-questions)

## 3. Identify participants to target

Remote testing services can give you control over many characteristics of your test participants. Think carefully about what kinds of users you want to target. In your early stages of data collection, it might be better to recruit a wide variety of participants to identify more obvious usability issues. Later, you might choose to target groups like advanced Office users, particular occupations, or specific age ranges.

## 4. Create the participant screener

The screener is the set of questions and requirements you will present to prospective test participants to screen them for your test. Keep in mind that participants for services like UserTesting.com have a financial interest in qualifying for your test. It's a good idea to include trick questions in your screener if you want to exclude certain users from the test.

For example, if you want to find participants who are familiar with GitHub, to filter out users who might misrepresent themselves, include fakes in the list of possible answers.

**Which of the following source code repositories are you familiar with?**  
a. SourceShelf [Reject]  
b. CodeContainer [Reject]  
c. GitHub [Must select]  
d. BitBucket [May select]  
e. CloudForge [May select]

If you are planning to test a live build of your add-in, the following questions can screen for users who will be able to do this.

**This test requires you to have Microsoft PowerPoint 2016. Do you have PowerPoint 2016?**  
a. Yes [Must select]  
b. No [Reject]  
c. I don’t know [Reject]

**This test requires you to install a free add-in for PowerPoint 2016, and create a free account to use it. Are you willing to install an add-in and create a free account?**  
a. Yes [Must select]  
b. No [Reject]

For more information, see [Screener Questions Best Practices.](http://help.usertesting.com/customer/en/portal/articles/2077835-screener-question-best-practices)

## 5. Create tasks and questions for participants

Try to prioritize what you want tested so that you can limit the number of tasks and questions for the participant. Some services pay participants only for a set amount of time, so you want to make sure not to go over.

Try to observe participant behaviors instead of asking about them, whenever possible. If you need to ask about behaviors, ask about what participants have done in the past, rather than what they would expect to do in a situation. This tends to give more reliable results.

The main challenge in unmoderated testing is making sure your participants understand your tasks and scenarios. Your directions should be clear and concise. Inevitably, if there is potential for confusion, someone will be confused.

Don't assume that your user will be on the screen they’re supposed to be on at any given point during the test. Consider telling them what screen they need to be on to start the next task.

For more information, see [Writing Great Tasks.](http://help.usertesting.com/customer/en/portal/articles/2077824-writing-great-tasks)

## 6. Create a prototype to match the tasks and questions

You can either test your live add-in, or you can test a prototype. Keep in mind that if you want to test the live add-in, you need to screen for participants that have Office 2016, are willing to install the add-in, and are willing to sign up for an account (unless you have logon credentials to provide them.) You'll then need to make sure that they successfully install your add-in.

On average, it takes about 5 minutes to walk users through how to install an add-in. The following is an example of clear, concise installation steps. Adjust the steps based on the specifics of your test.

**Please install the (insert your add-in name here) add-in for PowerPoint 2016, using the following instructions:**

1. Open Microsoft PowerPoint 2016.
2. Select **Blank Presentation.**
3. Go to **Insert > My Add-ins.**
4. In the popup window, choose **Store.**
5. Type (Add-in name) in the search box.
6. Choose (Add-in name).
7. Take a moment to look at the Store page to familiarize yourself with the add-in.
8. Choose **Add** to install the add-in.

You can test a prototype at any level of interaction and visual fidelity. For more complex linking and interactivity, consider a prototyping tool like [InVision](https://www.invisionapp.com/). If you just want to test static screens, you can host images online and send participants the corresponding URL, or give them a link to an online PowerPoint presentation.

## 7. Run a pilot test

It can be tricky to get the prototype and your task/question list right. Users might be confused by tasks, or might get lost in your prototype. You should run a pilot test with 1-3 users to work out the inevitable issues with the test format. This will help to ensure that your questions are clear, that the prototype is set up correctly, and that you’re capturing the type of data you’re looking for.

## 8. Run the test

After you order your test, you will get email notifications when participants complete it. Unless you’ve targeted a specific group of participants, the tests are usually completed within a few hours.

## 9. Analyze results

This is the part where you try to make sense of the data you’ve collected. While watching the test videos, record notes about problems and successes the user has. Avoid trying to interpret the meaning of the data until you have viewed all the results.

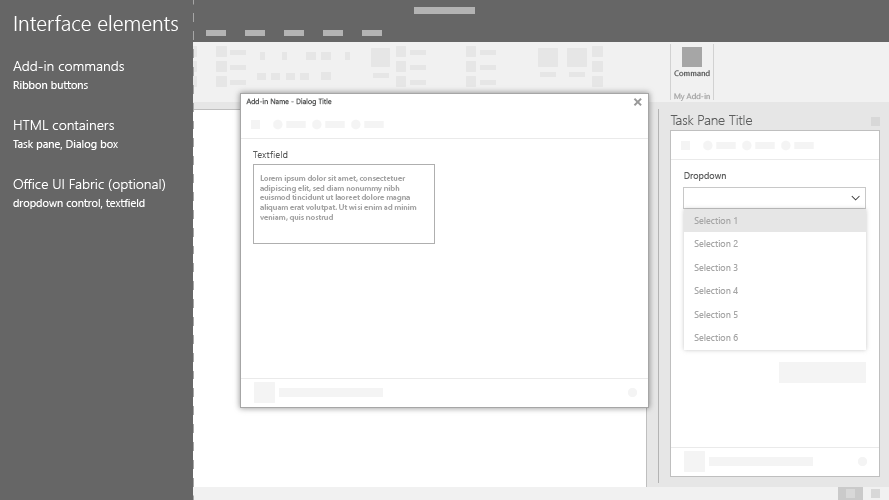
A single participant having a usability issue is not enough to warrant making a change to the design. Two or more participants encountering the same issue suggests that other users in the general population will also encounter that issue.

In general, be careful about how you use your data to draw conclusions. Don’t fall into the trap of trying to make the data fit a certain narrative; be honest about what the data actually proves, disproves, or simply fails to provide any insight about. Keep an open mind; user behavior frequently defies designer’s expectations.

**Office UI elements for Office Add-ins**

You can use several types of UI elements to extend the Office UI, including add-in commands and HTML containers. These UI elements look like a natural extension of Office and work across platforms. You can insert your custom web-based code into any of these elements.

The following image shows the types of Office UI elements that you can create.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_interfaceElements.png)

**Add-in commands**

Use [add-in commands](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-commands.md) to add entry points to your add-in to the Office ribbon. Commands start actions in your add-in either by running JavaScript code, or by launching an HTML container. You can create two types of add-in commands.

| **Command type** | **Description** |
| --- | --- |
| Ribbon buttons, menus, and tabs | Use to add custom buttons, menus (dropdowns), or tabs to the default ribbon in Office. Use Buttons and menus to trigger an action in Office. Use tabs to group and organize buttons and menus. |
| Context menus | Use to extend the default context menu. Context menus are displayed when users right-click text in an Office document or a table in Excel. |

**HTML containers**

Use HTML containers to embed HTML-based UI code within Office clients. These web pages can then reference the Office JavaScript API to interact with content in the document. You can create three types of HTML containers.

| **HTML container** | **Description** |
| --- | --- |
| [Task panes](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/task-pane-add-ins.md) | Display custom UI in the right pane of the Office document. Use task panes to allow users to interact with your add-in side-by-side with the Office document. |
| [Content add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/content-add-ins.md) | Display custom UI embedded within Office documents. Use content add-ins to allow users to interact with your add-in directly within the Office document. For example, you might want to show external content such as videos or data visualizations from other sources. |
| [Dialog boxes](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/dialog-boxes.md) | Display custom UI in a dialog box that overlays the Office document. Use a dialog box for interactions that require focus and more real estate, and do not require a side-by-side interaction with the document. |

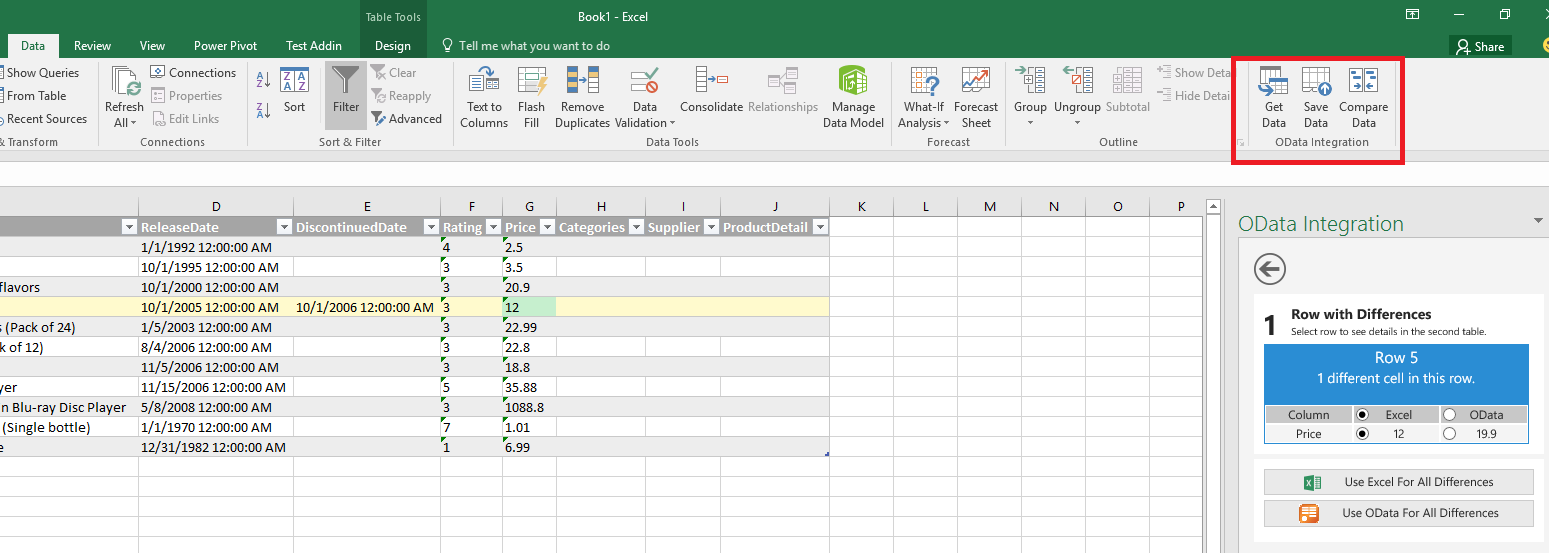
# Add-in commands for Excel, Word, and PowerPoint

Add-in commands are UI elements that extend the Office 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. 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. Add-in commands help users find and use your add-in, which can help increase your add-in's adoption and reuse, and improve customer retention.

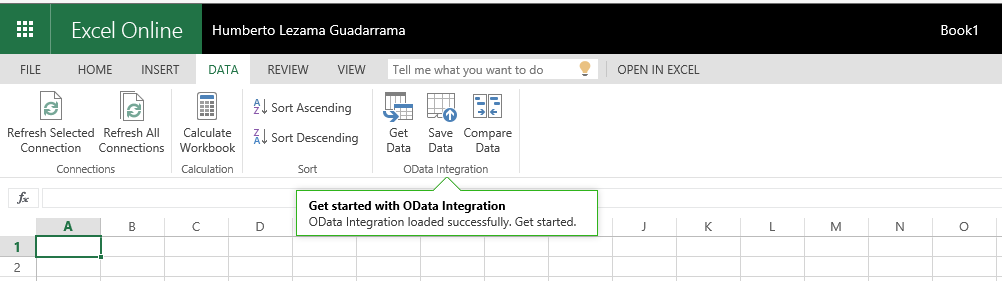
For an overview of the feature, see the video [Add-in Commands in the Office Ribbon](https://channel9.msdn.com/events/Build/2016/P551).

**Note:** SharePoint catalogs do not support add-in commands. You can deploy add-in commands via [centralized deployment](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/centralized-deployment.md) or the [Office Store](https://dev.office.com/officestore/docs/submit-to-the-office-store), or use [sideloading](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/create-a-network-shared-folder-catalog-for-task-pane-and-content-add-ins) to deploy your add-in command for testing.

**Add-in with commands running in Excel Desktop**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/addincommands1.png)

**Add-in with commands running in Excel Online**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/addincommands2.png)

## Command capabilities

The following command capabilities are currently supported.

**Note:** Content add-ins do not currently support add-in commands.

**Extension points**

* Ribbon tabs - Extend built-in tabs or create a new custom tab.
* Context menus - Extend selected context menus.

**Control types**

* Simple buttons - trigger specific actions.
* Menus - simple menu dropdown with buttons that trigger actions.

**Actions**

* ShowTaskpane - Displays one or multiple panes that load custom HTML pages inside them.
* ExecuteFunction - Loads an invisible HTML page and then execute a JavaScript function within it. To show UI within your function (such as errors, progress, or additional input) you can use the [displayDialog](http://dev.office.com/reference/add-ins/shared/officeui) API.

## Supported platforms

Add-in commands are currently supported on the following platforms:

* Office for Windows Desktop 2016 (build 16.0.6769+)
* Office for Mac (build 15.33+)
* Office Online

More platforms are coming soon.

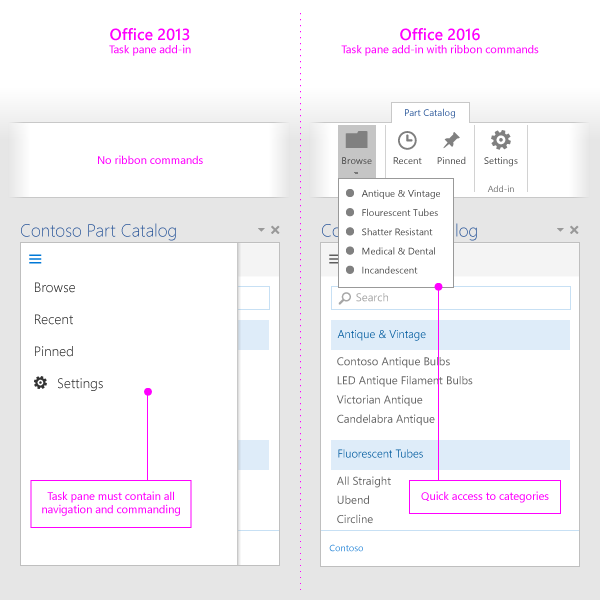
## Best practices

Apply the following best practices when you develop add-in commands:

* Use commands to represent a specific action with a clear and specific outcome for users. Do not combine multiple actions in a single button.
* Provide granular actions that make common tasks within your add-in more efficient to perform. Minimize the number of steps an action takes to complete.
* For the placement of your commands in the Office ribbon:
  + Place commands on an existing tab (Insert, Review, and so on) if the functionality provided fits there. For example, if your add-in enables users to insert media, add a group to the Insert tab. Note that not all tabs are available across all Office versions. For more information, see [Office Add-ins XML manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-manifests.md).
  + Place commands on the Home tab if the functionality doesn't fit on another tab, and you have fewer than six top-level commands. You can also add commands to the Home tab if your add-in needs to work across Office versions (such as Office Desktop and Office Online) and a tab is not available in all versions (for example, the Design tab doesn't exist in Office Online).
  + Place commands on a custom tab if you have more than six top-level commands.
  + Name your group to match the name of your add-in. If you have multiple groups, name each group based on the functionality that the commands in that group provide.
  + Do not add superfluous buttons to increase the real estate of your add-in.

**Note:** Add-ins that take up too much space might not pass [Office Store validation](https://dev.office.com/officestore/docs/validation-policies).

* For all icons, follow the [icon design guidelines](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/design-icons.md).
* Provide a version of your add-in that also works on hosts that do not support commands. A single add-in manifest can work in both command-aware (with commands) and non-command-aware (as a taskpane) hosts.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/4f90a3cc-8cc4-4879-9a03-0bb2b6079026.png)

## Next steps to get started

The best way to get started using add-in commands is to take a look at the [Office Add-in commands samples](https://github.com/OfficeDev/Office-Add-in-Commands-Samples/) on GitHub.

For more information about specifying add-in commands in your manifest, see [Define add-in commands in your manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/define-add-in-commands.md) and the [VersionOverrides](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/versionoverrides.md) reference content.

# Design icons for add-in commands

[Add-in commands](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/add-in-commands.md) add buttons, text, and icons to the Office UI. Your add-in command buttons should provide meaningful icons and labels that clearly identify the action the user is taking when they use a command. This article provides stylistic and production guidelines that help you design icons that integrate seamlessly with Office.

## Office icon design principles

The Office 2013 release of the Office desktop clients includes refreshed iconography. The overriding stylistic change is reduction. The new icons include only essential communicative elements. Non-essential elements including perspective, gradients, and light source are removed. The simplified icons support faster parsing of commands and controls. Follow this style to best fit with Office.

Office icons are based on the following design principles:

* Modern interpretation of Office icon collection
* Fresh yet familiar
* Simple, clear, and direct

The following image shows icons that apply the modern design principles.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/icons_image.PNG)

## Icon guidelines

Follow these guidelines when you create your icons:

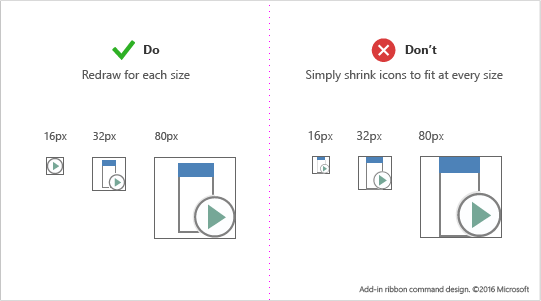
* Stick to the 1px grid and use a bitmap editing tool for best results.
* Redraw, don't resize. As you resize your icons for larger or smaller sizes, take the time to redraw cutouts, corners, and rounded edges to maximize line clarity.
* Remove artifacts that make your icon look messy.
* Don't reuse Office UI Fabric icons in the Office ribbon or contextual menu. Fabric icons are stylistically different and will not match.
* Avoid relying on your logo or brand to communicate what an add-in command does. Brand marks aren't always recognizable at smaller icon sizes and when modifiers are applied. Brand marks often conflict with Office ribbon icon styles, and can compete for user attention in a saturated environment.
* Use a white fill for accessibility. Most objects in your icons will require a white background to be legible across Office UI themes and in high-contrast modes.
* Use the PNG format with a transparent background.
* Avoid localizable content in your icons, including typographic characters, indications of paragraph rags, and question marks.
* Don't reuse visual metaphors for different commands. Using the same icon for different actions can cause confusion.
* Make your button labels clear and succinct. Use a combination of visual and textual information to convey meaning.

## Icon size recommendations and requirements

Office 2016 desktop icons are bitmap images. Different sizes will render depending on the user's DPI setting and touch mode. Include all eight supported sizes to create the best experience in all supported resolutions and contexts. The following are the supported sizes - three are required:

* 16 px (Required)
* 20 px
* 24 px
* 32 px (Required)
* 40 px
* 48 px
* 64 px (Recommended, best for Mac)
* 80 px (Required)

Make sure to redraw your icons for each size rather than shrink them to fit.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/icon_resizing.png)

## Icon anatomy and layout

Office icons are typically comprised of a base element with action and conceptual modifiers overlayed. Action modifiers represent concepts such as add, open, new, or close. Conceptual modifiers represent status, alteration, or a description of the icon.

To create commands that align with the Office UI, follow layout guidelines for the base element and modifiers. This ensures that your commands look professional and that your customers will trust your add-in. If you make exceptions to these guidelines, do so intentionally.

The following image shows the layout of base elements and modifiers in an Office icon.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/icon_layout.PNG)

* Center base elements in the pixel frame with empty padding all around.
* Place action modifiers on the top left.
* Place conceptual modifiers on the bottom right.
* Limit the number of elements in your icons. At 32px, limit the number of modifiers to a maximum of two. At 16px, limit the number of modifiers to one.

Place base elements consistently across sizes. If base elements can't be centered in the frame, align them to the top left, leaving the extra pixels on the bottom right. For best results, apply the padding guidelines listed in the following table.

| **Icon size** | **Padding around base element** |
| --- | --- |
| 16px | 0 |
| 20px | 1px |
| 24px | 1px |
| 32px | 2px |
| 40px | 2px |
| 48px | 3px |
| 64px | 5px |
| 80px | 5px |

All modifiers should have a 1px transparent cutout between each element, including the background. Elements should not directly overlap. Create whitespace between rules and edges. Modifiers can vary slightly in size, but use these dimensions as a starting point.

| **Icon size** | **Modifier size** |
| --- | --- |
| 16px | 9px |
| 20px | 10px |
| 24px | 12px |
| 32px | 14px |
| 40px | 20px |
| 48px | 22px |
| 64px | 29px |
| 80px | 38px |

## Icon colors

Office icons have a limited color palette. Use the colors listed in the following table to guarantee seamless integration with the Office UI. Apply the following guidelines to the use of color:

* Use color to communicate meaning rather than for embellishment. It should highlight or emphasize an action, status, or an element that explicitly differentiates the mark.
* If possible, use only one additional color beyond gray. Limit additional colors to two at the most.
* Colors should have a consistent appearance in all icon sizes. Office icons have slightly different color palettes for different icon sizes. 16px and smaller icons are slightly darker and more vibrant than 32px and larger icons. Without these subtle adjustments, colors appear to vary across sizes.

| **Color name** | **RGB** | **Hex** | **Color** | **Category** |
| --- | --- | --- | --- | --- |
| Text Gray (80) | 80, 80, 80 | #505050 | [Text gray 80 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textGray_80.gif) | Text |
| Text Gray (95) | 95, 95, 95 | #5F5F5F | [Text gray 95 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textGray_95.gif) | Text |
| Text Gray (105) | 105, 105, 105 | #696969 | [Text gray 105 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textGray_105.gif) | Text |
| Dark Gray 32 | 128, 128, 128 | #808080 | [Dark gray 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/darkGray_32.gif) | 32 and above |
| Medium Gray 32 | 158, 158, 158 | #9E9E9E | [Medium gray 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/mediumGray_32.gif) | 32 and above |
| Light Gray ALL | 179, 179, 179 | #B3B3B3 | [Light gray all color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/lightGray_all.gif) | All sizes |
| Dark Gray 16 | 114, 114, 114 | #727272 | [Dark gray 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/darkGray_16.gif) | 16 and below |
| Medium Gray 16 | 144, 144, 144 | #909090 | [Medium gray 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/mediumGray_16.gif) | 16 and below |
| Blue 32 | 77, 130, 184 | #4d82B8 | [Blue 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/blue_32.gif) | 32 and above |
| Blue 16 | 74, 125, 177 | #4A7DB1 | [Blue 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/blue_16.gif) | 16 and below |
| Yellow ALL | 234, 194, 130 | #EAC282 | [Yellow all color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/yellow_all.gif) | All sizes |
| Orange 32 | 231, 142, 70 | #E78E46 | [Orange 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/orange_32.gif) | 32 and above |
| Orange 16 | 227, 142, 70 | #E3751C | [Orange 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/orange_16.gif) | 16 and below |
| Pink ALL | 230, 132, 151 | #E68497 | [Pink all color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/pink_all.gif) | All sizes |
| Green 32 | 118, 167, 151 | #76A797 | [Green 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/green_32.gif) | 32 and above |
| Green 16 | 104, 164, 144 | #68A490 | [Green 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/green_16.gif) | 16 and below |
| Red 32 | 216, 99, 68 | #D86344 | [Red 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/red_32.gif) | 32 and above |
| Red 16 | 214, 85, 50 | #D65532 | [Red 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/red_16.gif) | 16 and below |
| Purple 32 | 152, 104, 185 | #9868B9 | [Purple 32 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/purple_32.gif) | 32 and above |
| Purple 16 | 137, 89, 171 | #8959AB | [Purple 16 color image](https://github.com/OfficeDev/office-js-docs/blob/master/images/purple_16.gif) | 16 and below |

## Icons in high contrast modes

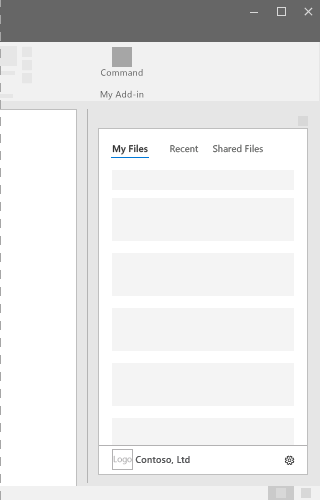
Office icons are designed to render well in high contrast modes. Foreground elements are well differentiated from backgrounds to maximize legibility and enable recoloring. In high contrast modes, Office will recolor any pixel of your icon with a red, green, or blue value less than 190 to full black. All other pixels will be white. In other words, each RGB channel is assessed where 0-189 values are black and 190-255 values are white. Other high-contrast themes recolor using the same 190 value threshold but with different rules. For example, the high-contrast white theme will recolor all pixels greater than 190 opaque but all other pixels as transparent. Apply the following guidelines to maximize legibility in high-contrast settings:

* Aim to differentiate foreground and background elements along the 190 value threshold.
* Follow Office icon visual styles.
* Use colors from our icon palette.
* Avoid the use of gradients.
* Avoid large blocks of color with similar values.

# Task panes in Office Add-ins

Task Panes are interface surfaces that typically appear on the right side of the window within Word, PowerPoint, Excel, and Outlook. Task panes give users access to interface controls that run code to modify documents or emails, or display data from a data source. Use task panes when you don't need to embed functionality directly into the document.

**Example: Task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_taskPane.png)

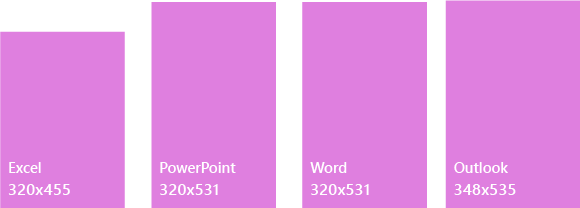
## Best practices

| **Do** | **Don't** |
| --- | --- |
| * Include the name of your add-in in the title. | * Don't append your company name to the title. |
| * Use short descriptive names in the title. | * Don't append strings such as “Add-in,” “For Word,” or “for Office” to the title of your add-in. |
| * Include some navigational or commanding element such as the CommandBar or Pivot at the top of your add-in. |  |
| * Include a branding element such as the BrandBar at the bottom of your add-in unless your add-in is to be used within Outlook. |  |

## Variants

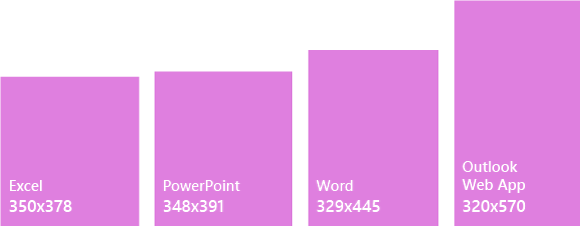
The following images show the various task pane sizes with the Office ribbon at a 1366x768 resolution. For Excel, additional vertical space is required to accommodate the formula bar.

**Office 2016 desktop task pane sizes**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/addinTaskpaneSizes_desktop.png)

* Excel - 320x455
* PowerPoint - 320x531
* Word - 320x531
* Outlook - 348x535

**Office 365 task pane sizes**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/addinTaskpaneSizes_online.png)

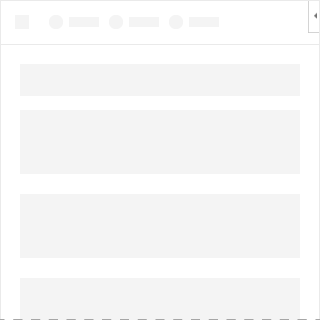
* Excel - 350x378
* PowerPoint - 348x391
* Word - 329x445
* Outlook Web App - 320x570

## Personality menu

Personality menus can obstruct navigational and commanding elements located near the top right of the add-in. The following are the current dimensions of the personality menu on Windows and Mac.

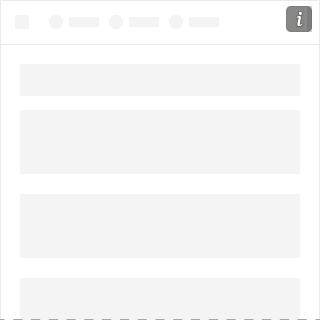
**Personality menu on Windows**

For Windows, the personality menu measures 12x32 pixels, as shown.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/personalityMenu_Win.png)

**Personality menu on Windows**

For Mac, the personality menu measures 26x26 pixels, but floats 8 pixels in from the right and 6 pixels from the top, which increases the space to 34x32 pixels, as shown.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/personalityMenu_Mac.png)

## Implementation

For a sample that implements a task pane, see [Excel Add-in JS WoodGrove Expense Trends](https://github.com/OfficeDev/Excel-Add-in-WoodGrove-Expense-Trends) on GitHub.

# Automatically open a task pane with a document

You can use add-in commands in your Office Add-in to extend the Office UI by adding buttons to the Office ribbon. When users click your command button, an action occurs, such as opening a task pane.

Some scenarios require that a task pane open automatically when a document opens, without explicit user interaction. You can use the autoopen taskpane feature, introduced in the AddInCommands 1.1 requirement set, to automatically open a task pane when your scenario requires it.

## How is the autoopen feature different from inserting a task pane?

When a user launches add-ins that don't use add-in commands - for example, add-ins that run in Office 2013 - they are inserted into the document, and persist in that document. As a result, when other users open the document, they are prompted to install the add-in, and the task pane opens. The challenge with this model is that in many cases, users don’t want the add-in to persist in the document. For example, a student who uses a dictionary add-in in a Word document might not want their classmates or teachers to be prompted to install that add-in when they open the document.

With the autoopen feature, you can explicitly define or allow the user to define whether a specific task pane add-in persists in a specific document.

## Support and availability

The autoopen feature is currently supported in the following products and platforms.

| **Products** | **Platforms** |
| --- | --- |
| * Word * Excel * PowerPoint | * Office for Windows Desktop. Build 16.0.8121.1000+ * Office for Mac. Build 15.34.17051500+ * Office Online |

## Best practices

Apply the following best practices when you use the autoopen feature:

* Use the autoopen feature when it will help make your add-in users more efficient, such as:
  + When the document needs the add-in in order to function properly. For example, a spreadsheet that includes stock values that are periodically refreshed by an add-in. The add-in should open automatically when the spreadsheet is opened to keep the values up to date.
  + When the user will most likely always use the add-in with a particular document. For example, an add-in that helps users fill in or change data in a document by pulling information from a backend system.
* Allow users to turn on or turn off the autoopen feature. Include an option in your UI for users to choose to no longer automatically open the add-in task pane.
* Use requirement set detection to determine whether the autoopen feature is available, and provide a fallback behavior if it isn’t.
* Don't use the autoopen feature to artificially increase usage of your add-in. If it doesn’t make sense for your add-in to open automatically with certain documents, this feature can annoy users.

**Note:** If Microsoft detects abuse of the autoopen feature, your add-in might be rejected from the Office Store.

* Don't use this feature to pin multiple task panes. You can only set one pane of your add-in to open automatically with a document.

## Implementation

To implement the autoopen feature:

* Specify the task pane to be opened automatically.
* Tag the document to automatically open the task pane.

**Important:** The pane that you designate to open automatically will only open if the add-in is already installed on the user's device. If the user does not have the add-in installed when they open a document, the autoopen feature will not work and the setting will be ignored. If you also require the add-in to be distributed with the document you need to set the visibility property to 1; this can only be done using OpenXML, an example is provided later in this article.

### Specify the task pane to open

To specify the task pane to open automatically, set the [TaskpaneId](https://dev.office.com/reference/add-ins/manifest/action#taskpaneid) value to **Office.AutoShowTaskpaneWithDocument**. You can only set this value on one task pane. If you set this value on multiple task panes, the first occurrence of the value will be recognized and the others will be ignored.

The following example shows the TaskPaneId value set to Office.AutoShowTaskpaneWithDocument.

<Action xsi:type="ShowTaskpane">

<TaskpaneId>Office.AutoShowTaskpaneWithDocument</TaskpaneId>

<SourceLocation resid="Contoso.Taskpane.Url" />

</Action>

### Tag the document to automatically open the task pane

You can tag the document to trigger the autoopen feature in one of two ways.

#### Tag the document on the client side

Use the Office.js [settings.set](https://dev.office.com/reference/add-ins/shared/settings.set) method to set **Office.AutoShowTaskpaneWithDocument** to **true**, as shown in the following example.

Office.context.document.settings.set("Office.AutoShowTaskpaneWithDocument", true);

Office.context.document.settings.saveAsync();

Use this method if you need to tag the document as part of your add-in interaction (for example, as soon as the user creates a binding, or chooses an option to indicate that they want the pane to open automatically).

#### Use Open XML to tag the document

You can use Open XML to create or modify a document and add the appropriate Open Office XML markup to trigger the autoopen feature. For a sample that shows you how to do this, see [Office-OOXML-EmbedAddin](https://github.com/OfficeDev/Office-OOXML-EmbedAddin).

Add two Open XML parts to the document:

* A webextension part
* A task pane part

The following example shows how to add the webextension part.

<we:webextension xmlns:we="http://schemas.microsoft.com/office/webextensions/webextension/2010/11" id="[ADD-IN ID PER MANIFEST]">

<we:reference id="[GUID or Office Store asset ID]" version="[your add-in version]" store="[Pointer to store or catalog]" storeType="[Store or catalog type]"/>

<we:alternateReferences/>

<we:properties>

<we:property name="Office.AutoShowTaskpaneWithDocument" value="true"/>

</we:properties>

<we:bindings/>

<we:snapshot xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"/>

</we:webextension>

The webextension part includes a property bag and a property named **Office.AutoShowTaskpaneWithDocument** that must be set to true.

The webextension part also includes a reference to the store or catalog with attributes for id, storeType, store, and version. Of the storeType values, only four are relevant to the autoopen feature. The values for the other three attributes depend on the value for storeType, as shown in the following table.

| **storeType value** | **id value** | **store value** | **version value** |
| --- | --- | --- | --- |
| OMEX (the Office Store) | The Office Store asset ID of the add-in.\* | The locale of the Office Store; for example, "en-us". | The version in the Office Store catalog.\* |
| FileSystem (a network share) | The GUID of the add-in in the add-in manifest. | The path of the network share; for example, "\\MyComputer\MySharedFolder". | The version in the add-in manifest. |
| EXCatalog (deployment via the Exchange server) | The GUID of the add-in in the add-in manifest. | "EXCatalog" | The version in the add-in manifest. |
| Registry (System registry) | The GUID of the add-in in the add-in manifest. | "developer" | The version in the add-in manifest. |

\* To find the asset ID and version of an add-in in the Office Store, go to the Office Store landing page for the add-in. The asset ID appears in the address bar in the browser. The version is listed in the **Details** section of the page.

For more information about the webextension markup, see [[MS-OWEXML] 2.2.5. WebExtensionReference](https://msdn.microsoft.com/en-us/library/hh695383(v=office.12).aspx).

The following example shows how to add the taskpane part.

<wetp:taskpane dockstate="right" visibility="0" width="350" row="4" xmlns:wetp="http://schemas.microsoft.com/office/webextensions/taskpanes/2010/11">

<wetp:webextensionref xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships" r:id="rId1" />

</wetp:taskpane>

Note that in this example, the visibility attribute is set to "0". This means that after the webextension and taskpane parts are added, the first time the document is opened, the user has to install the add-in from the **Add-in** button on the ribbon. Thereafter, the add-in task pane opens automatically when the file is opened. Also, when you set visibility to "0", you can use Office.js to enable users to turn on or turn off the autoopen feature. Specifically, your script sets the **Office.AutoShowTaskpaneWithDocument** document setting to true or false. (For details, see [Tag the document on the client side](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/automatically-open-a-task-pane-with-a-document.md#tag-the-document-on-the-client-side).)

If visibility is set to "1", the task pane opens automatically the first time the document is opened. The user is prompted to trust the add-in, and when trust is granted, the add-in opens. Thereafter, the add-in task pane opens automatically when the file is opened. However, when visibility is set to "1", you can't use Office.js to enable users to turn on or turn off the autoopen feature.

Setting visibility to "1" is a good choice when the add-in and the template or content of the document are so closely integrated that the user would not opt out of the autoopen feature.

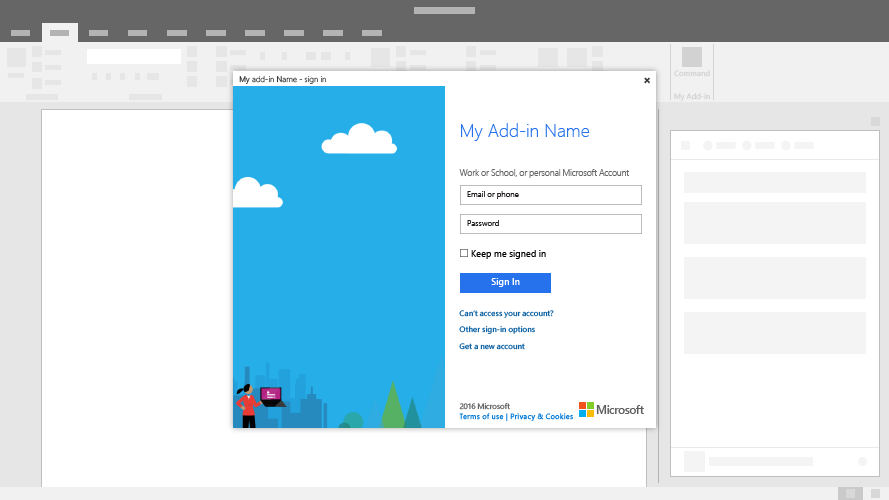
**Note:** If you want to distribute your add-in with the document, so that users are prompted to install it, you must set the visibility property to 1. You can only do this via Open XML.

An easy way to write the XML is to first run your add-in and [tag the document on the client side](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/automatically-open-a-task-pane-with-a-document.md#tag-the-document-on-the-client-side) to write the value, and then save the document and inspect the XML that is generated. Office will detect and provide the appropriate attribute values. You can also use the [Open XML SDK 2.5 Productivity Tool](https://www.microsoft.com/en-us/download/details.aspx?id=30425) tool to generate C# code to programmatically add the markup based on the XML you generate.

# Dialog boxes in Office Add-ins

Dialog boxes are surfaces that float above the active Office application window. You can use dialog boxes to provide additional screen space for tasks such as sign-in pages that can't be opened directly in a task pane or requests to confirm an action taken by a user, or to show videos that might be too small if confined to a task pane.

**Example: Dialog box**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_dialog.png)

### Best practices

| **Do** | **Don't** |
| --- | --- |
| * Include a descriptive title that includes your add-in name along with the current task. | * Don't append your company name to the title. |
|  | * Don't open a dialog box unless the scenario requires it. |

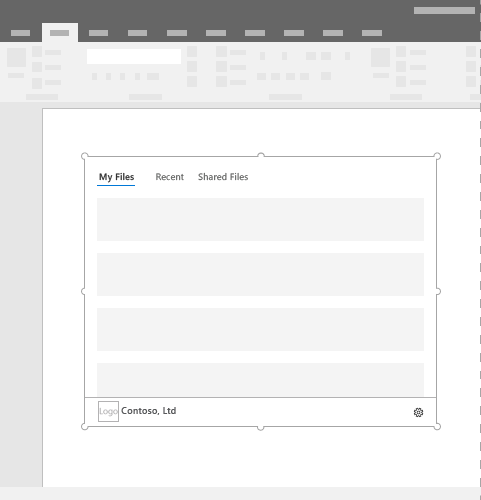
## Implementation

For a sample that implements a dialog box, see [Office Add-in Dialog API Example](https://github.com/OfficeDev/Office-Add-in-Dialog-API-Simple-Example) in GitHub.

# Content Office Add-ins

Content add-ins are surfaces that can be embedded directly into Word, Excel, or PowerPoint documents. Content add-ins give users access to interface controls that run code to modify documents or display data from a data source. Use content add-ins when you want to embed functionality directly into the document.

**Example: Content add-in**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_content.png)

## Best practices

* Include some navigational or commanding element such as the CommandBar or Pivot at the top of your add-in.
* Include a branding element such as the BrandBar at the bottom of your add-in (applies to Word, Excel, and PowerPoint add-ins only).

## Variants

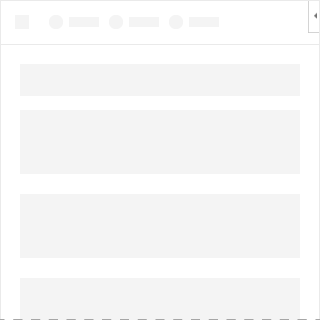
Content add-in sizes for Word, Excel, and PowerPoint in Office 2016 desktop and Office 365 are user specified.

## Personality menu

Personality menus can obstruct navigational and commanding elements located near the top right of the add-in. The following are the current dimensions of the personality menu on Windows and Mac.

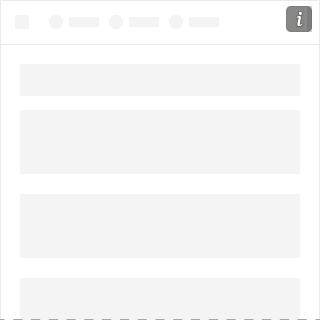
**Personality menu on Windows**

For Windows, the personality menu measures 12x32 pixels, as shown.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/personalityMenu_Win.png)

**Personality menu on Mac**

For Mac, the personality menu measures 26x26 pixels, but floats 8 pixels in from the right and 6 pixels from the top, which increases the occupied space to 34x32 pixels, as shown.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/personalityMenu_Mac.png)

## Implementation

For a sample that implements a content add-in, see [Excel Content Add-in Humongous Insurance](https://github.com/OfficeDev/Excel-Content-Add-in-Humongous-Insurance/compare/master) in GitHub.

# Data visualization style guidelines for Office Add-ins

Good data visualizations help users find insights in their data. They can use those insights to tell stories that inform and persuade. This article provides guidelines to help you design effective data visualizations in your add-ins for Excel and other Office apps.

We recommend that you use [Office UI Fabric](http://dev.office.com/fabric) to create the chrome for your data visualizations. Office UI Fabric includes styles and components that integrate seamlessly with the Office look and feel.

## Data visualization elements

Data visualizations share a general framework and common visual and interactive elements, including titles, labels, and data plots, as shown in the following figures.

### Chart titles

Follow these guidelines for chart titles:

* Make your chart titles easily readable. Position them to create a clear visual hierarchy in relation to the rest of the chart.
* In general, use sentence capitalization (capitalize the first word). To create contrast or to reinforce hierarchies, you can use all caps, but all caps should be used sparingly.
* Incorporate the [Office UI Fabric type ramp](http://dev.office.com/fabric#/styles/typography) to make your charts consistent with the Office UI, which uses Segoe. You can also use a different typeface to differentiate chart content from the UI.
* Use sans-serif typefaces with large counters.

The following examples show serif and sans-serif typefaces used in chart titles. Notice how the scale contrast and effective use of white space create a strong visual hierarchy.

### Axis labels

Make your axis labels dark enough to read clearly, with adequate contrast ratios between the text and background colors. Make sure that they are not so dark that they compete with data ink.

Light grays are most effective for axis labels. If you’re using Fabric, see the [Neutral Colors palette](http://dev.office.com/fabric#/styles/colors).

### Data ink

The pixels that represent the actual data in a chart are referred to as data ink. This should be the central focus of the visualization. Avoid the use of drop shadows, heavy outlines, or unnecessary design elements that distort or compete with the data. Use gradients only when data values are tied to color values. Avoid three-dimensional charts unless a measurable, objective value is bound to a third dimension.

### Color

Choose colors that follow operating system or application themes rather than hardcoded colors. At the same time, make sure that the colors you apply do not distort the data. Misuse of color in data visualizations can result in data distortion and incorrect reading of information.

For best practices for use of color in data visualizations, see the following:

* [Why rainbow colors aren't the best option for data visualizations](http://www.poynter.org/2013/why-rainbow-colors-arent-always-the-best-options-for-data-visualizations/224413/)
* [Color Brewer 2.0: Color Advice for Cartography](http://colorbrewer2.org/)
* [I Want Hue](http://tools.medialab.sciences-po.fr/iwanthue/)

### Gridlines

Gridlines are often necessary for accurately reading a chart, but should be presented as a secondary visual element, enhancing the data ink, not competing with it. Make static gridlines thin and light, unless they are designed specifically for high contrast. You can also use interaction to create dynamic, just-in-time gridlines that appear in context when a user interacts with a chart.

Light grays are most effective for gridlines. If you’re using Fabric, see the [Neutral Colors palette](http://dev.office.com/fabric#/styles/colors).

The following image shows a data visualization with gridlines.

### Legends

Add legends if necessary to:

* Distinguish between series
* Present scale or value changes

Make sure that your legends enhance the data ink and do not compete with it. Place legends:

* Flush left above the plot area by default, if all legend items fit above the chart.
* On the upper right side of the plot area, if all legend items do not fit above the chart, and make it scrollable, if necessary.

To optimize for readability and accessibility, map legend markers to the relevant chart shape. For example, use circle legend markers for scatter plot and bubble chart legends. Use line segment legend markers for line charts.

### Data labels and tooltips

Ensure that data labels and tooltips have adequate white space and type variation. Use algorithms to minimize occlusion and collision. For example, a tooltip might surface to the right of a data point by default, but surface to the left if right edges are detected.

## Design Principles

The Office Design team created the following set of design principles, which we use when designing new data visualizations for the Office product suite.

## Visual design principles

* Visualizations should honor and enhance the data, making it easy to understand. Highlight the data, adding supporting elements only as needed to provide context. Avoid unnecessary embellishments (drop shadows, outlines, etc), chart junk, or data distortion.
* Visualizations should encourage exploration by providing rich visual feedback. Use well-established interaction patterns, interface controls, and clear system feedback.
* Embody time-honored design principles. Use established typographic and visual communication design principles to enhance form, readability, and meaning.

## Interaction design principles

* Design to allow for exploration.
* Allow for direct interactions with objects that reveal new insights (sorting via drag, for example).
* Use simple, direct, familiar interaction models.

For more information about how to design user-friendly interactive data visualizations, see [UI Tenets and Traps](http://uitraps.com/).

## Motion design principles

Motion follows stimulus. Visual elements should move in the same direction at the same rate. This applies to:

* Chart creation
* Transition from one chart type to another chart type
* Filtering
* Sorting
* Adding or subtracting data
* Brushing or slicing data
* Resizing a chart

Create a perception of causality. When staging animations:

* Stage one thing at a time.
* Stage changes to axes before changes to data ink.
* Stage and animate objects as a group if they are moving at the same speed in the same direction.
* Stage data elements in groups of no more than 4-5 objects. Viewers have difficulty tracking more than 4-5 objects independently.

Motion adds meaning.

* Animations increase user comprehension of changes to the data, provide context, and act as a non-verbal annotation layer.
* Motion should occur in a meaningful coordinate space of the visualization.
* Tailor the animation to the visual.
* Avoid gratuitous animations.

Motion follows data.

* Preserve data mappings. If an area is tied to a measure, maintain that area in transition.
* Maintain a consistent animation design language. Where possible, map data visualization animation to existing Office motion design language. Use similar animations for similar chart types.

## Accessibility in data visualizations

* Do not use color as the only way to communicate information. People who are color blind will not be able to interpret the results. Use shape, size and texture in addition to color when possible to communicate information.
* Make all interactive elements, such as push buttons or pick lists, accessible from a keyboard.
* Send accessibility events to screen readers to announce focus changes, tooltips, and so on.

# Office UI Fabric in Office Add-ins

Office UI Fabric is a JavaScript front-end framework for building user experiences for Office and Office 365. Fabric provides visuals-focused components that you can extend, rework, and use in your Office Add-in. Because Fabric uses the Office Design Language, Fabric's UX components look like a natural extension of Office.

If you are building an add-in, we encourage you to use Office UI Fabric to create your user experience. Using Office UI Fabric is optional.

The following sections explain how to get started using Fabric to meet your requirements.

## Use Fabric Core: icons, fonts, colors

Fabric Core contains basic elements of the design language such as icons, colors, type, and grid. Fabric core is framework independent. Both Fabric React and Fabric JS use Fabric Core.

To get started using Fabric Core:

1. Add the CDN reference to the HTML on your page.

<link rel="stylesheet" href="https://static2.sharepointonline.com/files/fabric/office-ui-fabric-js/1.4.0/css/fabric.min.css">

1. Use Fabric icons and fonts.

To use a Fabric icon, include the "i" element on your page, and then reference the appropriate classes. You can control the size of the icon by changing the font size. For example, the following code shows how to make an extra-large table icon that uses the themePrimary (#0078d7) color.

<i class="ms-Icon ms-font-xl ms-Icon--Table ms-fontColor-themePrimary"></i>

To find more icons that are available in Office UI Fabric, use the search feature on the [Icons](https://dev.office.com/fabric#/styles/icons) page. When you find an icon to use in your add-in, be sure to prefix the icon name with ms-Icon--.

For information about font sizes and colors that are available in Office UI Fabric, see [Typography](https://dev.office.com/fabric#/styles/typography) and [Colors](https://dev.office.com/fabric#/styles/colors).

## Use Fabric Components

Fabric provides a variety of UX components that you can use to build your add-in, including the following types of components:

* Input components - for example, Button, Checkbox, and Toggle
* Navigation components - for example, Pivot Breadcrumb
* Notification components - for example, MessageBar and Callout

Not all Fabric components are recommended for use in add-ins. We provide guidance for how you can use the recommended components in this section. For example, for guidance for using a Fabric button in your add-in, see [Button](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/button.md).

You can use different JavaScript frameworks, such as Angular or React, to build your add-in. To get started using Fabric components with your framework, see the following resources.

| **Framework** | **Example** |
| --- | --- |
| **JavaScript only** (no framework) | [Using Office UI Fabric JS in Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/using-office-ui-fabric-js.md). |
| **React** | [Using Office UI Fabric React in Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/using-office-ui-fabric-react.md) |
| **Angular** | See [ngOfficeUIFabric](http://ngofficeuifabric.com/), which is a community project with Angular 1.5 directives, and [Consider wrapping Fabric components with Angular 2 components](https://dev.office.com/docs/add-ins/develop/add-ins-with-angular2#consider-wrapping-fabric-components-with-angular-2-components) |

# Use Office UI Fabric JS in Office Add-ins

Office UI Fabric is a JavaScript front-end framework for building user experiences for Office and Office 365. If you build an add-in using JavaScript only, without using a framework like Angular or React, consider using Fabric JS to create your user experience. For more information, see [Office UI Fabric JS](https://dev.office.com/fabric-js).

This article walks you through the basics of using Fabric JS.

## Add the Fabric CDN references

To reference Fabric from the CDN, add the following HTML code to your page.

<link rel="stylesheet" href="https://static2.sharepointonline.com/files/fabric/office-ui-fabric-js/1.4.0/css/fabric.min.css">

<link rel="stylesheet" href="https://static2.sharepointonline.com/files/fabric/office-ui-fabric-js/1.4.0/css/fabric.components.min.css">

<script src="https://static2.sharepointonline.com/files/fabric/office-ui-fabric-js/1.4.0/js/fabric.min.js"></script>

## Use Fabric JS UX components

Fabric JS provides several UX components, like buttons or checkboxes, that you can use in your add-in. The following is a list of the Fabric JS UX components that we recommend for use in an add-in. To use one of the Fabric components in your add-in, follow the link to the Fabric documentation, and then follow the instructions in **Using this component**.

* [Breadcrumb](https://dev.office.com/fabric-js/Components/Breadcrumb/Breadcrumb.html)
* [Button](https://dev.office.com/fabric-js/Components/Button/Button.html) (Consider using the small button variant in your add-in. Add 16px of padding to small buttons to ensure a 40px minimum touch target on touch devices.)
* [Checkbox](https://dev.office.com/fabric-js/Components/CheckBox/CheckBox.html)
* [ChoiceFieldGroup](https://dev.office.com/fabric-js/Components/ChoiceFieldGroup/ChoiceFieldGroup.html)
* [Date Picker](https://dev.office.com/fabric-js/Components/DatePicker/DatePicker.html) (For an example that shows how to implement the Date Picker in an add-in, see the [Excel Sales Tracker](https://github.com/OfficeDev/Excel-Add-in-JavaScript-SalesTracker) code sample.)
* [Dropdown](https://dev.office.com/fabric-js/Components/Dropdown/Dropdown.html)
* [Label](https://dev.office.com/fabric-js/Components/Label/Label.html)
* [Link](https://dev.office.com/fabric-js/Components/Link/Link.html)
* [List](https://dev.office.com/fabric-js/Components/List/List.html) (Consider changing the component's default styles in the CSS.)
* [MessageBanner](https://dev.office.com/fabric-js/Components/MessageBanner/MessageBanner.html)
* [MessageBar](https://dev.office.com/fabric-js/Components/MessageBar/MessageBar.html)
* [Overlay](https://dev.office.com/fabric-js/Components/Overlay/Overlay.html)
* [Panel](https://dev.office.com/fabric-js/Components/Panel/Panel.html)
* [Pivot](https://dev.office.com/fabric-js/Components/Pivot/Pivot.html)
* [ProgressIndicator](https://dev.office.com/fabric-js/Components/ProgressIndicator/ProgressIndicator.html)
* [Searchbox](https://dev.office.com/fabric-js/Components/SearchBox/SearchBox.html)
* [Spinner](https://dev.office.com/fabric-js/Components/Spinner/Spinner.html)
* [Table](https://dev.office.com/fabric-js/Components/Table/Table.html)
* [TextField](https://dev.office.com/fabric-js/Components/TextField/TextField.html)
* [Toggle](https://dev.office.com/fabric-js/Components/Toggle/Toggle.html)

## Updating your add-in to use Fabric JS

If you've been using a previous version of Office UI Fabric and you'd like to move to Fabric JS, make sure that you learn about, incorporate, and test the new components in your add-in. Keep the following points in mind to help you plan for your updates:

* Component initialization is simpler using Fabric JS. For previous versions of Fabric, you include the Fabric component's JavaScript file in your add-in project, included a <Script> reference to that file, and then initialize the component. In Fabric JS, you no longer need to include the Fabric component's JavaScript file and the associated <Script> reference. All you need to do is initialize the Fabric component.
* Several components now provide functions that control the behavior of the UX component. For example, the checkbox control has a toggle function that switches between the checked and unchecked states.
* Some icon class names and styles have been updated.
* The most notable change is the use of the <label> element in many components. The <label> element controls the style of the component. You might have to update your UX code to use the <label> element. For example, changing the value of the <input> element's checked attribute on a Fabric JS checkbox has no effect on the checkbox. Instead, you use the check, unCheck, or toggle functions.

## Next steps

If you're looking for an end-to-end code sample that shows you how to use Fabric JS, we've got you covered. See the following resource:

* [Excel Sales Tracker](https://github.com/OfficeDev/Excel-Add-in-JavaScript-SalesTracker)

## Related resources

If you're looking for code samples or documentation about a previous release of Fabric, see the following:

* [UX design patterns (uses Fabric 2.6.1)](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code)
* [Office Add-in Fabric UI sample (uses Fabric 1.0)](https://github.com/OfficeDev/Office-Add-in-Fabric-UI-Sample)
* [Using Fabric 2.6.1 in an Office Add-in](https://dev.office.com/docs/add-ins/design/ui-elements/using-office-ui-fabric)

# Use Office UI Fabric React in Office Add-ins

Office UI Fabric is a JavaScript front-end framework for building user experiences for Office and Office 365. If you build your add-in using React, consider using Fabric React to create your user experience. Fabric provides several React-based UX components, like buttons or checkboxes, that you can use in your add-in.

To get started using Fabric React's components in your add-in, perform the following steps.

**Note:** If you follow the steps in this section, Fabric Core is also available in your add-in.

## Step 1 - Create your project with the Yeoman generator for Office

To create an add-in that uses Fabric React, we recommend that you use the Yeoman generator for Office. The Yeoman generator for Office provides the project scaffolding and build management needed to develop an Office add-in.

To create your project, perform the following steps from [Create an Office Add-in using any editor](https://dev.office.com/docs/add-ins/get-started/create-an-office-add-in-using-any-editor) using **Windows PowerShell**(not the command prompt):

1. Install the prerequisites.
2. Run yo office to create the project files for your add-in.
3. When prompted to select an Office client application, choose **Word**.
4. Ensure you are in the directory with the project files, and then run npm start. A browser window showing a spinner opens automatically.
5. Sideload your manifest to view the full UI of the add-in.

## Step 2 - Add a Fabric React component

Next, add Fabric React components to your add-in. Create a new React component, called ButtonPrimaryExample, that consists of a Label and PrimaryButton from Fabric React. To create ButtonPrimaryExample:

1. Open the project folder created by the Yeoman generator, and go to **src\components**.
2. Create **button.tsx**.
3. In **button.tsx**, enter the following code to create the ButtonPrimaryExample component.

import \* as React from 'react';

import { PrimaryButton, IButtonProps } from 'office-ui-fabric-react/lib/Button';

import { Label } from 'office-ui-fabric-react/lib/Label';

export class ButtonPrimaryExample extends React.Component<IButtonProps, {}> {

public constructor() {

super();

}

insertText = async () => {

// In the click event, write text to the document.

await Word.run(async (context) => {

var body = context.document.body;

body.insertParagraph('Hello Office UI Fabric React!', Word.InsertLocation.end);

await context.sync();

});

}

public render() {

let { disabled } = this.props;

return (

<div className='ms-BasicButtonsExample'>

<Label>Click the button to insert text.</Label>

<PrimaryButton

data-automation-id='test'

disabled={ disabled }

text='Insert text...'

onClick={ this.insertText }

/>

</div>

);

}

}

This code does the following:

* References the React library using import \* as React from 'react';.
* References the Fabric components (PrimaryButton, IButtonProps, Label) that are used to create ButtonPrimaryExample.
* Declares and make public the new ButtonPrimaryExample component using export class ButtonPrimaryExample extends React.Component.
* Declares the insertText function to handle the onclick event.
* Defines the UI of the React component in the render function. Render defines the structure of the component. Within render, you wire up the onclick event using this.insertText.

## Step 3 - Add the React component to your add-in

Add ButtonPrimaryExample to your add-in by opening **src\components\app.tsx** and doing the following:

* Add the following import statement to reference ButtonPrimaryExample from **button.tsx** created in step 2 (no file extension is needed). import {ButtonPrimaryExample} from './button';
* Replace the default render() function with the following code that uses <ButtonPrimaryExample />.

render() {

return (

<div className='ms-welcome'>

<Header logo='assets/logo-filled.png' title={this.props.title} message='Welcome' />

<HeroList message='Discover what this add-in can do for you today!' items={this.state.listItems}>

<ButtonPrimaryExample />

</HeroList>

</div>

);

};

Save your changes. All open browser instances, including the add-in, update automatically and show the ButtonPrimaryExample React component. Notice that the default text and button is replaced with the text and primary button defined in ButtonPrimaryExample.

### Recommended components

The following is a list of the Fabric React UX components that we recommend for use in an add-in.

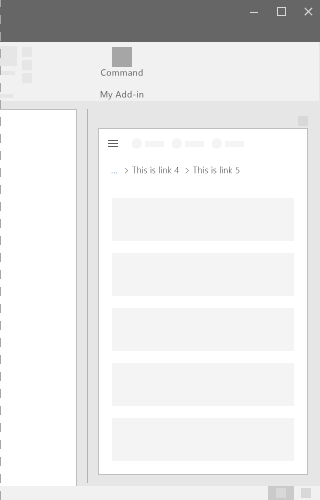
**Note:** We will add additional components over time.

* [Breadcrumb](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/breadcrumb.md)
* [Button](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/button.md)
* [Checkbox](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/checkbox.md)
* [ChoiceGroup](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/choicegroup.md)
* [Dropdown](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/dropdown.md)
* [Label](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/label.md)
* [List](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/list.md)
* [Pivot](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/pivot.md)
* [TextField](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/textfield.md)
* [Toggle](https://github.com/OfficeDev/office-js-docs/blob/master/docs/design/toggle.md)

**Breadcrumb component in Office UI Fabric**

In Office Add-ins, use breadcrumbs for navigation. They show the current page’s location in a hierarchy, and help users understand where they are in relation to the rest of the hierarchy. Additionally, Breadcrumbs provide one-click navigation to higher levels in the hierarchy.

**Example: Breadcrumb in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_breadcrumb.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Place Breadcrumbs at the top of a layout in an add-in, above a list of items, or above the main content of a layout. | Don’t use Breadcrumbs as the primary way to navigate to other pages. |
| [Do Breadcrumb image](https://github.com/OfficeDev/office-js-docs/blob/master/images/breadcrumbDo.png) | [Don't Breadcrumb image](https://github.com/OfficeDev/office-js-docs/blob/master/images/breadcrumbDont.png) |

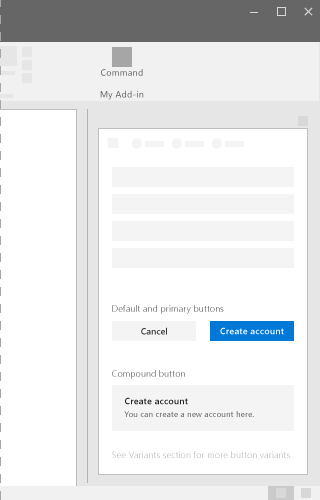
**Implementation**

For details, see [Breadcrumb](https://dev.office.com/fabric#/components/breadcrumb) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Button component in Office UI Fabric**

Use buttons in your Office Add-in to enable users to commit changes or complete steps in a task. Make sure that the text of the button communicates the intent of the interaction. Place buttons at the bottom of the UI container of a task pane, dialog, or content pane. For example, use buttons to allow users to submit a form, close a dialog, or to move to the next page.

**Example: Buttons in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_button.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Default buttons should always perform safe operations in add-ins. | Don’t place the default focus on a button that destroys data. Instead, place the focus on the button that performs the safe operation or cancels the action. |
| [Do Button example](https://github.com/OfficeDev/office-js-docs/blob/master/images/buttonDo.png) | [Don't Button example](https://github.com/OfficeDev/office-js-docs/blob/master/images/buttonDont.png) |
| Use only a single line of text in the label of the button. Keep text to a minimum. | Don’t put anything other than text in a button. |
| Make sure the label conveys a clear purpose of the button to the user. Use concise, specific, self-explanatory labels. Consider using a single word only. | Don’t use buttons for navigation, except for “Back” and “Next” steps. For navigation, consider using a link. |
| Expose only one or two buttons (actions) to the user. For example, “Accept” and “Cancel”. If you need to expose more actions, consider using checkboxes or radio buttons for users to select actions, and provide a single button to start the selected actions. |  |
| Style “Submit”, “OK”, and “Apply” buttons as primary buttons. When “Reset” or “Cancel” buttons appear alongside one of these, style them as default buttons. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Primary Button** | Primary buttons inherit the theme color at rest state. Use primary buttons to highlight the main call to action. | [Primary Button image](https://github.com/OfficeDev/office-js-docs/blob/master/images/button_primary.png) |
| **Default Button** | Default buttons should always perform safe operations and should never delete. | [Default Button image](https://github.com/OfficeDev/office-js-docs/blob/master/images/button_default.png) |
| **Compound Button** | Use compound buttons to cause actions that complete a task, or cause a transitional task. | [Compound Button image](https://github.com/OfficeDev/office-js-docs/blob/master/images/button_compound.png) |

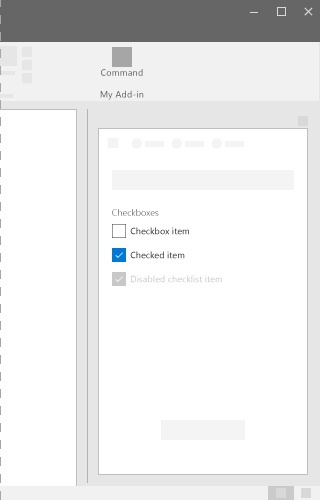
**Implementation**

For details, see [Button](https://dev.office.com/fabric#/components/button) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Checkbox component in Office UI Fabric**

A checkbox is a UI element that allows users to check or uncheck options in add-ins. Use checkboxes to allow users to select between options. Additionally, a checkbox may be paired with a related control. When the checkbox is checked or unchecked, the behavior of the related control changes. For example, the related control may toggle between the visible or hidden states.

**Example: Checkbox in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_checkbox.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Use checkboxes to indicate status. | Don’t use checkboxes to show/indicate an action. |
| [Do checkbox example](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkboxDo.png) | [Don't checkbox example](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkboxDont.png) |
| Use multiple checkboxes when users can select multiple options, and the options are not mutually exclusive. | Don’t use a checkbox when users can choose only one option. Use radio buttons when selecting only one option is required. |
| Allow users to choose any combination of options when several checkboxes are grouped together. | Don't put two groups of checkboxes next to each other. Separate the two groups with labels. |
| Use a single checkbox for a secondary setting. For example, the “Remember me?” checkbox is a secondary setting used in a login scenario. | Don’t use checkboxes to turn settings on and off. To change between an on and off state, use a Toggle. |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Uncontrolled checkbox** | Use as the default checkbox state. | [Uncontrolled Checkbox image](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkbox_unchecked.png) |
| **Uncontrolled checkbox with default checked true** | Use when the checkbox instance maintains its own state | [Uncontrolled Checkbox with default checked true image](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkbox_checked.png) |
| **Disabled uncontrolled checkbox with default checked true** | Disabled state of the checkbox. | [Disabled uncontrolled Checkbox with default checked true image](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkbox_disabled.png) |
| **Controlled checkbox** | The checked state of this checkbox is decided at another location in your UI. In this scenario, the correct value is passed to the checkbox by an onChange event and re-rendering the UI. | [Controlled Checkbox image](https://github.com/OfficeDev/office-js-docs/blob/master/images/checkbox_unchecked.png) |

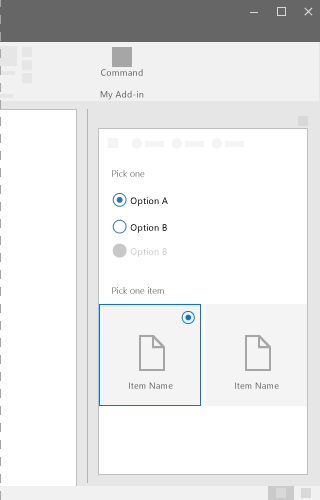
**Implementation**

For details, see [Checkbox](https://dev.office.com/fabric#/components/checkbox) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**ChoiceGroup component in Office UI Fabric**

The ChoiceGroup component, also known as a radio button, presents users with two or more mutually exclusive options. Users can select only one ChoiceGroup button in a group. Each option is represented by one ChoiceGroup button.

**Example: ChoiceGroup in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_choicegroup.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Keep ChoiceGroup options at the same level. | Don't use nested ChoiceGroups or CheckBoxes. |
| [Do ChoiceGroup example](https://github.com/OfficeDev/office-js-docs/blob/master/images/choiceDo.png) | [Don't ChoiceGroup example](https://github.com/OfficeDev/office-js-docs/blob/master/images/choiceDont.png) |
| Use ChoiceGroups with 2-7 options, ensuring there is enough screen space to show all options. Otherwise, use a Checkbox or Dropdown. | Don't use when the options are numbers with a fixed step. For example 10, 20, 30, and so on. Instead, use a slider component. |
| If users may not choose any of the options, consider including an option like "None" or "Does not apply". | Don’t use two ChoiceGroup buttons for a single binary choice. |
| If possible, align ChoiceGroup buttons vertically instead of horizontally. Horizontal alignment is harder to read and localize. |  |
| List options in logical order. For example, most likely option to be selected to least, simplest operation to most complex, or least risk to high risk. Alphabetical ordering is not recommended because it is language dependent. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **ChoiceGroups** | Use when imagery is not necessary for making a selection. | [ChoiceGroup variant image](https://github.com/OfficeDev/office-js-docs/blob/master/images/radio.png) |
| **ChoiceGroups using images** | Use when imagery is necessary for making a selection. | [ChoiceGroup variant with image](https://github.com/OfficeDev/office-js-docs/blob/master/images/radioImage.png) |

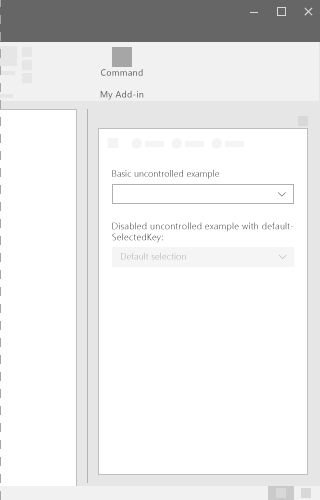
**Implementation**

For details, see [ChoiceGroup](https://dev.office.com/fabric#/components/choicegroup) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Dropdown component in Office UI Fabric**

A Dropdown is a list of options which is shown by clicking a dropdown button. Use Dropdowns to simplify the UI design, and when users should make a choice within the UI. When the list collapses, the selected items is visible. To change the selected item, users open the list, and select a new value.

**Example: Dropdown in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_dropdown.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Use a Dropdown when the default selected option is more likely to be selected than other options. By contrast, ChoiceGroup or radio buttons show all choices thereby putting equal emphasis on all options. | Don't use Dropdowns where all options are equally likely to be selected. |
| Use a Dropdown when there are multiple choices that can be collapsed into one field. Also, use Dropdowns for long lists of items, or when screen space is constrained. | Don’t use Dropdowns if there are fewer than two choices. Instead, use a checkbox. |
| Use shortened statements or words in Dropdowns. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Basic uncontrolled Dropdown** | Use when many options are available for selection. | [Basic uncontrolled Dropdown image](https://github.com/OfficeDev/office-js-docs/blob/master/images/dropdownUncontrolled.png) |
| **Disabled uncontrolled Dropdown with defaultSelectedKey** | Disabled state of the Dropdown. | [Disabled uncontrolled Dropdown with defaultSelectedKey image](https://github.com/OfficeDev/office-js-docs/blob/master/images/dropdownDisabled.png) |
| **Controlled Dropdown** | Use when the default selected item is influenced by another location in your UI, and the selected item in the DropDown must be maintained. | [Controlled Dropdown image](https://github.com/OfficeDev/office-js-docs/blob/master/images/dropdownControlled.png) |

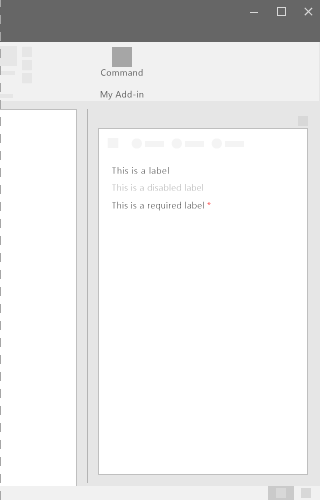
**Implementation**

For details, see [Dropdown](https://dev.office.com/fabric#/components/dropdown) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Label component in Office UI Fabric**

Use Labels to name or title a component or group of components. When paired with another component or group of components, Labels should be in close proximity to the related components or groups. Some components have predefined labels, such as a Dropdown or Toggle.

**Example: Label in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_label.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Use sentence casing, for example “First name.” | Don’t use title casing, for example “First Name.” |
| Be short and concise. | Don’t use full sentences or complex punctuation, such as colons or semicolons. |
| When adding a Label to components, use a noun or short noun phrase as the Label text. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Default label** | Use for standard labels. | [Default Label image](https://github.com/OfficeDev/office-js-docs/blob/master/images/label.png) |
| **Disabled label** | Use when the related component is disabled. | [Disabled Label image](https://github.com/OfficeDev/office-js-docs/blob/master/images/labelDisabled.png) |
| **Required label** | Use when the related component is required. | [Required Label image](https://github.com/OfficeDev/office-js-docs/blob/master/images/labelRequired.png) |

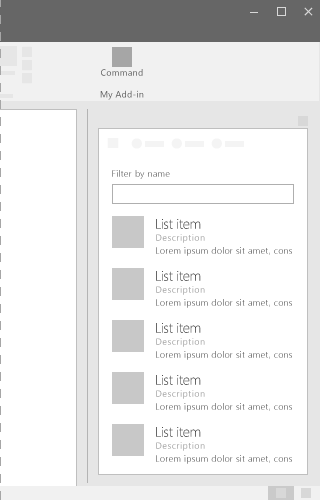
**Implementation**

For details, see [Label](https://dev.office.com/fabric#/components/label) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

# List component in Office UI Fabric

Lists provide a consistent layout when interacting with large collections of content, even when the content is text-heavy. Lists allow users to easily scan through content.

#### Example: List in a task pane

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_list.png)

## Best Practices

* End each entry with a period if all entries are complete sentences, if:
  + One or more of the entries contains both a fragment and a complete sentence that offers additional information.
  + The entries all complete the introductory sentence or fragment.
* Separate consecutive lists on a page with a heading.

## Variants

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Grid List** | Use when detailed information is needed with each list item. | [Grid List image](https://github.com/OfficeDev/office-js-docs/blob/master/images/list.png) |
| **Variable height item list** | Use when large imagery and text is needed. | [Variable height item List image](https://github.com/OfficeDev/office-js-docs/blob/master/images/listGrid.png) |

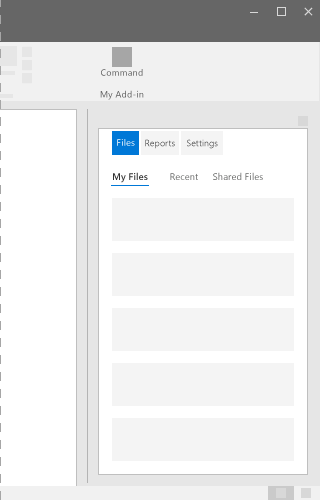
## Implementation

For details, see [List](https://dev.office.com/fabric#/components/list) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Pivot component in Office UI Fabric**

Pivots provide quick navigation to frequently accessed content. Pivots allow for navigation between two or more content views. Text headers specify which content is in each section of the Pivot. Content in each section of the Pivot may belong to distinct content categories. In Office Add-ins, use the Pivot control with tab styles. The tabs may use a combination of icons and text to communicate the type of content that tab contains.

**Example: Pivot in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_pivot.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Navigation labels should be concise, ideally using only one or two words rather than a phrase. | Don’t use full sentences or complex punctuation, such as colons or semicolons. |
| Persist Pivot headers on-screen even if another tab is selected. |  |
| Limit Pivot controls to 3-5 tabs. |  |
| Use Pivots as navigational elements close to the top of the page. Don't mix Pivots into page content. |  |
| Use Pivots on content-heavy pages that require a significant amount of scrolling. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Basic Example** | Use as the default pivot option. | [Basic Example image](https://github.com/OfficeDev/office-js-docs/blob/master/images/pivotBasic.png) |
| **Links of Tab Style** | Use when tab style pivot buttons are preferred. | [Links of Tab Style image](https://github.com/OfficeDev/office-js-docs/blob/master/images/pivotTab.png) |

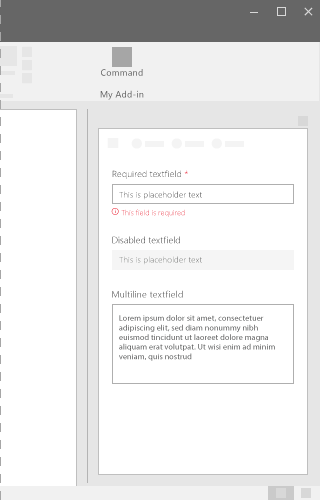
**Implementation**

For details, see [Pivot](https://dev.office.com/fabric#/components/pivot) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**TextField component in Office UI Fabric**

TextFields enable users to type text. It's typically used to capture a single line of text but can be configured to capture multiple lines of text. The text displays on the screen in a simple, uniform format.

**Example: TextField in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_textField.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Use TextFields to accept data input on a form or page. | Don’t use TextFields to render basic copy as part of a body element of a page. |
| Label TextFields with a helpful names. | Don’t use TextFields for date or time entry. Instead, use a Datetime picker. |
| Use concise placeholder text to specify what content should be entered. | Don’t use TextFields if you can predefine valid input options. Instead, use a Dropdown. |
| Provide all appropriate states for the TextField (static, hover, focus, engaged, unavailable, error). |  |
| Clearly mark required and optional fields. |  |
| Whenever possible, format TextFields according to the expected data format. For example, when capturing a 10-digit phone number, use 3 separate fields to store the different parts of the phone number. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Default TextField** | Use as the default Textfield. | [Default TextField image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textfieldDefault.png) |
| **Disabled TextField** | Use when the Textfield is disabled. | [Disabled TextField image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textfieldDisabled.png) |
| **Required TextField** | Use when the Textfield input is required. | [Required TextField image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textfieldRequired.png) |
| **TextField with a placeholder** | Use when placeholder text is needed. | [TextField with a placeholder image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textfieldPlaceholder.png) |
| **TextField with multiple lines** | Use when many lines of text are needed. | [TextField with a placeholder image](https://github.com/OfficeDev/office-js-docs/blob/master/images/textfieldMulti.png) |

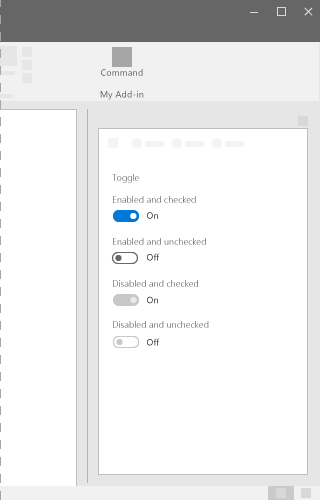
**Implementation**

For details, see [TextField](https://dev.office.com/fabric#/components/textfield) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

**Toggle component in Office UI Fabric**

Toggles represent a physical switch to turn things on or off. Use Toggles to present two mutually exclusive options (for example, on and off), where choosing an option results in an immediate action.

**Example: Toggle in a task pane**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/overview_withApp_toggle.png)

**Best Practices**

| **Do** | **Don't** |
| --- | --- |
| Use Toggles for binary settings when changes are immediately applied (see image below). | Don’t use Toggles if users must perform an extra step before changes take effect (see image below). |
| [Do Toggle example](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleDo.png) | [Don't Toggle example](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleDont.png) |
| Only replace the On and Off labels if there are more specific labels to use for a setting. Use short (3-4 character) labels that represent binary opposites. |  |

**Variants**

| **Variation** | **Description** | **Example** |
| --- | --- | --- |
| **Enabled and checked** | Use when the toggled state is active. | [Enabled and checked image](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleEnabledOn.png) |
| **Enabled and unchecked** | Use when the toggled state is inactive. | [Enabled and unchecked image](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleEnabledOff.png) |
| **Disabled and checked** | Use when the active state cannot be changed. | [Disabled and checked image](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleDisabledOn.png) |
| **Disabled and unchecked** | Use when the inactive state cannot be changed. | [Disabled and unchecked image](https://github.com/OfficeDev/office-js-docs/blob/master/images/toggleDisabledOff.png) |

**Implementation**

For details, see [Toggle](https://dev.office.com/fabric#/components/toggle) and [Getting started with Fabric React code sample](https://github.com/OfficeDev/Word-Add-in-GettingStartedFabricReact).

# Office Add-ins development lifecycle

**Note:** When you build your add-in, if you plan to [publish](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md) your add-in to the Office Store, make sure that you conform to the [Office Store validation policies](https://msdn.microsoft.com/en-us/library/jj220035.aspx). For example, to pass validation, your add-in must work across all platforms that support the methods that you define (for more information, see [section 4.12](https://msdn.microsoft.com/en-us/library/jj220035.aspx#Anchor_3) and the [Office Add-in host and availability page](https://dev.office.com/add-in-availability)).

The typical development lifecycle of an Office Add-in includes the following steps:

1. **Decide on the purpose of the add-in.**

Ask the following questions:

* + How is the add-in useful?
  + How does it help your customers be more productive?
  + What scenarios does your add-in's features support?

Decide the most important features and scenarios and focus your design around them.

1. **Identify the data and data source for the add-in.**

Is the data in a document, workbook, presentation, project, or an Access browser-based database, or about an item or items in an Exchange Server or Exchange Online mailbox? Is the data from an external source such as a web service?

1. **Identify the type of add-in and Office host applications that best support the purpose of the add-in.**

Consider the following to identify the scenarios:

* + Will customers use the add-in to enrich the content of a document or Access browser-based database? If so, you may want to consider creating a content add-in.
  + Will customers use the add-in while viewing or composing an email message or appointment? Is being able to expose the add-in according to the current context important? Is making the add-in available on not just the desktop, but also on tablets and phones a priority?

If you answer yes to any of these questions, consider creating an Outlook add-in. Then identify the context that will trigger your add-in (for example, the user being in a compose form, specific message types, the presence of an attachment, address, task suggestion, or meeting suggestion, or certain string patterns in the contents of an email or appointment). See [Activation rules for Outlook add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/manifests/activation-rules.md) to find out how you can contextually activate the Outlook add-in.

* + Will customers use the add-in to enhance the viewing or authoring experience of a document? If so, you may want to consider creating a task pane add-in.

Support for certain Add-in APIs may differ between Office applications and the platform they are running on (Windows, Mac, Web, Mobile). To see the current API coverage by client and platform, please see our [Office Add-in host and platform availability](https://dev.office.com/add-in-availability) page.

1. **Design and implement the user experience and user interface for the add-in.**

Design a fast and fluid user experience that is consistent, easy to learn, with primary scenarios that require only a few steps to complete. Depending on the purpose of the add-in, make use of third party APIs or web services.

You can choose from a variety of web development tools and use HTML and JavaScript to implement the user interface.

1. **Create an XML manifest file based on the Office Add-ins manifest schema.**

Create an XML manifest to identify the add-in and its requirements, specify the locations of the HTML and any JavaScript and CSS files that the add-in uses and, depending on the type of the add-in, the default size and permissions.

For Outlook add-ins, you can specify the context, based on the current message or appointment, under which your add-in is relevant and you would like Outlook to make it available in the UI. You can also decide which devices you want the add-in to support. In the manifest, specify the context as activation rules and the supported devices.

1. **Install and test the add-in.**

Place the HTML files and any JavaScript and CSS files on the web servers that are specified in the add-in manifest file. The process to install an add-in depends on the type of the add-in. For details, see [Sideload Office Add-ins for testing](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/create-a-network-shared-folder-catalog-for-task-pane-and-content-add-ins.md).

For Outlook add-ins, install it in an Exchange mailbox, and specify the location of the add-in manifest file in the Exchange Admin Center (EAC). For more information, see [Deploy and install Outlook add-ins for testing](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/testing-and-tips.md).

1. **Publish the add-in.**

You can submit the add-in to the Office Store, from which customers can install the add-in. In addition, you can publish task pane and content add-ins to a private folder add-in catalog on SharePoint or to a shared network folder, and you can deploy an Outlook add-in directly on an Exchange server for your organization. For details, see [Publish your Office Add-in](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md).

1. **Maintain the add-in**

If your add-in calls a web service, and if you make updates to the web service after publishing the add-in, you do not have to republish the add-in. However, if you change any items or data you submitted for your add-in, such as the add-in manifest, screenshots, icons, HTML or JavaScript files, you will need to republish the add-in. In particular, if you have published the add-in to the Office Store, you'll need to resubmit your add-in so that the Office Store can implement those changes. You must resubmit your add-in with an updated add-in manifest that includes a new version number. You must also make sure to update the add-in version number in the submission form to match the new manifest's version number. For Outlook add-ins, you should make sure the [Id](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/id.md) element contains a different UUID in the add-in manifest.

# Install the latest version of Office 2016

New developer features, including those still in preview, are delivered first to subscribers who opt in to get the latest builds of Office. To opt in the latest builds of Office 2016:

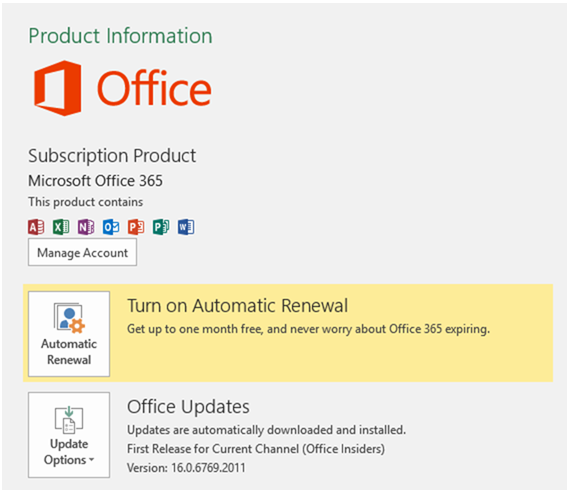
* If you're an Office 365 Home, Personal, or University subscriber, see [Be an Office Insider](https://products.office.com/en-us/office-insider).
* If you're an Office 365 for business customer, see [Install the First Release build for Office 365 for business customers](https://support.office.com/en-us/article/Install-the-First-Release-build-for-Office-365-for-business-customers-4dd8ba40-73c0-4468-b778-c7b744d03ead?ui=en-US&rs=en-US&ad=US).
* If you're running Office 2016 on a Mac:
  + Start an Office 2016 for Mac program.
  + Select **Check for Updates** on the Help menu.
  + In the Microsoft AutoUpdate box, check the box to join the Office Insider program.

To get the latest build:

1. Download the [Office 2016 Deployment Tool](https://www.microsoft.com/en-us/download/details.aspx?id=49117).
2. Run the tool. This extracts the following two files: Setup.exe and configuration.xml.
3. Replace the configuration.xml file with the [First Release Configuration File](https://raw.githubusercontent.com/OfficeDev/Office-Add-in-Commands-Samples/master/Tools/FirstReleaseConfig/configuration.xml).
4. Run the following command as an administrator: setup.exe /configure configuration.xml

**Note:** The command might take a long time to run without indicating progress.

When the installation process finishes, you will have the latest Office 2016 applications installed. To verify that you have the latest build, go to **File** > **Account** from any Office application. Under Office Updates, you'll see the (Office Insiders) label above the version number.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/officeinsider.PNG)

## Minimum Office builds for Office JavaScript API requirement sets

For information about the minimum product builds for each platform for the API requirement sets, see the following:

* [Word JavaScript API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/word-api-requirement-sets.md)
* [Excel JavaScript API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/excel-api-requirement-sets.md)
* [OneNote JavaScript API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/onenote-api-requirement-sets.md)
* [Dialog API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/dialog-api-requirement-sets.md)
* [Office common API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/office-add-in-requirement-sets.md)

# Develop Office Add-ins for the iPad

The following table lists the tasks to perform to develop an Office Add-in to run in Office for iPad.

| **Task** | **Description** | **Resources** |
| --- | --- | --- |
| Update your add-in to support Office.js version 1.1. | Update the JavaScript files (Office.js and app-specific .js files) and the add-in manifest validation file used in your Office Add-in project to version 1.1. | [What's changed in the JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/what's-changed-in-the-javascript-api-for-office.md) |
| Apply UI design best practices. | Integrate your add-in UI seamlessly with the iOS experience. | [Designing for iOS](https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/) |
| Apply add-in design best practices. | Ensure that your add-in provides clear value, is engaging, and performs consistently. | [Best practices for developing Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-development-best-practices.md) |
| Optimize your add-in for touch. | Make your UI responsive to touch inputs in addition to mouse and keyboard. | [Apply UX design principles](https://msdn.microsoft.com/EN-US/library/mt590883.aspx#Anchor_3) |
| Make your add-in free. | Office on iPad is a channel through which you can reach more users and promote your services. These new users have the potential to become your customers. | [Validation policy 10.8](http://msdn.microsoft.com/library/cd90836a-523e-42f5-ab02-5123cdf9fefe%28Office.15%29.aspx) |
| Make your add-in commerce free. | Your add-in must be free of in-app purchases, trial offers, UI that aims to upsell to paid or links to any online stores where users can purchase or acquire other content, apps, or add-ins. Your Privacy Policy and Terms of Use pages must also be free of any commerce UI or Store links. | [Validation policy 3.4](http://msdn.microsoft.com/library/cd90836a-523e-42f5-ab02-5123cdf9fefe%28Office.15%29.aspx) |
| Resubmit your add-in to the Office Store. | In the Seller Dashboard, select the **Make this add-in available in the Office Add-in Catalog on iPad** check box, and provide your Apple developer ID in the Apple ID box. Review the [Office Store Application Provider Agreement](https://sellerdashboard.microsoft.com/Assets/Content/Agreements/en-US/Office_Store_Seller_Agreement_20120927.htm) to make sure you understand agreement. | [Submit Office and SharePoint Add-ins and Office 365 web apps to the Office Store](http://msdn.microsoft.com/library/ff075782-1303-4517-91cc-b3d730e9b9ae%28Office.15%29.aspx) |

Your add-in can remain as-is for Office applications that are running on other platforms. You can also serve a different UI based on the browser/device that your add-in is running on. To detect whether your add-in is running on an iPad, you can use the following APIs:

* var isTouchEnabled = [Office.context.touchEnabled](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.touchenabled.md)
* var allowCommerce = [Office.context.commerceAllowed](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.commerceallowed.md)

## Best practices for developing Office Add-ins for iOS and Mac

Apply the following best practices for developing add-ins that run on iOS:

* **Use Visual Studio to develop your add-in.**

If you develop your add-in with Visual Studio, you can [set breakpoints and debug its code](https://github.com/OfficeDev/office-js-docs/blob/master/docs/get-started/create-and-debug-office-add-ins-in-visual-studio.md#Test) in an Office host application running on Windows, before sideloading your add-in on the iPad or Mac. Because an add-in that runs in Office for iOS or Office for Mac supports the same APIs as an add-in running in Office for Windows, your add-in's code should run the same way on both platforms.

* **Specify API requirements in your add-in's manifest or with runtime checks.**

When you specify API requirements in your add-in's manifest, Office will determine if the host application supports those API members. If the API members are available in the host, then your add-in will be available in that host application. Alternatively, you can perform a runtime check to determine if a method is available in the host before using it in your add-in. Runtime checks ensure that your add-in is always available in the host, and provides additional functionality if the methods are available. For more information, see [Specify Office hosts and API requirements](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md).

For general add-in development best practices, see [Best practices for developing Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-development-best-practices.md).

# Tips for creating Office Add-ins with Angular

This article provides guidance for using Angular 2+ to create an Office Add-in as a single page application.

**Note:** Do you have something to contribute based on your experience using Angular to create Office Add-ins? You can contribute to this article in [GitHub](https://github.com/OfficeDev/office-js-docs) or provide your feedback by submitting an [issue](https://github.com/OfficeDev/office-js-docs/issues) in the repo.

For an Office Add-ins sample that's built using the Angular framework, see [Word Style Checking Add-in Built on Angular](https://github.com/OfficeDev/Word-Add-in-Angular2-StyleChecker).

## Install the TypeScript type definitions

Open an nodejs window and enter the following at the command line: npm install --save-dev @types/office-js.

## Bootstrapping must be inside Office.initialize

On any page that calls the Office, Word, or Excel JavaScript APIs, your code must first assign a method to the Office.initialize property. (If you have no initialization code, the method body can be just empty "{}" symbols, but you must not leave the Office.initialize property undefined. For details, see [Initializing your add-in](http://dev.office.com/docs/add-ins/develop/understanding-the-javascript-api-for-office#initializing-your-add-in).) Office calls this method immediately after it has initialized the Office JavaScript libraries.

**Your Angular bootstrapping code must be called inside the method that you assign to Office.initialize** to ensure that the Office JavaScript libraries have initialized first. The following is a simple example that shows how to do this. This code should be in the main.ts file of the project.

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

Office.initialize = function () {

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);

};

## Use the hash location strategy in the Angular application

Navigating between routes in the application might not work if you don't specify the hash location strategy. You can do this in one of two ways. First, you can specify a provider for the location strategy in your app module, as shown in the following example. It goes into the app.module.ts file.

import { LocationStrategy, HashLocationStrategy } from '@angular/common';

// Other imports suppressed for brevity

@NgModule({

providers: [

{ provide: LocationStrategy, useClass: HashLocationStrategy },

// Other providers suppressed

],

// Other module properties suppressed

})

export class AppModule { }

If you define your routes in a separate routing module, there is an alternative way to specify the hash location strategy. In your routing module's .ts file, pass a configuration object to the forRoot function that specifies the strategy. The following code is an example.

import { RouterModule, Routes } from '@angular/router';

// Other imports suppressed for brevity

const routes: Routes = // route definitions go here

@NgModule({

imports: [RouterModule.forRoot(routes, { useHash: true })],

exports: [RouterModule]

})

export class AppRoutingModule { }

## Consider wrapping Fabric components with Angular components

We recommend using [Office UI Fabric](http://dev.office.com/fabric#/fabric-js) styling in your add-in. Fabric includes components that come in several versions, including a version [based on TypeScript](https://github.com/OfficeDev/office-ui-fabric-js). Consider using Fabric components in your add-in by wrapping them in Angular components. For an example that shows you how to do this, see [Word Style Checking Add-in Built on Angular](https://github.com/OfficeDev/Word-Add-in-Angular2-StyleChecker). Note, for example, how the Angular component defined in [fabric.textfield.wrapper](https://github.com/OfficeDev/Word-Add-in-Angular2-StyleChecker/blob/master/app/shared/office-fabric-component-wrappers/fabric.textfield.wrapper.component.ts) imports the Fabric file TextField.ts, where the Fabric component is defined.

## Using the Office Dialog API with Angular

The Office add-in Dialog API enables your add-in to open a page in a semimodal dialog box that can exchange information with the main page, which is typically in a task pane.

The [displayDialogAsync](http://dev.office.com/reference/add-ins/shared/officeui.displaydialogasync) method takes a parameter that specifies the URL of the page that should open in the dialog box. Your add-in can have a separate HTML page (different from the base page) to pass to this parameter, or you can pass the URL of a route in your Angular appication.

It is important to remember, if you pass a route, that the dialog box creates a new window with its own execution context. Your base page and all its initialization and bootstrapping code run again in this new context, and any variables are set to their initial values in the dialog box. So this technique launches a second instance of your single page application in the dialog box. Code that changes variables in the dialog box does not change the task pane version of the same variables. Similarly, the dialog box has its own session storage, which is not accessible from code in the task pane.

## Trigger the UI update

In an Angular app, the UI sometimes does not update. This is because that part of the code runs out of the Angular zone. The solution is to put the code in the zone, as shown in the following example.

import { NgZone } from '@angular/core';

export class MyComponent {

constructor(private zone: NgZone) { }

myFunction() {

this.zone.run(() => {

// the codes that need update the UI

});

}

}

## Using Observable

Angular uses RxJS (Reactive Extensions for JavaScript), and RxJS introduces Observable and Observer objects to implement asynchronous processing. This section provides a brief introduction to using Observables; for more detailed information, see the official [RxJS](http://reactivex.io/rxjs/) documentation.

An Observable is like a Promise object in some ways - it is returned immediately from an asynchronous call, but it might not resolve until some time later. However, while a Promise is a single value (which can be an array object), an Observable is an array of objects (possibly with only a single member). This enables code to call [array methods](http://www.w3schools.com/jsref/jsref_obj_array.asp), such as concat, map, and filter, on Observable objects.

### Pushing instead of pulling

Your code "pulls" Promise objects by assigning them to variables, but Observable objects "push" their values to objects that subscribe to the Observable. The subscribers are Observer objects. The benefit of the push architecture is that new members can be added to the Observable array over time. When a new member is added, all the Observer objects that subscribe to the Observable receive a notification.

The Observer is configured to process each new object (called the "next" object) with a function. (It is also configured to respond to an error and a completion notification. See the next section for an example.) For this reason, Observable objects can be used in a wider range of scenarios than Promise objects. For example, in addition to returning an Observable from an AJAX call, the way you can return a Promise, an Observable can be returned from an event handler, such as the "changed" event handler for a text box. Each time a user enters text in the box, all the subscribed Observer objects react immediately using the latest text and/or the current state of the application as input.

### Waiting until all asynchronous calls have completed

When you want to ensure that a callback only runs when every member of a set of Promise objects has resolved, use the Promise.all() method.

myPromise.all([x, y, z]).then(

// TODO: Callback logic goes here

)

To do the same thing with an Observable object, you use the [Observable.forkJoin()](https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/forkjoin.md) method.

const source = Observable.forkJoin([x, y, z]);

const subscription = source.subscribe(

x => {

// TODO: Callback logic goes here

},

err => console.log('Error: ' + err),

() => console.log('Completed')

);

# Convert an Office Add-in task pane template in Visual Studio to TypeScript

You can use the Office Add-in JavaScript template in Visual Studio to create an add-in that uses TypeScript. After you create the new add-in in Visual Studio, you can convert the project to TypeScript. That way, you don't have to start the Office Add-in TypeScript project from scratch.

**Note:** To learn how to create an Office Add-in TypeScript project without using Visual Studio, see [Create an Office Add-in using any editor](https://github.com/OfficeDev/office-js-docs/blob/master/docs/get-started/create-an-office-add-in-using-any-editor.md).

In your TypeScript project, you can have a mix of TypeScript and JavaScript files and your project will compile. This is because TypeScript is a typed superset of JavaScript that compiles JavaScript.

This article shows you how to convert an Excel add-in task pane template in Visual Studio from JavaScript to TypeScript. You can use the same steps to convert other Office Add-in JavaScript templates to TypeScript.

To view or download the code sample that this article is based on, see [Excel-Add-In-TS-Start](https://github.com/OfficeDev/Excel-Add-In-TS-Start) on GitHub.

## Prerequisites

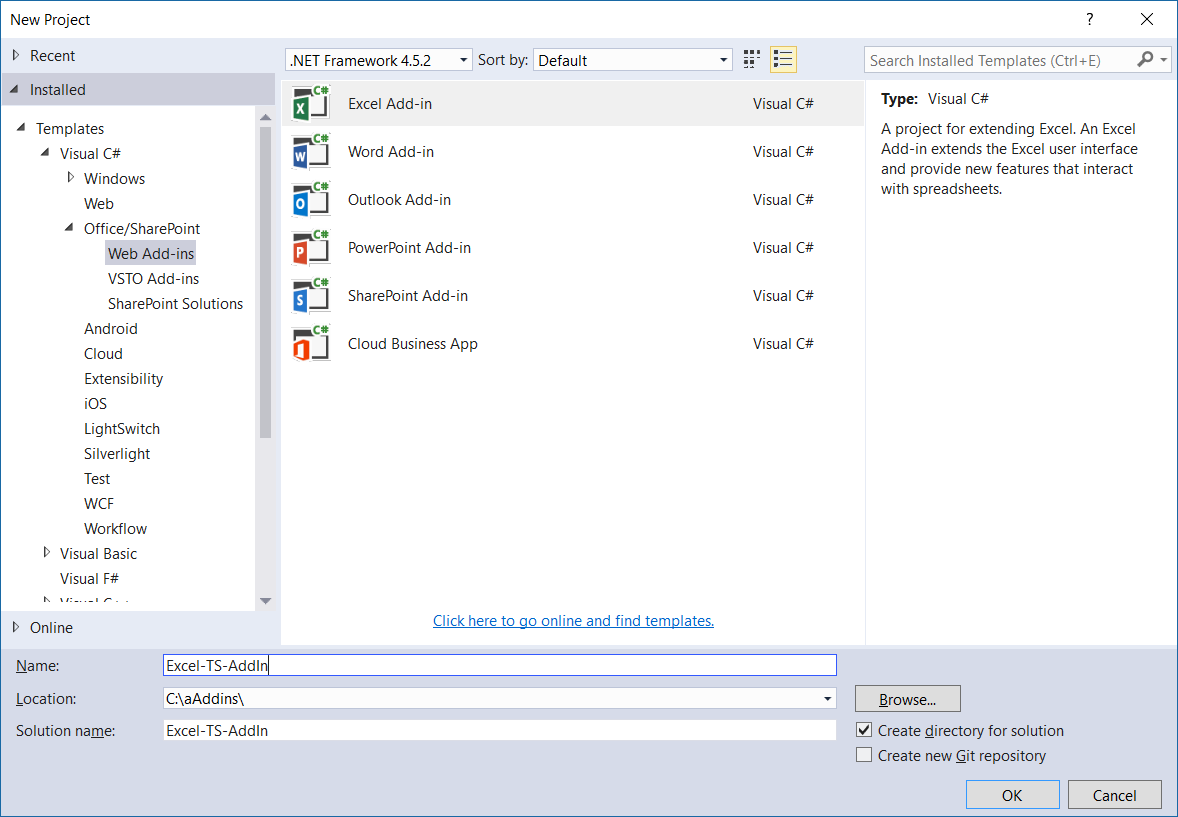
Make sure that you have the following installed:

* [Visual Studio 2015 or later](https://www.visualstudio.com/downloads/)
* [Office Developer Tools for Visual Studio](https://www.visualstudio.com/en-us/features/office-tools-vs.aspx)
* [Cumulative Servicing Release for Microsoft Visual Studio 2015 Update 3 (KB3165756)](https://msdn.microsoft.com/en-us/library/mt752379.aspx)
* Excel 2016
* [TypeScript 2.1 for Visual Studio 2015](http://download.microsoft.com/download/6/D/8/6D8381B0-03C1-4BD2-AE65-30FF0A4C62DA/TS2.1-dev14update3-20161206.2/TypeScript_Dev14Full.exe) (after you install Visual Studio 2015 Update 3)

**Note:** For more information about installing TypeScript 2.1, see [Announcing TypeScript 2.1](https://blogs.msdn.microsoft.com/typescript/2016/12/07/announcing-typescript-2-1/).

## Create new add-in project

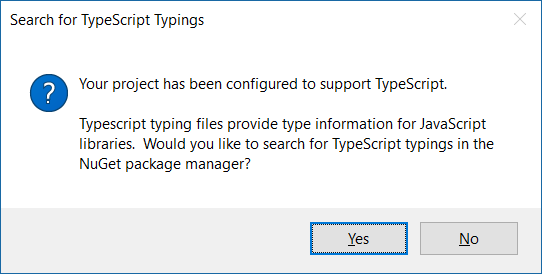
1. Open Visual Studio and go to **File** > **New** > **Project**.
2. Under **Office/SharePoint**, choose **Excel Add-in** and then choose **OK**.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/visual-studio-addin-template.png)

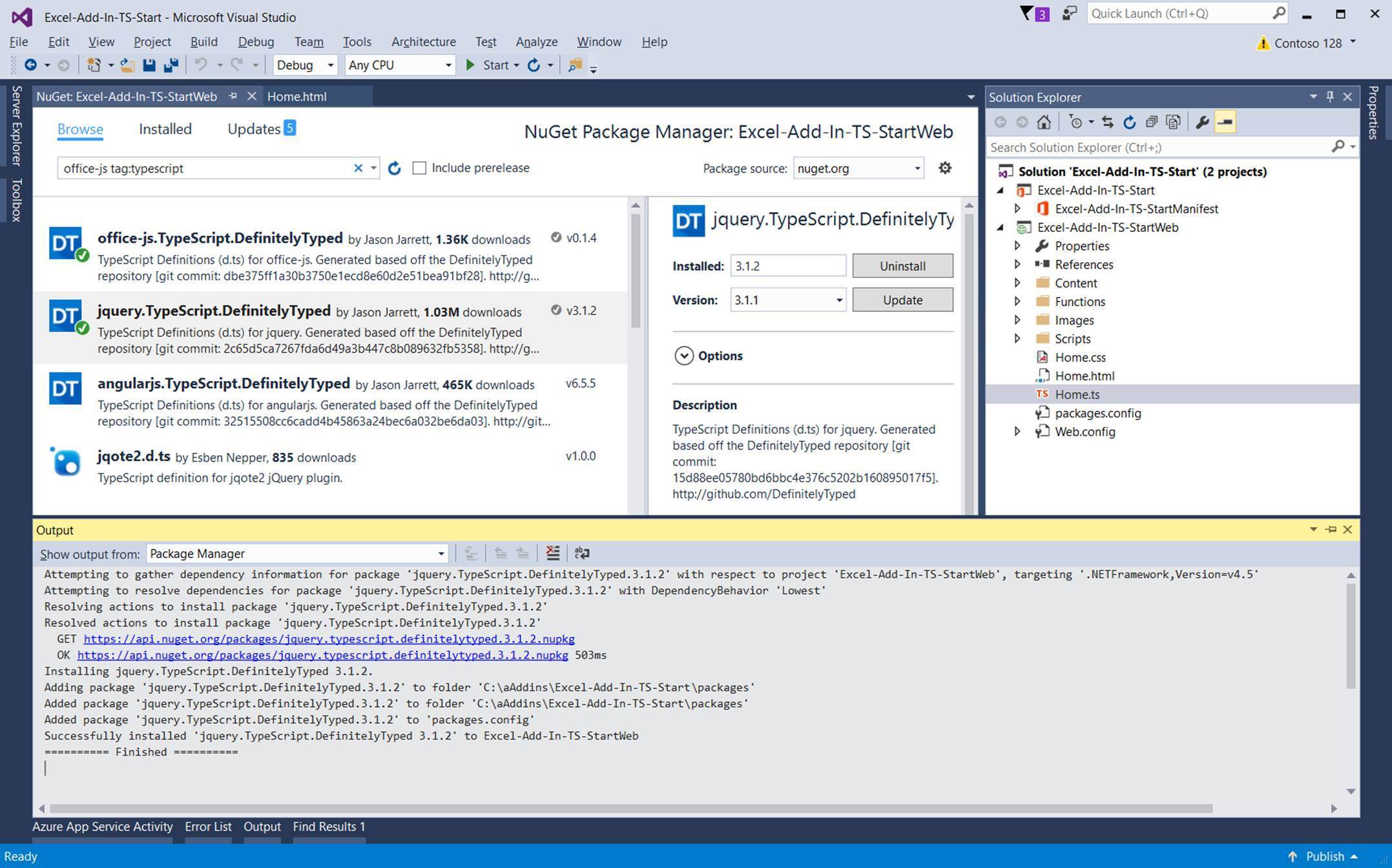
1. In the app creation wizard, choose **Add new functionalities to Excel** and choose **Finish**.
2. Do a quick test of the newly created Excel add-in by pressing F5 or the green **Start** button to launch the add-in. The add-in will be hosted locally on IIS, and Excel will open with the add-in loaded.

## Convert the add-in project to TypeScript

1. In **Solution Explorer**, change the Home.js file to Home.ts.
2. Select **Yes** when asked if you're sure you want to change file name extension.
3. Select **Yes** when asked if you want to search for TypeScript typings search on nuget, as shown in the following screenshot. This opens the **Nuget Package Manager**.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/search-typescript-typings.png)

1. Choose **Browse** in the **Nuget Package Manager**.
2. In the search box, type **office-js tag:typescript**.
3. Install **office.js.TypeScript.DefinitelyTyped** and **jquery.TypeScript.DefinitelyTyped**, as shown in the following screenshot.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/typescript-definitelytyped-nugets.png)

1. Open Home.ts (formerly Home.js). Remove the following reference from the top of the Home.ts file:

///<reference path="/Scripts/FabricUI/MessageBanner.js" />

1. Add the following declaration at the top of the Home.ts file:

declare var fabric: any;

1. Change **‘1.1’** to **1.1**; that is, remove the quotes from the following line in the Home.ts file:

if (!Office.context.requirements.isSetSupported('ExcelApi', 1.1)) {

## Run the converted add-in project

1. Press F5 or the green **Start** button to launch the add-in.
2. After Excel launches, press the **Show Taskpane** button on the **Home** ribbon.
3. Select all the cells with numbers.
4. Press the **Highlight** button on the task pane.

## Home.ts code file

For your reference, the following is the code included in the Home.ts file. This file includes the minimum number of changes needed in order for your add-in to run.

**Note:** For a complete example of a JavaScript file that has been converted to TypeScript, see [Excel-Add-In-TS-StartWeb/Home.ts](https://github.com/OfficeDev/Excel-Add-In-TS-Start/blob/master/Excel-Add-In-TS-StartWeb/Home.ts).

declare var fabric: any;

(function () {

"use strict";

var cellToHighlight;

var messageBanner;

// The initialize function must be run each time a new page is loaded.

Office.initialize = function (reason) {

$(document).ready(function () {

// Initialize the FabricUI notification mechanism and hide it

var element = document.querySelector('.ms-MessageBanner');

messageBanner = new fabric.MessageBanner(element);

messageBanner.hideBanner();

// If not using Excel 2016, use fallback logic.

if (!Office.context.requirements.isSetSupported('ExcelApi', 1.1)) {

$("#template-description").text("This sample will display the value of the cells you have selected in the spreadsheet.");

$('#button-text').text("Display!");

$('#button-desc').text("Display the selection");

$('#highlight-button').click(

displaySelectedCells);

return;

}

$("#template-description").text("This sample highlights the highest value from the cells you have selected in the spreadsheet.");

$('#button-text').text("Highlight!");

$('#button-desc').text("Highlights the largest number.");

loadSampleData();

// Add a click event handler for the highlight button.

$('#highlight-button').click(

hightlightHighestValue);

});

}

function loadSampleData() {

var values = [

[Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000)],

[Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000)],

[Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000), Math.floor(Math.random() \* 1000)]

];

// Run a batch operation against the Excel object model.

Excel.run(function (ctx) {

// Create a proxy object for the active sheet

var sheet = ctx.workbook.worksheets.getActiveWorksheet();

// Queue a command to write the sample data to the worksheet

sheet.getRange("B3:D5").values = values;

// Run the queued-up commands, and return a promise to indicate task completion

return ctx.sync();

})

.catch(errorHandler);

}

function hightlightHighestValue() {

// Run a batch operation against the Excel object model.

Excel.run(function (ctx) {

// Create a proxy object for the selected range and load its address and values properties.

var sourceRange = ctx.workbook.getSelectedRange().load("values, address, rowIndex, columnIndex, rowCount, columnCount");

// Run the queued-up command, and return a promise to indicate task completion

return ctx.sync().

.then(function () {

var highestRow = 0;

var highestCol = 0;

var highestValue = sourceRange.values[0][0];

// Find the cell to highlight

for (var i = 0; i < sourceRange.rowCount; i++) {

for (var j = 0; j < sourceRange.columnCount; j++) {

if (!isNaN(sourceRange.values[i][j]) && sourceRange.values[i][j] > highestValue) {

highestRow = i;

highestCol = j;

highestValue = sourceRange.values[i][j];

}

}

}

cellToHighlight = sourceRange.getCell(highestRow, highestCol);

sourceRange.worksheet.getUsedRange().format.fill.clear();

sourceRange.worksheet.getUsedRange().format.font.bold = false;

cellToHighlight.load("values");

})

// Run the queued-up commands.

.then(ctx.sync)

.then(function () {

// Highlight the cell

cellToHighlight.format.fill.color = "orange";

cellToHighlight.format.font.bold = true;

})

.then(ctx.sync)

})

.catch(errorHandler);

}

function displaySelectedCells() {

Office.context.document.getSelectedDataAsync(Office.CoercionType.Text,

function (result) {

if (result.status === Office.AsyncResultStatus.Succeeded) {

showNotification('The selected text is:', '"' + result.value + '"');

} else {

showNotification('Error', result.error.message);

}

});

}

// Helper function for treating errors.

function errorHandler(error) {

// Always be sure to catch any accumulated errors that bubble up from the Excel.run execution

showNotification("Error", error);

console.log("Error: " + error);

if (error instanceof OfficeExtension.Error) {

console.log("Debug info: " + JSON.stringify(error.debugInfo));

}

}

// Helper function for displaying notifications

function showNotification(header, content) {

$("#notificationHeader").text(header);

$("#notificationBody").text(content);

messageBanner.showBanner();

messageBanner.toggleExpansion();

}

})();

# Use the Dialog API in your Office Add-ins

You can use the [Dialog API](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/officeui.md) to open dialog boxes in your Office Add-in. This article provides guidance for using the Dialog API in your Office Add-in.

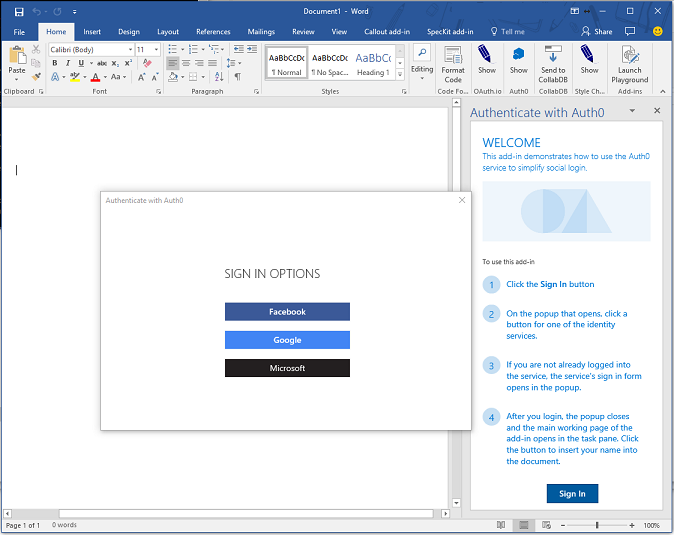
**Note:** For information about where the Dialog API is currently supported, see [Dialog API requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/requirement-sets/dialog-api-requirement-sets.md). The Dialog API is currently supported for Word, Excel, PowerPoint, and Outlook.

You might want to open a dialog box from a task pane or content add-in or [add-in command](https://dev.office.com/docs/add-ins/design/add-in-commands) to:

* Display sign-in pages that cannot be opened directly in a task pane.
* Provide more screen space, or even a full screen, for some tasks in your add-in.
* Host a video that would be too small if confined to a task pane.

**Note:** Because overlapping UI can annoy users, avoid opening a dialog from a task pane unless your scenario requires it. When you consider how to use the surface area of a task pane, note that task panes can be tabbed. For an example, see the [Excel Add-in JavaScriptSalesTracker](https://github.com/OfficeDev/Excel-Add-in-JavaScript-SalesTracker) sample.

The following image shows an example of a dialog box.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/Auth0DialogOpen.PNG)

Note that the dialog box always opens in the center of the screen. The user can move and resize it. The window is nonmodal - a user can continue to interact with both the document in the host Office application and with the host page in the task pane, if there is one.

## Dialog API scenarios

The Office JavaScript APIs support the following scenarios with a [Dialog](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/officeui.dialog.md) object and two functions in the [Office.context.ui namespace](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/officeui.md).

### Opening a dialog box

To open a dialog box, your code in the task pane calls the [displayDialogAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/officeui.displaydialogasync.md) method and passes to it the URL of the resource that should open. This is usually a page, but it can be a controller method in an MVC application, a route, a web service method, or any other resource. In this article, 'page' or 'website' refers to the resource in the dialog. The following is a simple example.

Office.context.ui.displayDialogAsync('https://myAddinDomain/myDialog.html');

**Notes:**

* The URL uses the HTTP**S** protocol. This is mandatory for all pages loaded in a dialog box, not just the first page loaded.
* The domain is the same as the domain of the host page, which can be the page in a task pane or the [function file](https://dev.office.com/reference/add-ins/manifest/functionfile) of an add-in command. This is required: the page, controller method, or other resource that is passed to the displayDialogAsync method must be in the same domain as the host page.

After the first page (or other resource) is loaded, a user can go to any website (or other resource) that uses HTTPS. You can also design the first page to immediately redirect to another site.

By default, the dialog box will occupy 80% of the height and width of the device screen, but you can set different percentages by passing a configuration object to the method, as shown in the following example.

Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html', {height: 30, width: 20});

For a sample add-in that does this, see [Office Add-in Dialog API Example](https://github.com/OfficeDev/Office-Add-in-Dialog-API-Simple-Example).

Set both values to 100% to get what is effectively a full screen experience. (The effective maximum is 99.5%, and the window is still moveable and resizable.)

**Note:** Only one dialog box can be open from a host window. An attempt to open another dialog box generates an error. (See [Errors from displayDialogAsync](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#errors-from-displaydialogAsync) for more information.) So, for example, if a user opens a dialog box from a task pane, she cannot open a second dialog box, from a different page in the task pane. However, when a dialog box is opened from an [add-in command](https://dev.office.com/docs/add-ins/design/add-in-commands), the command opens a new (but unseen) HTML file each time it is selected. This creates a new (unseen) host window, so each such window can launch its own dialog box.

### Take advantage of a performance option in Office Online

The displayInIframe property is an additional property in the configuration object that you can pass to displayDialogAsync. When this property is set to true, and the add-in is running in a document opened in Office Online, the dialog will open as a floating iframe rather than an independent window, which makes it open faster. The following is an example.

Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html', {height: 30, width: 20, displayInIframe: true});

The default value is false, which is the same as leaving the property out entirely.

If the add-in is not running in Office Online, the displayInIframe is ignored, but it does no harm for it to be present.

**Note:** You should ***not*** use displayInIframe: true if the dialog will at any point redirect to a page that cannot be opened in an iframe. For example, the sign in pages of many popular web services, such as Google and Microsoft Account, cannot be opened in an iframe.

### Sending information from the dialog box to the host page

The dialog box cannot communicate with the host page in the task pane unless:

* The current page in the dialog box is in the same domain as the host page.
* The Office JavaScript library is loaded in the page. (Like any page that uses the Office JavaScript library, script for the page must assign a method to the Office.initialize property, although it can be an empty method. For details, see [Initializing your add-in](http://dev.office.com/docs/add-ins/develop/understanding-the-javascript-api-for-office#initializing-your-add-in).)

Code in the dialog page uses the messageParent function to send either a Boolean value or a string message to the host page. The string can be a word, sentence, XML blob, stringified JSON, or anything else that can be serialized to a string. The following is an example.

if (loginSuccess) {

Office.context.ui.messageParent(true);

}

**Notes:**

* The messageParent function is one of only two Office APIs that can be called in the dialog box. (The other is Office.context.requirements.isSetSupported. For information about it, see [Specify Office hosts and API requirements](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md).)
* The messageParent function can only be called on a page with the same domain (including protocol and port) as the host page.

In the next example, googleProfile is a stringified version of the user's Google profile.

if (loginSuccess) {

Office.context.ui.messageParent(googleProfile);

}

The host page must be configured to receive the message. You do this by adding a callback parameter to the original call of displayDialogAsync. The callback assigns a handler to the DialogMessageReceived event. The following is an example. Note the following about this code:

* Office passes an [AsyncResult](https://dev.office.com/reference/add-ins/shared/asyncresult) object to the callback. It represents the result of the attempt to open the dialog box. It does not represent the outcome of any events in the dialog box. For more on this distinction, see the section [Handling errors and events](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#handling-errors-and-events).
* The value property of the asyncResult is set to a [Dialog](https://dev.office.com/reference/add-ins/shared/officeui.dialog) object, which exists in the host page, not in the dialog box's execution context.
* The processMessage is the function that handles the event. You can give it any name you want.
* The dialog variable is declared at a wider scope than the callback because it is also referenced in processMessage.

var dialog;

Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html', {height: 30, width: 20},

function (asyncResult) {

dialog = asyncResult.value;

dialog.addEventHandler(Office.EventType.DialogMessageReceived, processMessage);

}

);

The following is a simple example of a handler for the DialogMessageReceived event. Note the following about this code:

* Office passes the arg object to the handler. Its message property is the Boolean or string sent by the call of messageParent in the dialog. In this example, it is a stringified representation of a user's profile from a service such as Microsoft Account or Google, so it is deserialized back to an object with JSON.parse.
* The showUserName implementation is not shown. It might display a personalized welcome message on the task pane.

function processMessage(arg) {

var messageFromDialog = JSON.parse(arg.message);

showUserName(messageFromDialog.name);

}

When the user interaction with the dialog box is completed, your message handler should close the dialog box, as shown in this example. Note the following about this code:

* The dialog object must be the same one that is returned by the call of displayDialogAsync.
* The call of dialog.close tells Office to immediately close the dialog box.

function processMessage(arg) {

dialog.close();

// message processing code goes here;

}

For a sample add-in that uses these techniques, see [Office Add-in Dialog API Example](https://github.com/OfficeDev/Office-Add-in-Dialog-API-Simple-Example).

If the add-in needs to open a different page of the task pane after receiving the message, you can use the window.location.replace method (or window.location.href) as the last line of the handler. The following is an example.

function processMessage(arg) {

// message processing code goes here;

window.location.replace("/newPage.html");

// Alternatively ...

// window.location.href = "/newPage.html";

}

For an example of an add-in that does this, see the [Insert Excel charts using Microsoft Graph in a PowerPoint Add-in](https://github.com/OfficeDev/PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart) sample.

#### Conditional messaging

Because you can send multiple messageParent calls from the dialog box, but you have only one handler in the host page for the DialogMessageReceived event, the handler has to use conditional logic to distinguish different messages. For example, if the dialog box prompts a user to sign in to an identity provider such as Microsoft Account or Google, it sends the user's profile as a message. If authentication fails, the dialog box should send error information to the host page, as in the following example. Note the following about this code:

* The loginSuccess variable would be initialized by reading the HTTP response from the identity provider.
* The the implementation of the getProfile and getError functions are not not shown. They each get data from a query parameter or from the body of the HTTP response.
* Anonymous objects of different types are sent depending on whether the sign in was successful. Both have a messageType property, but one has a profile property and the other has an error property.

if (loginSuccess) {

var userProfile = getProfile();

var messageObject = {messageType: "signinSuccess", profile: userProfile};

var jsonMessage = JSON.stringify(messageObject);

Office.context.ui.messageParent(jsonMessage);

} else {

var errorDetails = getError();

var messageObject = {messageType: "signinFailure", error: errorDetails};

var jsonMessage = JSON.stringify(messageObject);

Office.context.ui.messageParent(jsonMessage);

}

For samples that use conditional messaging, see

* [Office Add-in that uses the Auth0 Service to Simplify Social Login](https://github.com/OfficeDev/Office-Add-in-Auth0)
* [Office Add-in that uses the OAuth.io Service to Simplify Access to Popular Online Services](https://github.com/OfficeDev/Office-Add-in-OAuth.io)

The handler code in the host page uses the value of the messageType property to branch as in the following example. Note that the showUserName function is the same as in the example above and showNotification function displays the error in the host page's UI.

function processMessage(arg) {

var messageFromDialog = JSON.parse(arg.message);

if (messageFromDialog.messageType === "signinSuccess") {

dialog.close();

showUserName(messageFromDialog.profile.name);

window.location.replace("/newPage.html");

} else {

dialog.close();

showNotification("Unable to authenticate user: " + messageFromDialog.error);

}

}

### Closing the dialog box

You can implement a button in the dialog box that will close it. To do this, the click event handler for the button should use messageParent to tell the host page that the button has been clicked. The following is an example.

function closeButtonClick() {

var messageObject = {messageType: "dialogClosed"};

var jsonMessage = JSON.stringify(messageObject);

Office.context.ui.messageParent(jsonMessage);

}

The host page handler for DialogMessageReceived would call dialog.close, as in this example. (See examples earlier in this article that show how the dialog object is initialized.)

function processMessage(arg) {

var messageFromDialog = JSON.parse(arg.message);

if (messageFromDialog.messageType === "dialogClosed") {

dialog.close();

}

}

For a sample that uses this technique, see the [dialog navigation design pattern](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/dialog/navigation) in the [UX design patterns for Office Add-ins](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code)repo.

Even when you don't have your own close dialog UI, an end user can close the dialog box by choosing the **X** in the upper-right corner. This action triggers the DialogEventReceived event. If your host pane needs to know when this happens, it should declare a handler for this event. See the section [Errors and events in the dialog window](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#errors-and-events-in-the-dialog-window) for details.

## Handling errors and events

Your code should handle two categories of events:

* Errors returned by the call of displayDialogAsync because the dialog box cannot be created.
* Errors, and other events, in the dialog window.

### Errors from displayDialogAsync

In addition to general platform and system errors, three errors are specific to calling displayDialogAsync.

| **Code number** | **Meaning** |
| --- | --- |
| 12004 | The domain of the URL passed to displayDialogAsync is not trusted. The domain must be the same domain as the host page (including protocol and port number). |
| 12005 | The URL passed to displayDialogAsync uses the HTTP protocol. HTTPS is required. (In some versions of Office, the error message returned with 12005 is the same one returned for 12004.) |
| 12007 | A dialog box is already opened from this host window. A host window, such as a task pane, can only have one dialog box open at a time. |

When displayDialogAsync is called, it always passes an [AsyncResult](https://dev.office.com/reference/add-ins/shared/asyncresult) object to its callback function. When the call is successful - that is, the dialog window is opened - the value property of the AsyncResult object is a [Dialog](https://dev.office.com/reference/add-ins/shared/officeui.dialog) object. An example of this is in the section [Sending information from the dialog to the host page](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#sending-information-from-the-dialog-to-the-host-page). When the call to displayDialogAsync fails, the window is not created, the status property of the AsyncResult object is set to "failed", and the error property of the object is populated. You should always have a callback that tests the status and responds when it's an error. The following is an example that simply reports the error message regardless of its code number.

var dialog;

Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html',

function (asyncResult) {

if (asyncResult.status === "failed") {

showNotification(asynceResult.error.code = ": " + asyncResult.error.message);

} else {

dialog = asyncResult.value;

dialog.addEventHandler(Office.EventType.DialogMessageReceived, processMessage);

}

});

### Errors and events in the dialog window

Three errors and events, known by their code numbers, in the dialog box will trigger a DialogEventReceived event in the host page.

| **Code number** | **Meaning** |
| --- | --- |
| 12002 | One of the following: - No page exists at the URL that was passed to displayDialogAsync. - The page that was passed to displayDialogAsync loaded, but the dialog box was directed to a page that it cannot find or load, or it has been directed to a URL with invalid syntax. |
| 12003 | The dialog box was directed to a URL with the HTTP protocol. HTTPS is required. |
| 12006 | The dialog box was closed, usually because the user chooses the **X** button. |

Your code can assign a handler for the DialogEventReceived event in the call to displayDialogAsync. The following is a simple example.

var dialog;

Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html',

function (result) {

dialog = result.value;

dialog.addEventHandler(Office.EventType.DialogEventReceived, processDialogEvent);

}

);

The following is an example of a handler for the DialogEventReceived event that creates custom error messages for each error code.

function processDialogEvent(arg) {

switch (arg.error) {

case 12002:

showNotification("The dialog box has been directed to a page that it cannot find or load, or the URL syntax is invalid.");

break;

case 12003:

showNotification("The dialog box has been directed to a URL with the HTTP protocol. HTTPS is required."); break;

case 12006:

showNotification("Dialog closed.");

break;

default:

showNotification("Unknown error in dialog box.");

break;

}

}

For a sample add-in that handles errors in this way, see [Office Add-in Dialog API Example](https://github.com/OfficeDev/Office-Add-in-Dialog-API-Simple-Example).

## Passing information to the dialog box

Sometimes the host page needs to pass information to the dialog box. You can do this in two primary ways:

* Add query parameters to the URL that is passed to displayDialogAsync.
* Store the information somewhere that is accessible to both the host window and dialog box. The two windows do not share a common session storage, but if they have the same domain (including port number, if any), they share a common [local storage](http://www.w3schools.com/html/html5_webstorage.asp).

### Using local storage

To use local storage, your code calls the setItem method of the window.localStorage object in the host page before the displayDialogAsync call, as in the following example.

localStorage.setItem("clientID", "15963ac5-314f-4d9b-b5a1-ccb2f1aea248");

Code in the dialog window reads the item when it's needed, as in the following example.

var clientID = localStorage.getItem("clientID");

// You can also use property syntax:

// var clientID = localStorage.clientID;

For sample add-ins that uses local storage in this way, see

* [Office Add-in that uses the Auth0 Service to Simplify Social Login](https://github.com/OfficeDev/Office-Add-in-Auth0)
* [Office Add-in that uses the OAuth.io Service to Simplify Access to Popular Online Services](https://github.com/OfficeDev/Office-Add-in-OAuth.io)

### Using query parameters

The following example shows how to pass data with a query parameter.

Office.context.ui.displayDialogAsync('https://myAddinDomain/myDialog.html?clientID=15963ac5-314f-4d9b-b5a1-ccb2f1aea248');

For a sample that uses this technique, see [Insert Excel charts using Microsoft Graph in a PowerPoint Add-in](https://github.com/OfficeDev/PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart).

Code in your dialog window can parse the URL and read the parameter value.

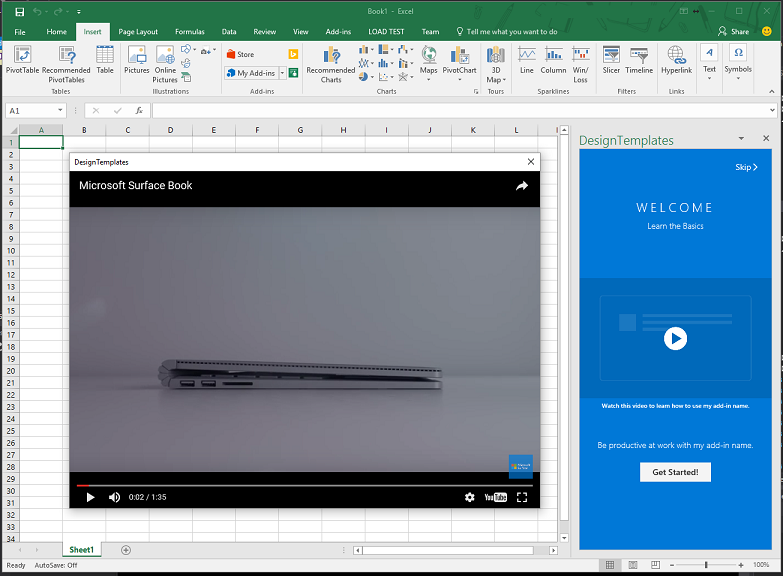
**Note**: Office automatically adds a query parameter called \_host\_info to the URL that is passed to displayDialogAsync. (It is appended after your custom query parameters, if any. It is not appended to any subsequent URLs that the dialog box navigates to.) Microsoft may change the content of this value, or remove it entirely, in the future, so your code should not read it. The same value is added to the dialog box's session storage. Again, your code should neither read nor write to this value.

## Using the Dialog APIs to show a video

To show a video in a dialog box:

1. Create a page whose only content is an iframe. The src attribute of the iframe points to an online video. The protocol of the video's URL must be HTTP**S**. In this article we'll call this page "video.dialogbox.html". The following is an example of the markup.
2. <iframe class="ms-firstrun-video\_\_player" width="640" height="360"
3. src="https://www.youtube.com/embed/XVfOe5mFbAE?rel=0&autoplay=1"
4. frameborder="0" allowfullscreen>
5. </iframe>
6. The video.dialogbox.html page must be in the same domain as the host page.
7. Use a call of displayDialogAsync in the host page to open video.dialogbox.html.
8. If your add-in needs to know when the user closes the dialog box, register a handler for the DialogEventReceived event and handle the 12006 event. For details, see the section [Errors and events in the dialog window](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#errors-and-events-in-the-dialog-window).

For a sample that shows a video in a dialog box, see the [video placemat design pattern](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code/tree/master/templates/first-run/video-placemat) in the [UX design patterns for Office Add-ins](https://github.com/OfficeDev/Office-Add-in-UX-Design-Patterns-Code) repo.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/VideoPlacematDialogOpen.PNG)

## Using the Dialog APIs in an authentication flow

A primary scenario for the Dialog APIs is to enable authentication with a resource or identity provider that does not allow its sign-in page to open in an Iframe, such as Microsoft Account, Office 365, Google, and Facebook.

**Note:** When you are using the Dialog APIs for this scenario, do not use the displayInIframe: true option in the call to displayDialogAsync. See earlier in this article for details about this option.

The following is a simple and typical authentication flow.

1. The first page that opens in the dialog box is a local page (or other resource) that is hosted in the add-in's domain; that is, the host window's domain. This page can have a simple UI that says "Please wait, we are redirecting you to the page where you can sign in to NAME-OF-PROVIDER." Code in this page constructs the URL of the identity provider's sign-in page by using information that is passed to the dialog box as described in [Passing information to the dialog box](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#passing-information-to-the-dialog-box).
2. The dialog window then redirects to the sign-in page. The URL includes a query parameter that tells the identity provider to redirect the dialog window, after the user signs in, to a specific page. In this article, we'll call this page "redirectPage.html". (This must be a page in the same domain as the host window, because the only way for the dialog window to pass the results of the sign-in attempt is with a call of messageParent, which can only be called on a page with the same domain as the host window.)
3. The identity provider's service processes the incoming GET request from the dialog window. If the user is already logged on, it immediately redirects the window to redirectPage.html and includes user data as a query parameter. If the user is not already signed in, the provider's sign-in page appears in the window, and the user signs in. For most providers, if the user cannot sign in successfully, the provider shows an error page in the dialog window and does not redirect to redirectPage.html. The user must close the window by selecting the **X** in the corner. If the user successfully signs in, the dialog window is redirected to redirectPage.html and user data is included as a query parameter.
4. When the redirectPage.html page opens, it calls messageParent to report the success or failure to the host page and optionally also report user data or error data.
5. The DialogMessageReceived event fires in the host page and its handler closes the dialog window and optionally does other processing of the message.

For sample add-ins that use this pattern, see:

* [Insert Excel charts using Microsoft Graph in a PowerPoint Add-in](https://github.com/OfficeDev/PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart): The resource that is initially opened in the dialog window is a controller method that has no view of its own. It redirects to the Office 365 sign in page.
* [Office Add-in Office 365 Client Authentication for AngularJS](https://github.com/OfficeDev/Word-Add-in-AngularJS-Client-OAuth): The resource that is initially opened in the dialog window is a page.

#### Supporting multiple identity providers

If your add-in gives the user a choice of providers, such as Microsoft Account, Google, or Facebook, you need a local first page (see preceding section) that provides a UI for the user to select a provider. Selection triggers the construction of the sign-in URL and redirection to it.

For a sample that uses this pattern, see [Office Add-in that uses the Auth0 Service to Simplify Social Login](https://github.com/OfficeDev/Office-Add-in-Auth0).

#### Authorization of the add-in to an external resource

In the modern web, web applications are security principals just as users are, and the application has its own identity and permissions to an online resource such as Office 365, Google Plus, Facebook, or LinkedIn. The application is registered with the resource provider before it is deployed. The registration includes:

* A list of the permissions that the application needs to a user's resources.
* A URL to which the resource service should return an access token when the application accesses the service.

When a user invokes a function in the application that accesses the user's data in the resource service, the user is prompted to sign in to the service and is then prompted to grant the application the permissions it needs to the user's resources. The service then redirects the sign-in window to the previously registered URL and passes the access token. The application uses the access token to access the user's resources.

You can use the Dialog APIs to manage this process by using a flow that is similar to the one described for users to sign in or the variation described in [Addressing a slow network](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/dialog-api-in-office-add-ins.md#addressing-a-slow-network). The only differences are:

* If the user hasn't previously granted the application the permissions it needs, she is prompted to do so in the dialog box after signing in.
* The dialog window sends the access token to the host window either by using messageParent to send the stringified access token or by storing the access token where the host window can retrieve it. The token has a time limit, but while it lasts, the host window can use it to directly access the user's resources without any further prompting.

The following samples use the Dialog APIs for this purpose:

* [Insert Excel charts using Microsoft Graph in a PowerPoint Add-in](https://github.com/OfficeDev/PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart) - Stores the access token in a database.
* [Office Add-in that uses the OAuth.io Service to Simplify Access to Popular Online Services](https://github.com/OfficeDev/Office-Add-in-OAuth.io)

#### More information about authentication and authorization in add-ins

* [Authorize external services in your Office Add-in](https://dev.office.com/docs/add-ins/develop/auth-external-add-ins)
* [Office JavaScript API Helpers library](https://github.com/OfficeDev/office-js-helpers)

## Using the Office Dialog API with single-page applications and client-side routing

If your add-in uses client-side routing, as single-page applications typically do, you have the option to pass the URL of a route to the [displayDialogAsync](http://dev.office.com/reference/add-ins/shared/officeui.displaydialogasync) method, instead of the URL of a complete and separate HTML page.

**Important:** The dialog box is in an new window with its own execution context. If you pass a route, your base page and all its initialization and bootstrapping code run again in this new context, and any variables are set to their initial values in the dialog window. So this technique launches a second instance of your application in the dialog window. Code that changes variables in the dialog window does not change the task pane version of the same variables. Similarly, the dialog window has its own session storage, which is not accessible from code in the task pane.

# Authorize external services in your Office Add-in

Popular online services, including Office 365, Google, Facebook, LinkedIn, SalesForce, and GitHub, enable developers to give users access to their accounts in other applications. This gives you the ability to include these services in your Office Add-in.

The industry standard framework for enabling web application access to an online service is called OAuth 2.0. In most situations, you don't need to know the details of how the framework works to make use of it in your add-in. Many libraries are available that abstract the details for you.

A fundamental idea of OAuth is that an application can be a security principal unto itself, just like a user or a group, with its own identity and set of permissions. In the most typical scenarios, when the user takes an action in the Office add-in that requires the online service, the add-in sends the service a request for a specific set of permissions to the user's account. The service then prompts the user to grant the add-in those permissions. After the permissions are granted, the service sends the add-in a small encoded access token. The add-in can use the service by including the token in all its requests to the service's APIs. But the add-in can act only within the permissions that the user granted it. The token also expires after a specified time.

Several OAuth patterns, called flows or grant types, are designed for different scenarios. The following are the two most important:

* **Implicit flow**: Communication between the add-in and the online service is implemented with client-side JavaScript.
* **Authorization Code flow**: Communication is server-to-server between your add-in's web application and the online service. So, it is implemented with server-side code.

The purpose of the flows is to secure the identity and authorization of the application. In the Authorization Code flow, you are provided a client secret that needs to be kept hidden. A Single Page Application (SPA) has no way to protect the secret, so we recommend that you use the Implicit flow in SPAs.

You should be familiar with the other pros and cons of the two flows. The official definitions at [Authorization Code](https://tools.ietf.org/html/rfc6749#section-1.3.1) and [Implicit](https://tools.ietf.org/html/rfc6749#section-1.3.2) are a good starting place.

**Note:** You also have the option of having a middleman service do all the authorizing for you and passing the access token to your add-in. For details, see the section Middleman services later in this article.

## Using the Implicit flow in Office Add-ins

The best way to find out if the online service supports the Implicit flow is to consult the documentation.

For services that support it, we provide a JavaScript library that does all the detailed work for you:

[Office-js-helpers](https://github.com/OfficeDev/office-js-helpers)

See also the **Libraries** section later in this article.

## Using the Authorization Code flow in Office Add-ins

We have some sample add-ins that use the Authorization Code flow:

* [Office-Add-in-Nodejs-ServerAuth](https://github.com/OfficeDev/Office-Add-in-Nodejs-ServerAuth) (NodeJS)
* [PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart](https://github.com/OfficeDev/PowerPoint-Add-in-Microsoft-Graph-ASPNET-InsertChart) (ASP.NET MVC)

Many libraries are available for implementing the Authorization Code flow in various languages and frameworks. For details, see the **Libraries** section later in this article.

### Relay/Proxy functions

You can use the Authorization Code flow even with a serverless web application by storing the client ID and client secretvalues in a simple function that is hosted in a service such as [Azure Functions](https://azure.microsoft.com/en-us/services/functions) or [Amazon Lambda](https://aws.amazon.com/lambda). The function exchanges a given code for an appropriate access token and relays it back to the client. The security of this approach depends on how well access to the function is guarded.

To use this technique, your add-in displays a UI/popup to show the login screen for the online service (Google, Facebook, and so on). When the user is logged on and grants the add-in permission to her resources in the online service, the developer receives a code which can be then sent to the online function. The services described in **Middleman services** in this article use a flow similar to this.

## Libraries

Libraries are available for many languages and platforms, and for both flows. Some are general purpose, others are for specific online services.

**Office 365 and other services that use Azure Active Directory as the authorization provider**: [Azure Active Directory Authentication Libraries](https://azure.microsoft.com/en-us/documentation/articles/active-directory-authentication-libraries/). A preview is also available for the [Microsoft Authentication Library](https://www.nuget.org/packages/Microsoft.Identity.Client).

**Google**: Search [GitHub.com/Google](https://github.com/google) for "auth" or the name of your language. Most of the relevant repos are named google-auth-library-[name of language].

**Facebook**: Search [Facebook for Developers](https://developers.facebook.com/) for "library" or "sdk".

**General OAuth 2.0**: A page of links to libraries for over a dozen languages is maintained by the IETF OAuth Working Group at: [OAuth Code](http://oauth.net/code/). Note that some of these libraries are for implementing an OAuth compliant service. The libraries of interest to you as a an add-in developer are called client libraries on this page because your web server is a client of the OAuth compliant service.

## Middleman services

Your add-in can use a middleman service, such as OAuth.io or Auth0, that either provides access tokens for many popular online services, or simplifies the process of enabling social login for your add-in, or both. With very little code, your add-in can use either client-side script or server-side code to connect to the middleman and it will send back any required tokens for the online service. All the authorization implementation code is in the middleman service.

We have a sample that uses Auth0 to enable social login with Facebook, Google, and Microsoft Accounts:

[Office-Add-in-Auth0](https://github.com/OfficeDev/Office-Add-in-Auth0)

We have a sample that uses OAuth.io to get access tokens from Facebook and Google:

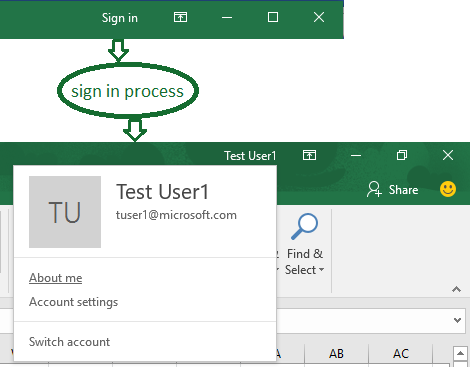
[Office-Add-in-OAuth.io](https://github.com/OfficeDev/Office-Add-in-OAuth.io)

## What is CORS?

CORS stands for [Cross Origin Resource Sharing](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS). For information about how you can use work with CORS inside add-ins, see [Addressing same-origin policy limitations in Office Add-ins](http://dev.office.com/docs/add-ins/develop/addressing-same-origin-policy-limitations).

# Enable single sign-on for Office Add-ins (preview)

You can take advantage of the fact that users can sign in to Office (online, mobile, and desktop platforms), which they can do with either their personal Microsoft account or their work or school (Office 365) account.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/OfficeHostTitleBarLogin.png)

Your add-in can use SSO to do the following - without requiring the user to sign in a second time:

* Authorize the user to your add-in.
* Authorize the add-in to access [Microsoft Graph](https://developer.microsoft.com/graph/docs).

**Note:** This feature is currently in preview and is subject to change in future releases. For this preview, single sign-on is supported only for work or school (Office 365) accounts and only for desktop versions of Office. Also, for Outlook, SSO only works if the Outlook account matches the Office user account listed in **File** > **Office account**.

For users, this makes running your add-in a smooth experience that involves at most a one-time consent screen. For developers, this means that your add-in can authenticate users and gain authorized access to the user’s data via Microsoft Graph with credentials that the user has already provided to the Office application.

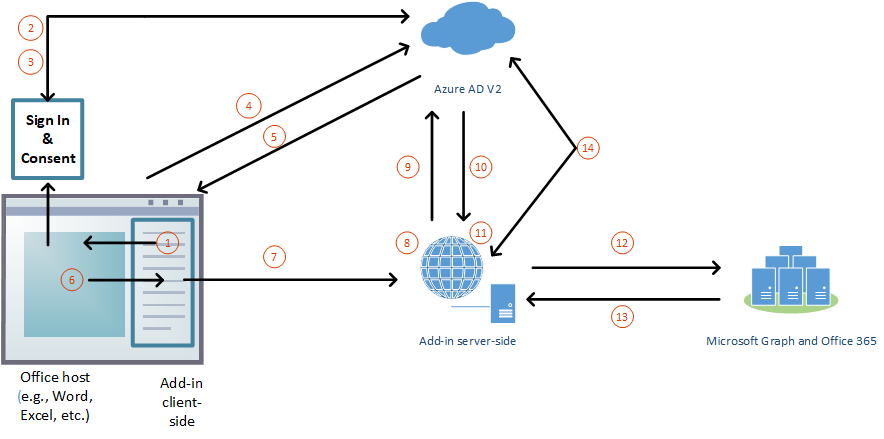
### SSO add-in architecture

In addition to hosting the pages and JavaScript of the web application, the add-in must also host, at the same [fully qualified domain name](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682135(v=vs.85).aspx#_dns_fully_qualified_domain_name_fqdn__gly), one or more web APIs that will get an access token to Microsoft Graph and make requests to it.

The add-in manifest contains markup that specifies how the add-in is registered in the Azure Active Directory (Azure AD) v2.0 endpoint, and it specifies any permissions to Microsoft Graph that the add-in needs.

### How it works at runtime

The following diagram shows how the SSO process works.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/SSOOverviewDiagram.png)

1. JavaScript in the add-in calls a new Office.js API getAccessTokenAsync. This tells the Office host application to obtain an access token to the add-in. (Hereafter, this is called the “add-in token”.)
2. [Occurs only if needed] If the user is not signed in, the Office host application opens a pop-up window for the user to sign in.
3. [Occurs only if needed] If this is the first time the current user has used your add-in, he or she is prompted to consent.
4. The Office host application requests the add-in token from the Azure AD v2.0 endpoint for the current user.
5. Azure AD sends the add-in token to the Office host application.
6. The Office host application sends the add-in token to the add-in as part of the result object returned by the call getAccessTokenAsync call.
7. JavaScript in the add-in makes an HTTP request to a web API that is hosted at the same fully-qualified domain as the add-in, and it includes the add-in token as authorization proof.
8. Server-side code validates the incoming add-in token.
9. Server-side code uses the “on behalf of” flow (defined at [OAuth2 Token Exchange](https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-02) and the [daemon or server application to web API Azure scenario](https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-authentication-scenarios#daemon-or-server-application-to-web-api)) to obtain an access token for Microsoft Graph (hereafter, the "MSG token") in exchange for the add-in token.
10. Azure AD returns the MSG token (and a refresh token, if the add-in requests offline\_access permission) to the add-in.
11. Server-side code caches the token(s).
12. Server-side code makes requests to Microsoft Graph and includes the MSG token.
13. Microsoft Graph returns data to the add-in, which can pass it on to the add-in’s UI.
14. [Occurs as needed] When the MSG token expires, the server-side code can use its refresh token to get a new MSG token.

### Develop an SSO add-in

The following sections describe the tasks involved in creating an Office Add-in that uses SSO. These tasks are described here in a language and framework agnostic way. Links to detailed walkthroughs are below.

#### Create the service application

Register the add-in at the registration portal for the Azure v2.0 endpoint: [https://apps.dev.microsoft.com](https://apps.dev.microsoft.com/). This is a 5–10 minute process that includes the following tasks:

* Get a client ID and secret for the add-in.
* Specify the permissions that your add-in needs to Microsoft Graph.
* Grant the Office host application trust to the add-in.
* Preauthorize the Office host application to the add-in with the default permission access\_as\_user.

#### Configure the add-in

Add new markup to the add-in manifest:

* **WebApplicationInfo** - The parent of the following elements.
* **Id** - The client ID of the add-in.
* **Resource** - The URL of the add-in.
* **Scopes** - The the parent of one or more **Scope** elements.
* **Scope** - Specifies a permission that the add-in needs to Microsoft Graph. For example, User.Read, Mail.Read, or offline\_access.

For Office hosts other than Outlook, add the markup to the end of the <VersionOverrides ... xsi:type="VersionOverridesV1\_0"> section. For Outlook, add the markup to the end of the <VersionOverrides ... xsi:type="VersionOverridesV1\_1"> section.

#### Add client-side code

Add JavaScript to the add-in to:

* Call Office.context.auth.getAccessTokenAsync(myTokenHandler).
* Create a handler that passes the add-in token to the add-in’s server-side code. For example:

function mytokenHandler(asyncResult) {

// Passes asyncResult.value (which has the add-in access token)

// to the add-in’s web API as an Authorization header.

}

#### Add server-side code

Create one or more Web API methods that get Microsoft Graph data. Depending on your language and framework, libraries might be available that will simplify the code you have to write. Your server-side code needs to do the following:

* Validate the add-in token that is received from the token handler you created earlier.
* Initiate the “on behalf of” flow with a call to the Azure AD v2.0 endpoint that includes the add-in access token, some metadata about the user, and the credentials of the add-in (its ID and secret).
* Cache the returned MSG token.
* Get data from Microsoft Graph by using the MSG token.

### Walkthroughs

* [Create a Node.js Office Add-in that uses single sign-on](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/create-sso-office-add-ins-nodejs.md)
* [Create an ASP.NET Office Add-in that uses single sign-on](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/create-sso-office-add-ins-aspnet.md)

# Create an ASP.NET Office Add-in that uses single sign-on (preview)

Users can sign into Office and your Office Web Add-in can take advantage of this sign-in process to authorize users to your add-in and to Microsoft Graph without requiring users to sign-on a second time. For an overview, see [Enable SSO in an Office Add-in](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/sso-in-office-add-ins.md).

This article walks you through the process of enabling single sign-on (SSO) in an add-in that is built with ASP.NET, OWIN, and Microsoft Authentication Library (MSAL) for .NET.

**Note:** For a similar article about a Node.js-based add-in, see [Create a Node.js Office Add-in that uses single sign-on](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/create-sso-office-add-ins-nodejs.md).

## Prerequisites

* The latest available version of Visual Studio 2017 Preview.

**Note:** The latest version of Visual Studio 2017 Preview is not currently compatible with the add-in manifest markup that is required for SSO. Details about how to work around this are provided in the procedures that follow.

* Office 2016, Version 1708, build 8424.nnnn or later (the Office 365 subscription version, sometimes called “Click to Run”). You might need to be an Office Insider to get this version. For more information, see [Be an Office Insider](https://products.office.com/en-us/office-insider?tab=tab-1).

## Set up the starter project

1. Clone or download the repo at [Office Add-in ASPNET SSO](https://github.com/officedev/office-add-in-aspnet-sso).
2. Open the **Before** folder and open the .sln file in Visual Studio. This is a starter project. The UI and other aspects of the add-in that are not directly connected to SSO or authorization are already done.

Note: There is also a completed version of the sample in the same repo. It is just like the add-in that you would have if you completed the procedures of this article, except that the completed project has code comments that would be redundant with the text of this article. To use the completed version, just open the \*.sln file and follow the instructions in this article, but skip the sections **Code the client side** and **Code the server** side..

1. After the project opens, build it in Visual Studio, which will cause Visual Studio to install the packages listed in the packages.config file. This can take a few seconds to several minutes depending on how many of the packages are in the computer's local package cache.
2. After the project has completely built, press F5. PowerPoint opens and there is an **SSO ASP.NET** group on the **Home**ribbon.
3. Press the **Show Add-in** button in this group to see the add-in’s UI in the task pane. The button in the task pane is not wired up yet.
4. In Visual Studio, stop the debugger.

## Register the add-in with Azure AD v2.0 endpoint

1. Navigate to <https://apps.dev.microsoft.com/> .
2. Sign-in with the admin credentials to your Office 365 tenancy. For example, [MyName@contoso.onmicrosoft.com](mailto:MyName@contoso.onmicrosoft.com)
3. Click **Add an app**.
4. When prompted, use “Office-Add-in-ASPNET-SSO” as the app name, and then press **Create application**.
5. When the configuration page for the app opens, copy the **Application Id** and save it. You will use it in a later procedure.

Note: This ID is the “audience” value when other applications, such as the Office host application (e.g., PowerPoint, Word, Excel), seek authorized access to the application. It is also the “client ID” of the application when it, in turn, seeks authorized access to Microsoft Graph.

1. In the **Application Secrets** section, press **Generate New Password**. A popup dialog opens with a new password (also called an “app secret”) displayed. Copy the password immediately and save it with the application ID. You will need it in a later procedure. Then close the dialog.
2. In the **Platforms** section, click **Add Platform**.
3. In the dialog that opens, select **Web API**.
4. An **Application ID URI** has been generated of the form “api://{App ID GUID}”. Insert the string “localhost:44355/” between the double forward slashes and the GUID. The entire ID should read api://localhost:44355/{App ID GUID}. (The domain part of the **Scope** name just below the **Application ID URI** will automatically change to match. It should read api://localhost:44355/{App ID GUID}/access\_as\_user.)
5. In the **Pre-authorized applications** section, there is an empty **Application ID** box. Enter the following ID in the box (this is the ID of Microsoft Office): d3590ed6-52b3-4102-aeff-aad2292ab01c.
6. Open the **Scope** drop down beside the **Application ID** and check the box for api://localhost:44355/{App ID GUID}/access\_as\_user.
7. Near the top of the **Platforms** section, click **Add Platform** again and select **Web**.
8. In the new **Web** section under **Platforms**, enter the following as a **Redirect URL**: https://localhost:44355.

Note: As of this writing, the **Web API** platform sometimes disappears from the **Platforms** section, particularly if the page is refreshed after the **Web** platform is added and the registration page is saved. For reassurance that your **Web API** platform is still part of the registration, click the **Edit Application Manifest** button near the bottom of the page. You should see the api://localhost:44355/{App ID GUID} string in the **identifierUris** property of the manifest. There will also be a **oauth2Permissions** property whose **value** subproperty has the value access\_as\_user.

1. Scroll down to the **Microsoft Graph Permissions** section, the **Delegated Permissions** subsection. Use the **Add** button to open a **Select Permissions** dialog.
2. In the dialog box, check the boxes for the following permissions (some may already be checked by default). (Only the first is really required by your add-in itself; but the MSAL library that the server-side code uses requires the other two.)

* Files.Read.All
* offline\_access
* openid

1. Click **OK** at the bottom of the dialog.
2. Click **Save** at the bottom of the registration page.

## Grant admin consent to the add-in

1. If the add-in is not running in Visual Studio, press F5 to run it. It needs to be running in IIS for this procedure to complete smoothly.
2. In the following string, replace the placeholder “{application\_ID}” with the Application ID that you copied when you registered your add-in.

https://login.microsoftonline.com/common/adminconsent?client\_id={application\_ID}&state=12345

1. Paste the resulting URL into a browser address bar and navigate to it.
2. When prompted, sign-in with the admin credentials to your Office 365 tenancy.
3. You are then prompted to grant permission for your add-in to access your Microsoft Graph data. Click **Accept**.
4. The browser window/tab is then redirected to the **Redirect URL** that you specified when you registered the add-in, so the home page of the add-in opens in the browser.
5. In the browser's address bar you'll see a "tenant" query parameter with a GUID value. This is the ID of your Office 365 tenancy. Copy and save this value. You will use it in a later step.
6. Close the window/tab.
7. Stop the debugger in Visual Studio.

## Configure the add-in

1. In the following string, replace the placeholder “{tenant\_ID}” with the Office 365 tenant ID you obtained earlier. If for any reason, you didn't get the ID earlier, use one of the methods in [Find your Office 365 tenant ID](https://support.office.com/en-us/article/Find-your-Office-365-tenant-ID-6891b561-a52d-4ade-9f39-b492285e2c9b) to obtain it.

https://login.microsoftonline.com/{tenant\_ID}/v2.0

1. In Visual Studio, open the web.config. There are some keys in the **appSettings** section to which you need to assign values.
2. Use the string you constructed in step 1 as the value to the key named “ida:Issuer”. Be sure there are no blank spaces in the value.
3. Give the following values to the corresponding keys:

| **Key** | **Value** |
| --- | --- |
| ida:ClientID | The application ID you obtained when you registered the add-in. |
| ida:Audience | The application ID you obtained when you registered the add-in. |
| ida:Password | TThe password you obtained when you registered the add-in. |

The following is an example of what the four keys you changed should look like. (Note that ClientID and Audience are the same. You can also use a single key for both purposes, but your web.config markup will be more reusable if you keep them separate because they aren't always the same. Also, having separate keys reinforces the idea that your add-in is both an OAuth resource - relative to the Office host - and an OAuth client - relative to Microsoft Graph.)

```xml

<add key=”ida:ClientID" value="12345678-1234-1234-1234-123456789012" />

<add key="ida:Audience" value="12345678-1234-1234-1234-123456789012" />

<add key="ida:Password" value="rFfv17ezsoGw5XUc0CDBHiU" />

<add key="ida:Issuer" value="https://login.microsoftonline.com/aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee/v2.0" />

```

**Note:** Leave the other settings in the **appSettings** section unchanged.

1. Save and close the file.
2. In the add-in project, open the add-in manifest file “Office-Add-in-ASPNET-SSO.xml”.
3. Scroll to the bottom of the file.
4. Just above the end </VersionOverrides> tag, you will find the following markup:
5. <WebApplicationInfo>
6. <Id>{application\_GUID here}</Id>
7. <Resource>api://localhost:44355/{application\_GUID here}<Resource>
8. <Scopes>
9. <Scope>openid</Scope>
10. <Scope>offline\_access</Scope>
11. <Scope>files.read.all</Scope>
12. </Scopes>

</WebApplicationInfo>

1. Replace the placeholder “{application\_GUID here}” in both places in the markup with the Application ID that you copied when you registered your add-in. This is the same ID you used in for the ClientID and Audience in the web.config.

Note:

* + The **Resource** value is the **Application ID URI** you set when you added the Web API platform to the registration of the add-in.
  + The **Scopes** section is used only to generate a consent dialog box if the add-in is sold through the Office Store.

1. Open the **Warnings** tab of the **Error List** in Visual Studio. If there is a warning that <WebApplicationInfo> is not a valid child of <VersionOverrides>, your version of Visual Studio 2017 Preview does not recognize the SSO markup. As a workaround, do the following:
   * Remove the <WebApplicationInfo> section from the manifest and save it in a text file.
   * Press F5 to start a debugging session. This will create a copy of the manifest in the following folder (which is easier to access in **File Explorer** than in Visual Studio): Office-Add-in-ASP.NET-SSO\Complete\Office-Add-in-ASPNET-SSO\bin\Debug\OfficeAppManifests
   1. Open the copy of the manifest in a text editor and paste in the <WebApplicationInfo> section just above the end </VersionOverrides>.
   2. Save the copy of the manifest.
   3. You'll need to prevent Visual Studio from overwriting the copy of the manifest when you end the next time you press F5. Right-click the solution node at the top of **Solution Explorer** (not either of the project nodes).
   4. Select **Properties** from the context menu and a **Solution Property Pages** dialog box opens.
   5. Expand **Configuration Properties** and select **Configuration**.
   6. Deselect **Build** and **Deploy** in the row for the **Office-Add-in-ASPNET-SSO** project (not the **Office-Add-in-ASPNET-SSO-WebAPI** project).
   7. Press **OK** to close the dialog box.
2. Save and close the main manifest file in Visual Studio.

## Code the client side

1. Open the Home.js file in the **Scripts** folder. It already has some code in it:
   * An assignment to the Office.initialize method that, in turn, assigns a handler to the getGraphAccessTokenButtonbutton click event.
   * A showResult method that will display data returned from Microsoft Graph (or an error message) at the bottom of the task pane.
2. Below the assignment to Office.initialize, add the code below. Note the following about this code:
   * The getAccessTokenAsync is the new API in Office.js that enables an add-in to ask the Office host application (Excel, PowerPoint, Word, etc.) for an access token to the add-in (for the user signed into Office). The Office host application, in turn, asks the Azure AD 2 endpoint for the token. Since you preauthorized the Office host to your add-in when you registered it, Azure AD will send the token.
   * If no user is signed into Office, the Office host will prompt the user to sign in.
   * The options parameter sets forceConsent to false, so the user will not be prompted to consent to giving the Office host access to your add-in.
3. function getOneDriveItems() {
4. Office.context.auth.getAccessTokenAsync({ forceConsent: false },
5. function (result) {
6. if (result.status === "succeeded") {
7. // TODO1: Use the access token to get Microsoft Graph data.
8. }
9. else {
10. console.log("Code: " + result.error.code);
11. console.log("Message: " + result.error.message);
12. console.log("name: " + result.error.name);
13. document.getElementById("getGraphAccessTokenButton").disabled = true;
14. }
15. });

}

1. Replace the TODO1 with the following lines. You create the getData method and the server-side “/api/values” route in later steps. A relative URL is used for the endpoint because it must be hosted on the same domain as your add-in.
2. accessToken = result.value;

getData("/api/values", accessToken);

1. Below the getOneDriveFiles method, add the following. This utility method calls a specified Web API endpoint and passes it the same access token that the Office host application used to get access to your add-in. On the server-side, this access token will be used in the “on behalf of” flow to obtain an access token to Microsoft Graph.
2. function getData(relativeUrl, accessToken) {
3. $.ajax({
4. url: relativeUrl,
5. headers: { "Authorization": "Bearer " + accessToken },
6. type: "GET",
7. })
8. .done(function (result) {
9. showResult(result);
10. })
11. .fail(function (result) {
12. console.log(result.error);
13. });

}

1. Save and close the file.

## Code the server side

### Configure the OWIN middleware

1. Open the Startup.cs file in the root of the project.
2. Add the keyword partial to the declaration of the Startup class, if it is not already there. It should look like this:

public partial class Startup

1. Add the following line to the body of the Configuration method. You create the ConfigureAuth method in a later step.

ConfigureAuth(app);

1. Save and close the file.
2. Right-click the **App\_Start** folder and select **Add | Class**.
3. In the **Add new item** dialog name the file **Startup.Auth.cs** and then click **Add**.
4. Shorten the namespace name in the new file to Office\_Add\_in\_ASPNET\_SSO\_WebAPI.
5. Ensure that all of the following using statements are at the top of the file.
6. using Owin;
7. using System.IdentityModel.Tokens;
8. using System.Configuration;
9. using Microsoft.Owin.Security.OAuth;
10. using Microsoft.Owin.Security.Jwt;
11. using Office\_Add\_in\_ASPNET\_SSO\_WebAPI.App\_Start;
12. Add the keyword partial to the declaration of the Startup class, if it is not already there. It should look like this:

public partial class Startup

1. Add the following method to the Startup class. This method specifies how the OWIN middleware will validate the access tokens that are passed to it from the getData method in the client-side Home.js file. The authorization process is triggered whenever a Web API endpoint that is decorated with the [Authorize] attribute is called.
2. public void ConfigureAuth(IAppBuilder app)
3. {
4. // TODO2: Configure the validation settings
5. // TODO3: Specify the type of authorization and the discovery endpoint
6. // of the secure token service.
7. }
8. Replace the TODO2 with the following. Note:
   * The code instructs OWIN to ensure that the audience and token issuer specified in the access token that comes from the Office host (and is passed on by the client-side call of getData) must match the values specified in the web.config.
   * Setting SaveSigninToken to true causes OWIN to save the raw token from the Office host. The add-in needs it to obtain an access token to Microsoft Graph with the “on behalf of” flow.
   * Scopes are not validated by the OWIN middleware. The scopes of the access token, which should include access\_as\_user, is validated in the controller.
9. var tvps = new TokenValidationParameters
10. {
11. ValidAudience = ConfigurationManager.AppSettings["ida:Audience"],
12. ValidIssuer = ConfigurationManager.AppSettings["ida:Issuer"],
13. SaveSigninToken = true
14. };
15. Replace TODO3 with the following. Note:
    * The method UseOAuthBearerAuthentication is called instead of the more common UseWindowsAzureActiveDirectoryBearerAuthentication because the latter is not compatible with the Azure AD V2 endpoint.
    * The discovery URL that is passed to the method is where the OWIN middleware obtains instructions for getting the key it needs to verify the signature on the access token received from the Office host.
16. app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions
17. {
18. AccessTokenFormat = new JwtFormat(tvps, new OpenIdConnectCachingSecurityTokenProvider("https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration"))
19. });
20. Save and close the file.

### Create the /api/values controller

1. Open the file **Controllers\ValueController.cs**.
2. Ensure that the following using statements are at the top of the file.
3. using Microsoft.Identity.Client;
4. using System.IdentityModel.Tokens;
5. using System.Collections.Generic;
6. using System.Configuration;
7. using System.Linq;
8. using System.Security.Claims;
9. using System.Threading.Tasks;
10. using System.Web.Http;
11. using Office\_Add\_in\_ASPNET\_SSO\_WebAPI.Helpers;
12. using Office\_Add\_in\_ASPNET\_SSO\_WebAPI.Models;
13. Just above the line that declares the ValuesController, add the attribute [Authorize]. This ensures that your add-in will run the authorization process that you configured in the last procedure whenever a controller method is called; so only callers with a valid access token to your add-in can invoke the methods of the controller.
14. Add the following method to the ValuesController:
15. // GET api/values
16. public async Task<IEnumerable<string>> Get()
17. {
18. // TODO4: Validate the scopes of the access token.
19. }
20. Replace TODO4 with the following code to validate that the scopes that are specified in the token include access\_as\_user.
21. string[] addinScopes = ClaimsPrincipal.Current.FindFirst("http://schemas.microsoft.com/identity/claims/scope").Value.Split(' ');
22. if (addinScopes.Contains("access\_as\_user"))
23. {
24. // TODO5: Get the raw token that the add-in page received from the Office host.
25. // TODO6: Get the access token for MS Graph.
26. // TODO7: Get the names of files and folders in OneDrive for Business by using the Microsoft Graph API.
27. // TODO8: Remove excess information from the data and send the data to the client.
28. }
29. return new string[] { "Error", "Microsoft Office does not have permission to get Microsoft Graph data on behalf of the current user." };
30. Replace TODO5 with the following code which turns the raw access token received from the Office host into a UserAssertion object that will be passed to another method.
31. var bootstrapContext = ClaimsPrincipal.Current.Identities.First().BootstrapContext as BootstrapContext;
32. UserAssertion userAssertion = new UserAssertion(bootstrapContext.Token);
33. Replace TODO6 with the following code. Note:
    * Your add-in is no longer playing the role of a resource (or audience) to which the Office host and user need access. Now it is itself a client that needs access to Microsoft Graph. ConfidentialClientApplication is the MSAL “client context” object.
    * The third parameter to the ConfidentialClientApplication constructor is a redirect URL which is not actually used in the “on behalf of” flow, but it is a good practice to use the correct URL. The fourth and fifth parameters can be used to define a persistent store that would enable the reuse of unexpired tokens across different sessions with the add-in. This sample does not implement any persistent storage.
    * MSAL requires the openid, and offline\_access scopes to function, but it throws an error if your code redundantly requests them, so only Files.Read.All is explicitly requested.
    * The ConfidentialClientApplication.AcquireTokenOnBehalfOfAsync method will first look in the MSAL cache, which is in memory, for a matching access token. Only if there isn't one, does it initiate the "on behalf of" flow with the Azure AD V2 endpoint.
34. ClientCredential clientCred = new ClientCredential(ConfigurationManager.AppSettings["ida:Password"]);
35. ConfidentialClientApplication cca =
36. new ConfidentialClientApplication(ConfigurationManager.AppSettings["ida:ClientID"],
37. "https://localhost:44355", clientCred, null, null);
38. string[] graphScopes = { "Files.Read.All" };
39. AuthenticationResult result = await cca.AcquireTokenOnBehalfOfAsync(graphScopes, userAssertion, "https://login.microsoftonline.com/common/oauth2/v2.0");
40. Replace TODO7 with the following. Note:
    * The GraphApiHelper and ODataHelper classes are defined in files in the **Helpers** folder. The OneDriveItem class is defined in a file in the **Models** folder. Detailed discussion of these classes is not relevant to authorization or SSO, so it is out-of-scope for this article.
    * Performance is improved by asking Microsoft Graph for only the data actually needed, so the code uses a $selectquery parameter to specify that we only want the name property, and a $top parameter to specify that we want only the first 3 folder of file names.
41. var fullOneDriveItemsUrl = GraphApiHelper.GetOneDriveItemNamesUrl("?$select=name&$top=3");
42. var getFilesResult = await ODataHelper.GetItems<OneDriveItem>(fullOneDriveItemsUrl, result.AccessToken);
43. Replace TODO8 with the following. Note that although the code above asked for only the name property of the OneDrive items, Microsoft Graph always includes the eTag property for OneDrive items. To reduce the payload sent to the client, the code below reconstructs the results with only the item names.
44. List<string> itemNames = new List<string>();
45. foreach (OneDriveItem item in getFilesResult)
46. {
47. itemNames.Add(item.Name);
48. }
49. return itemNames;

## Run the add-in

1. Make sure you have some files in your OneDrive for Business.
2. In Visual Studio, press F5. PowerPoint opens and there is an **SSO ASP.NET** group on the **Home** ribbon.
3. Press the **Show Add-in** button in this group to see the add-in’s UI in the task pane.
4. Press the button **Get My Files from OneDrive**. If you are not signed in to Office, you will be prompted to sign in.
5. After you are signed in, a list of your files and folders on OneDrive for Business will appear below the button. This may take over 15 seconds, especially the first time.

# Create a Node.js Office Add-in that uses single sign-on (preview)

Users can sign into Office, and your Office Web Add-in can take advantage of this sign-in process to authorize users to your add-in and to Microsoft Graph without requiring users to sign-on a second time. For an overview, see [Enable SSO in an Office Add-in](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/sso-in-office-add-ins.md).

This article walks you through the process of enabling single sign-on (SSO) in an add-in that is built with Node.js and express.

**Note:** For a similar article about an ASP.NET-based add-in, see [Create an ASP.NET Office Add-in that uses single sign-on](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/create-sso-office-add-ins-aspnet.md).

## Prerequisites

* [Node and npm](https://nodejs.org/en/), version 6.9.4 or later.
* [Git Bash](https://git-scm.com/downloads) (Or another git client.)
* TypeScript version 2.2.2 or later.
* Office 2016, Version 1708, build 8424.nnnn or later (the Office 365 subscription version, sometimes called “Click to Run”). You might need to be an Office Insider to get this version. For more information, see [Be an Office Insider](https://products.office.com/en-us/office-insider?tab=tab-1).

## Set up the starter project

1. Clone or download the repo at [Office Add-in NodeJS SSO](https://github.com/officedev/office-add-in-nodejs-sso).

**Note:** There are two versions of the sample.

* + The **Before** folder is a starter project. The UI and other aspects of the add-in that are not directly connected to SSO or authorization are already done. Later sections of this article walk you through the process of completing it.
  + The **Completed** version of the sample is just like the add-in that you would have if you completed the procedures of this article, except that the completed project has code comments that would be redundant with the text of this article. To use the completed version, just follow the instructions in this article, but replace "Before" with "Completed" and skip the sections **Code the client side** and **Code the server** side.

1. Open a Git bash console in the **Before** folder.
2. Enter npm install in the console to install all of the dependencies itemized in the package.json file.
3. Enter npm run build in the console to build the project.

Note: You may see some build errors saying that some variables are declared but not used. Ignore these errors. They are a side effect of the fact that the "Before" version of the sample missing some code that will be added later.

## Register the add-in with Azure AD V2 endpoint

1. Navigate to [https://apps.dev.microsoft.com](https://apps.dev.microsoft.com/) .
2. Sign-in with the admin credentials to your Office 365 tenancy. For example, [MyName@contoso.onmicrosoft.com](mailto:MyName@contoso.onmicrosoft.com)
3. Click **Add an app**.
4. When prompted, use “Office-Add-in-NodeJS-SSO” as the app name, and then press **Create application**.
5. When the configuration page for the app opens, copy the **Application Id** and save it. You will use it in a later procedure.

Note: This ID is the “audience” value when other applications, such as the Office host application (e.g., PowerPoint, Word, Excel), seek authorized access to the application. It is also the “client ID” of the application when it, in turn, seeks authorized access to Microsoft Graph.

1. In the **Application Secrets** section, press **Generate New Password**. A popup dialog opens with a new password (also called an “app secret”) displayed. Copy the password immediately and save it with the application ID. You will need it in a later procedure. Then close the dialog.
2. In the **Platforms** section, click **Add Platform**.
3. In the dialog that opens, select **Web API**.
4. An **Application ID URI** has been generated of the form “api://{App ID GUID}”. Insert the string “localhost:3000” between the double forward slashes and the GUID. The entire ID should read api://localhost:3000/{App ID GUID}. (The domain part of the **Scope** name just below the **Application ID URI** will automatically change to match. It should read api://localhost:3000/{App ID GUID}/access\_as\_user.)
5. This step and the next one give the Office host application access to your add-in. In the **Pre-authorized applications**section, there is an empty **Application ID** box. Enter the following ID in the box (this is the ID of Microsoft Office):d3590ed6-52b3-4102-aeff-aad2292ab01c.
6. Open the **Scope** drop down beside the **Application ID** and check the box for api://localhost:3000/{App ID GUID}/access\_as\_user.
7. Near the top of the **Platforms** section, click **Add Platform** again and select **Web**.
8. In the new **Web** section under **Platforms**, enter the following as a **Redirect URL**: https://localhost:3000.

Note: As of this writing, the **Web API** platform sometimes disappears from the **Platforms** section, particularly if the page is refreshed after the **Web** platform is added and the registration page is saved. For reassurance that your **Web API** platform is still part of the registration, click the **Edit Application Manifest** button near the bottom of the page. You should see the api://localhost:3000/{App ID GUID} string in the **identifierUris** property of the manifest. There will also be a **oauth2Permissions** property whose **value** subproperty has the value access\_as\_user.

1. Scroll down to the **Microsoft Graph Permissions** section, the **Delegated Permissions** subsection. Use the **Add** button to open a **Select Permissions** dialog.
2. In the dialog box, check the box for the following permission:
   * Files.Read.All
3. Click **OK** at the bottom of the dialog.
4. Click **Save** at the bottom of the registration page.

## Grant admin consent to the add-in

1. In the following string, replace the placeholder “{application\_ID}” with the Application ID that you copied when you registered your add-in.

https://login.microsoftonline.com/common/adminconsent?client\_id={application\_ID}&state=12345

1. Paste the resulting URL into a browser address bar and navigate to it.
2. When prompted, sign-in with the admin credentials to your Office 365 tenancy.
3. You are then prompted to grant permission for your add-in to access your Microsoft Graph data. Click **Accept**.
4. The browser window/tab is then redirected to the **Redirect URL** that you specified when you registered the add-in; so, if the add-in is running, you the home page of the add-in opens in the browser. If the add-in is not running, you will get an error saying that the resource at localhost:3000 cannot be found or opened. But the fact that the redirection was attempted means that the admin consent process completed successfully. So regardless of whether the home page opened or you got the error, you can go on to the next step.
5. In the browser's address bar you'll see a "tenant" query parameter with a GUID value. This is the ID of your Office 365 tenancy. Copy and save this value. You will use it in a later step.
6. Close the window/tab.

## Configure the add-in

1. In your code editor, open the src\server.ts file. Near the top there is a call to a constructor of an AuthModule class. There are some string parameters in the constructor to which you need to assign values.
2. For the client\_id property, replace the placeholder {client GUID} with the application ID that you saved when you registered the add-in. When you are done, there should just be a GUID in single quotation marks. There should not be any "{}" characters.
3. For the client\_secret property, replace the placeholder {client secret} with the application secret that you saved when you registered the add-in.
4. For the audience property, replace the placeholder {audience GUID} with the application ID that you saved when you registered the add-in. (The very same value that you assigned to the client\_id property.)
5. In the string assigned to the issuer property, you will see the placeholder {O365 tenant GUID}. Replace this with the Office 365 tenancy ID that you saved at the end of the last procedure. If for any reason, you didn't get the ID earlier, use one of the methods in [Find your Office 365 tenant ID](https://support.office.com/en-us/article/Find-your-Office-365-tenant-ID-6891b561-a52d-4ade-9f39-b492285e2c9b) to obtain it. When you are done, the issuer property value should look something like this:

https://login.microsoftonline.com/12345678-1234-1234-1234-123456789012/v2.0

**Note:** Leave the other parameters in the AuthModule constructor unchanged.

1. Save and close the file.
2. In the root of the project, open the add-in manifest file “Office-Add-in-NodeJS-SSO.xml”.
3. Scroll to the bottom of the file.
4. Just above the end </VersionOverrides> tag, you will find the following markup:
5. <WebApplicationInfo>
6. <Id>{application\_GUID here}</Id>
7. <Resource>api://localhost:3000/{application\_GUID here}<Resource>
8. <Scopes>
9. <Scope>files.read.all</Scope>
10. </Scopes>

</WebApplicationInfo>

1. Replace the placeholder “{application\_GUID here}” in both places in the markup with the Application ID that you copied when you registered your add-in. This is the same ID you used in for the ClientID and Audience in the web.config.

Note:

* + The **Resource** value is the **Application ID URI** you set when you added the Web API platform to the registration of the add-in.
  + The **Scopes** section is used only to generate a consent dialog box if the add-in is sold through the Office Store.

1. Save and close the file.

## Code the client side

1. Open the program.js file in the **public** folder. It already has some code in it:
   * An assignment to the Office.initialize method that, in turn, assigns a handler to the getGraphAccessTokenButtonbutton click event.
   * A showResult method that will display data returned from Microsoft Graph (or an error message) at the bottom of the task pane.
2. Below the assignment to Office.initialize, add the code below. Note the following about this code:
   * The getAccessTokenAsync is the new API in Office.js that enables an add-in to ask the Office host application (Excel, PowerPoint, Word, etc.) for an access token to the add-in (for the user signed into Office). The Office host application, in turn, asks the Azure AD 2 endpoint for the token. Since you preauthorized the Office host to your add-in when you registered it, Azure AD will send the token.
   * If no user is signed into Office, the Office host will prompt the user to sign in.
   * The options parameter sets forceConsent to false, so the user will not be prompted to consent to giving the Office host access to your add-in.
3. function getOneDriveItems() {
4. Office.context.auth.getAccessTokenAsync({ forceConsent: false },
5. function (result) {
6. if (result.status === "succeeded") {
7. // TODO1: Use the access token to get Microsoft Graph data.
8. }
9. else {
10. console.log("Code: " + result.error.code);
11. console.log("Message: " + result.error.message);
12. console.log("name: " + result.error.name);
13. document.getElementById("getGraphAccessTokenButton").disabled = true;
14. }
15. });
16. }
17. Replace the TODO1 with the following lines. You create the getData method and the server-side “/api/values” route in later steps. A relative URL is used for the endpoint because it must be hosted on the same domain as your add-in.
18. accessToken = result.value;
19. getData("/api/onedriveitems", accessToken);
20. Below the getOneDriveFiles method, add the following. This utility method calls a specified Web API endpoint and passes it the same access token that the Office host application used to get access to your add-in. On the server-side, this access token will be used in the “on behalf of” flow to obtain an access token to Microsoft Graph.
21. function getData(relativeUrl, accessToken) {
22. $.ajax({
23. url: relativeUrl,
24. headers: { "Authorization": "Bearer " + accessToken },
25. type: "GET",
26. })
27. .done(function (result) {
28. showResult(result);
29. })
30. .fail(function (result) {
31. console.log(result.error);
32. });
33. }
34. Save and close the file.

## Code the server side

There are two server-side files that need to be modified.

* The src\auth.js provides authorization helper functions. It already has generic members that are used in a variety of authorization flows. We need to add functions to it that implement the "on behalf of" flow.
* The src\server.js file has the basic members need to run a server and express middleware. We need to add functions to it that serve the home page and a Web API for obtaining Microsoft Graph data.

### Create a method to exchange tokens

1. Open the \src\auth.ts file. Add the method below to the AuthModule class. Note the following about this code:
   * The jwt parameter is the access token to the application. In the "on behalf of" flow, it is exchanged with AAD for an access token to the resource.
   * The scopes parameter has a default value, but in this sample it will be overridden by the calling code.
   * The resource parameter is optional. It should not be used when the STS is the AAD V2 endpoint. The latter infers the resource from the scopes and it returns an error if a resource is sent in the HTTP Request.
2. private async exchangeForToken(jwt: string, scopes: string[] = ['openid'], resource?: string) {
3. try {
4. // TODO2: Construct the parameters that will be sent in the body of the
5. // HTTP Request to the STS that starts the "on behalf of" flow.
6. // TODO3: Send the request to the STS.
7. // TODO4: Process the response and persist the access token to resource.
8. }
9. catch (exception) {
10. throw new UnauthorizedError('Unable to obtain an access token to the resource'
11. + ' ' + exception.message,
12. exception);
13. }
14. }
15. Replace TODO2 with the following code. About this code, note:
    * An STS that supports the "on behalf of" flow expects certain property/value pairs in the body of the HTTP request. This code constructs an object that will become the body of the request.
    * A resource property is added to the body if, and only if, a resource was passed to the method.
16. const v2Params = {
17. client\_id: this.clientId,
18. client\_secret: this.clientSecret,
19. grant\_type: 'urn:ietf:params:oauth:grant-type:jwt-bearer',
20. assertion: jwt,
21. requested\_token\_use: 'on\_behalf\_of',
22. scope: scopes.join(' ')
23. };
24. let finalParams = {};
25. if (resource) {
26. // In JavaScript we could just add the resource property to the v2Params
27. // object, but that won't compile in TypeScript.
28. let v1Params = { resource: resource };
29. for(var key in v2Params) { v1Params[key] = v2Params[key]; }
30. finalParams = v1Params;
31. } else {
32. finalParams = v2Params;
33. }
34. Replace TODO3 with the following code which sends the HTTP request to the token endpoint of the STS.
35. const res = await fetch(`${this.stsDomain}/${this.tenant}/${this.tokenURLsegment}`, {
36. method: 'POST',
37. body: form(finalParams),
38. headers: {
39. 'Accept': 'application/json',
40. 'Content-Type': 'application/x-www-form-urlencoded'
41. }
42. });
43. Replace TODO4 with the following code. Note that the code persists the access token to the resource, and it's expiration time, in addition to returning it. Calling code can avoid unnecessary calls to the STS by reusing an unexpired access token to the resource. You'll see how to do that in the next section.
44. if (res.status !== 200) {
45. const exception = await res.json();
46. throw exception;
47. }
48. const json = await res.json();
49. // Persist the token and it's expiration time.
50. const resourceToken = json['access\_token'];
51. ServerStorage.persist('ResourceToken', resourceToken);
52. const expiresIn = json['expires\_in']; // seconds until token expires.
53. const resourceTokenExpiresAt = moment().add(expiresIn, 'seconds');
54. ServerStorage.persist('ResourceTokenExpiresAt', resourceTokenExpiresAt);
55. return resourceToken;
56. Save the file, but don't close it.

### Create a method to get access to the resource using the "on behalf of" flow

1. Still in src/auth.ts, add the method below to the AuthModule class. Note the following about this code:
   * The comments above about the parameters to the the exchangeForToken method apply to the parameters of this method as well.
   * The method first checks the persistent storage for an access token to the resource that has not expired and is not going to in the next minute. It calls the method you created in the last section only if it needs to.
2. async acquireTokenOnBehalfOf(jwt: string, scopes: string[] = ['openid'], resource?: string) {
3. const resourceTokenExpirationTime = ServerStorage.retrieve('ResourceTokenExpiresAt');
4. if (moment().add(1, 'minute').diff(resourceTokenExpirationTime) < 1 ) {
5. return ServerStorage.retrieve('ResourceToken');
6. } else if (resource) {
7. return this.exchangeForToken(jwt, scopes, resource);
8. } else {
9. return this.exchangeForToken(jwt, scopes);
10. }
11. }
12. Save and close the file.

### Create the endpoints that will serve the add-in's home page and data

1. Open the src\server.ts file.
2. Add the following method to the bottom of the file. This method will serve the add-in's home page. The add-in manifest specifies the home page URL.
3. app.get('/index.html', handler(async (req, res) => {
4. return res.sendfile('index.html');
5. }));
6. Add the following method to bottom of the file. This method will handle any requests for the onedriveitems API.
7. app.get('/api/onedriveitems', handler(async (req, res) => {
8. // TODO5: Initialize the AuthModule object and validate the access token
9. // that the client-side received from the Office host.
10. // TODO6: Get a token to Microsoft Graph from either persistent storage
11. // or the "on behalf of" flow.
12. // TODO7: Use the token to get data from Microsoft Graph.
13. // TODO8: Send to the client only the data that it actually needs.
14. }));
15. Replace TODO5 with the following code which validates the access token received from the Office host application. The verifyJWT method is defined in the src\auth.ts file. It always validates the audience and the issuer. We use the optional parameter to specify that we also want it to verify that the scope in the access token is access\_as\_user. This is the only permisison to the add-in that the user and the Office host need in order to get an access token to Microsoft Graph by means of the "on behalf flow".
16. await auth.initialize();
17. const { jwt } = auth.verifyJWT(req, { scp: 'access\_as\_user' });
18. Replace TODO6 with the following line. Note the following about this code:
    * The call to acquireTokenOnBehalfOf does not include a resource parameter because we constructed the AuthModuleobject (auth) with the AAD V2 endpoint which does not support a resource property.
    * The second parameter of the call specifies the permissions the add-in will need to get a list of the user's files and folders on OneDrive for Business.

const graphToken = await auth.acquireTokenOnBehalfOf(jwt, ['Files.Read.All']);

1. Replace TODO7 with the following line. Note the following about this code:
   * The MSGraphHelper class is defined in src\msgraph-helper.ts.
   * We minimize the data that must be returned by specifying that we only want the name property and only the first 3 items.

const graphData = await MSGraphHelper.getGraphData(graphToken, "/me/drive/root/children", "?$select=name&$top=3");

1. Replace TODO8 with the following code. Note that Microsoft Graph returns some OData metadata and an **eTag** property for every item, even if name is the only property requested. The code sends only the item names to the client.
2. const itemNames: string[] = [];
3. const oneDriveItems: string[] = graphData['value'];
4. for (let item of oneDriveItems){
5. itemNames.push(item['name']);
6. }
7. return res.json(itemNames);
8. Save and close the file.

## Deploy the add-in

Now you need to let Office know where to find the add-in.

1. Create a network share, or [share a folder to the network](https://technet.microsoft.com/en-us/library/cc770880.aspx).
2. Place a copy of the Office-Add-in-NodeJS-SSO.xml manifest file, from the root of the project, into the shared folder.
3. Launch PowerPoint and open a document.
4. Choose the **File** tab, and then choose **Options**.
5. Choose **Trust Center**, and then choose the **Trust Center Settings** button.
6. Choose **Trusted Add-ins Catalogs**.
7. In the **Catalog Url** field, enter the network path to the folder share that contains Office-Add-in-NodeJS-SSO.xml, and then choose **Add Catalog**.
8. Select the **Show in Menu** check box, and then choose **OK**.
9. A message is displayed to inform you that your settings will be applied the next time you start Microsoft Office. Close PowerPoint.

## Build and run the project

There are two ways to build and run the project depending on whether you are using Visual Studio Code. For both ways, the project builds and automatically rebuilds and reruns when you make changes to the code.

1. If you are not using Visual Studio Code:
2. Open a node terminal and navigate to the root folder of the project.
3. In the terminal, enter **npm run build**.
4. Open a second node terminal and navigate to the root folder of the project.
5. In the terminal, enter **npm run start**.
6. If you are using VS Code:
7. Open the project in VS Code.
8. Press CTRL-SHIFT-B to build the project.
9. Press F5 to run the project in a debugging session.

## Add the add-in to an Office document

1. Restart PowerPoint and open or create a presentation.
2. On the **Developer** tab in PowerPoint, choose **My Add-ins**.
3. Select the **SHARED FOLDER** tab.
4. Choose **SSO NodeJS Sample**, and then select **OK**.
5. On the **Home** ribbon is a new group called **SSO NodeJS** with a button labeled **Show Add-in** and an icon.

## Test the add-in

**Note:** The preview version of the getAccessTokenAsync API only supports work or school (Office 365) identities. If you are signed into Office with a personal identity (Microsoft Account), sign out before preceding. To test the add-in, you must be either signed out entirely from Office, or signed in with a work or school account.

1. Make sure you have some files or folders in your OneDrive for Business account.
2. Click **Show Add-in** button to open the add-in.
3. The add-in opens with a Welcome page. Click the **Get my files from OneDrive** button.
4. If you are are signed into Office, a list of your files and folders on OneDrive will appear below the button. This may take more than 15 seconds the first time.
5. If you are not signed into Office, a popup will open and prompt you to sign in. After you have completed the sign-in, the list of your files and folders will appear after a few seconds. You do not press the button a second time.

# Office Add-ins XML manifest

The XML manifest file of an Office Add-in describes how your add-in should be activated when an end user installs and uses it with Office documents and applications.

An XML manifest file based on this schema enables an Office Add-in to do the following:

* Describe itself by providing an ID, version, description, display name, and default locale.
* Specify how the add-in integrates with Office, including any custom UI, such as ribbon buttons the add-in creates.
* Specify the requested default dimensions for content add-ins, and requested height for Outlook add-ins.
* Declare permissions that the Office Add-in requires, such as reading or writing to the document.
* For Outlook add-ins, define the rule or rules that specify the context in which they will be activated and interact with a message, appointment, or meeting request item.

**Note:** When you build your add-in, if you plan to [publish](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md) your add-in to the Office Store, make sure that you conform to the [Office Store validation policies](https://msdn.microsoft.com/en-us/library/jj220035.aspx). For example, to pass validation, your add-in must work across all platforms that support the methods that you define (for more information, see [section 4.12](https://msdn.microsoft.com/en-us/library/jj220035.aspx#Anchor_3) and the [Office Add-in host and availability page](https://dev.office.com/add-in-availability)).

## Required elements

The following table specifies the elements that are required for the three types of Office Add-ins.

**Required elements by Office Add-in type**

| **Element** | **Content** | **Task pane** | **Outlook** |
| --- | --- | --- | --- |
| [OfficeApp](http://msdn.microsoft.com/en-us/library/68f1cada-66f8-4341-45f5-14e0634c24fb%28Office.15%29.aspx) | X | X | X |
| [Id](http://msdn.microsoft.com/en-us/library/67c4344a-935c-09d6-1282-55ee61a2838b%28Office.15%29.aspx) | X | X | X |
| [Version](http://msdn.microsoft.com/en-us/library/6a8bbaa5-ee8c-6824-4aba-cb1a804269f6%28Office.15%29.aspx) | X | X | X |
| [ProviderName](http://msdn.microsoft.com/en-us/library/0062693a-fafa-ea2d-051a-75dac0f6c323%28Office.15%29.aspx) | X | X | X |
| [DefaultLocale](http://msdn.microsoft.com/en-us/library/04796a3a-3afa-dc85-db66-4677560c185c%28Office.15%29.aspx) | X | X | X |
| [DisplayName](http://msdn.microsoft.com/en-us/library/529159ca-53bf-efcf-c245-e572dab0ef57%28Office.15%29.aspx) | X | X | X |
| [Description](http://msdn.microsoft.com/en-us/library/bcce6bad-23d0-7631-7d8c-1064b8453b5a%28Office.15%29.aspx) | X | X | X |
| [IconUrl](http://msdn.microsoft.com/library/c7dac2d4-4fda-6fc7-3774-49f02b2d3e1e%28Office.15%29.aspx) | X | X | X |
| [HighResolutionIconUrl](http://msdn.microsoft.com/library/ff7b2647-ec8e-70dc-4e4a-e1a1377ff3f2%28Office.15%29.aspx) | X | X | X |
| [DefaultSettings (ContentApp)](http://msdn.microsoft.com/en-us/library/f7edc689-551f-1a17-ea81-ffd58f534557%28Office.15%29.aspx) [DefaultSettings (TaskPaneApp)](http://msdn.microsoft.com/en-us/library/36e3d139-56a4-fb3d-0a21-cbd14e606765%28Office.15%29.aspx) | X | X |  |
| [SourceLocation (ContentApp)](http://msdn.microsoft.com/en-us/library/00d95bb0-e8f5-647f-790a-0aa3aabc8141%28Office.15%29.aspx) [SourceLocation (TaskPaneApp)](http://msdn.microsoft.com/en-us/library/e6ea8cd4-7c8b-1da7-d8f8-8d3c80a088bc%28Office.15%29.aspx) | X | X |  |
| [DesktopSettings](http://msdn.microsoft.com/en-us/library/da9fd085-b8cc-2be0-d329-2aa1ef5d3f1c%28Office.15%29.aspx) |  |  | X |
| [SourceLocation (MailApp)](http://msdn.microsoft.com/en-us/library/3792d389-bebd-d19a-9d90-35b7a0bfc623%28Office.15%29.aspx) |  |  | X |
| [Permissions (ContentApp)](http://msdn.microsoft.com/en-us/library/9f3dcf9c-fced-c115-4f0d-38d60fb7c583%28Office.15%29.aspx) [Permissions (TaskPaneApp)](http://msdn.microsoft.com/en-us/library/d4cfe645-353d-8240-8495-f76fb36602fe%28Office.15%29.aspx) [Permissions (MailApp)](http://msdn.microsoft.com/en-us/library/c20cdf29-74b0-564c-e178-b75d148b36d1%28Office.15%29.aspx) | X | X | X |
| [Rule (RuleCollection)](http://msdn.microsoft.com/en-us/library/c6ce9d52-4b53-c6a6-de7e-c64106135c81%28Office.15%29.aspx) [Rule (MailApp)](http://msdn.microsoft.com/en-us/library/56dfc32e-2b8c-1724-05be-5595baf38aa3%28Office.15%29.aspx) |  |  | X |
| [\*Requirements (MailApp)](http://msdn.microsoft.com/en-us/library/9536ea30-34f7-76b5-7f30-1508626840e4%28Office.15%29.aspx) |  |  | X |
| [\*Set](http://msdn.microsoft.com/en-us/library/1506daa1-332c-30e1-6402-3371bcd0b895%28Office.15%29.aspx) [\*\*Sets (MailAppRequirements)](http://msdn.microsoft.com/en-us/library/2a6a2484-eeee-37e4-43bc-c185e8ae0d1d%28Office.15%29.aspx) |  |  | X |
| [\*Form](http://msdn.microsoft.com/en-us/library/77a8ac83-c22b-1225-4fc4-ba4038b68648%28Office.15%29.aspx) [\*\*FormSettings](http://msdn.microsoft.com/en-us/library/0d1a311d-939d-78c1-e968-89ddf7ebc4b4%28Office.15%29.aspx) |  |  | X |
| [\*Sets (Requirements)](http://msdn.microsoft.com/en-us/library/509be287-b532-87c6-71ac-64f3a4bbd3af%28Office.15%29.aspx) | X | X |  |
| [\*Hosts](http://msdn.microsoft.com/library/f9a739c1-3daf-c03a-2bd9-4a2a6b870101%28Office.15%29.aspx) | X | X |  |

\*Added in the Office Add-in Manifest Schema version 1.1.

**Important Notes**:

* Make sure that the add-in ID is a valid and unique GUID. Various GUID generator tools are available on the web that you can use to create a unique GUID.
* All URLs, such as the source file locations specified in the [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) element, must be **SSL-secured (HTTPS)**.
* All icon URLs, such as those used on command surfaces, must **allow caching**. The web server should NOT return HTTP headers like no-cache/no-store.
* Add-ins submitted to the Office Store must also include the [SupportUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/supporturl.md) element. For more information, see [What are some common submission errors to avoid?](http://msdn.microsoft.com/library/0ceb385c-a608-40cc-8314-78e39d6c75d0%28Office.15%29.aspx#bk_q2)
* As a best practice, only use the [AppDomains](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/appdomains.md) element to specify domains other than the one specified in the [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) element for authentication scenarios.

## Specify domains you want to open in the add-in window

By default, if your add-in tries to go to a URL in a domain other than the domain that hosts the start page (as specified in the [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) element of the manifest file), that URL will open in a new browser window outside the add-in pane of the Office host application. This default behavior protects the user against unexpected page navigation within the add-in pane from embedded **iframe** elements.

To override this behavior, specify each domain you want to open in the add-in window in the list of domains specified in the [AppDomains](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/appdomains.md) element of the manifest file. If the add-in tries to go to a URL in a domain that isn't in the list, that URL will open in a new browser window (outside the add-in pane).

The following XML manifest example hosts its main add-in page in the https://www.contoso.com domain as specified in the **SourceLocation** element. It also specifies the https://www.northwindtraders.com domain in an [AppDomain](http://msdn.microsoft.com/en-us/library/2a0353ec-5e09-6fbf-1636-4bb5dcebb9bf%28Office.15%29.aspx) element within the **AppDomains** element list. If the add-in goes to a page in the [www.northwindtraders.com](http://www.northwindtraders.com/) domain, that page will open in the add-in pane.

<?xml version="1.0" encoding="UTF-8"?>

<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="TaskPaneApp">

<Id>c6890c26-5bbb-40ed-a321-37f07909a2f0</Id>

<Version>1.0</Version>

<ProviderName>Contoso, Ltd</ProviderName>

<DefaultLocale>en-US</DefaultLocale>

<DisplayName DefaultValue="Northwind Traders Excel" />

<Description DefaultValue="Search Northwind Traders data from Excel"/>

<AppDomains>

<AppDomain>https://www.northwindtraders.com</AppDomain>

</AppDomains>

<DefaultSettings>

<SourceLocation DefaultValue="https://www.contoso.com/search\_app/Default.aspx" />

</DefaultSettings>

<Permissions>ReadWriteDocument</Permissions>

</OfficeApp>

## Manifest v1.1 XML file examples and schemas

The following sections show examples of manifest v1.1 XML files for content, task pane, and Outlook add-ins.

### Office Add-in manifest v1.1 example with commands and fallback task pane

[Task pane manifest schema](https://github.com/OfficeDev/office-js-docs/tree/master/docs/overview/schemas/taskpane)

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bt="http://schemas.microsoft.com/office/officeappbasictypes/1.0" xmlns:ov="http://schemas.microsoft.com/office/taskpaneappversionoverrides" xsi:type="TaskPaneApp">

<!-- See https://github.com/OfficeDev/Office-Add-in-Commands-Samples for documentation-->

<!-- BeginBasicSettings: Add-in metadata, used for all versions of Office unless override provided -->

<!--IMPORTANT! Id must be unique for your add-in. If you clone this manifest ensure that you change this id to your own GUID -->

<Id>e504fb41-a92a-4526-b101-542f357b7acb</Id>

<Version>1.0.0.0</Version>

<ProviderName>Contoso</ProviderName>

<DefaultLocale>en-US</DefaultLocale>

<!-- The display name of your add-in. Used on the store and various placed of the Office UI such as the add-ins dialog -->

<DisplayName DefaultValue="Add-in Commands Sample" />

<Description DefaultValue="Sample that illustrates add-in commands basic control types and actions" />

<!--Icon for your add-in. Used on installation screens and the add-ins dialog -->

<IconUrl DefaultValue="https://i.imgur.com/oZFS95h.png" />

<!--BeginTaskpaneMode integration. Office 2013 and any client that doesn't understand commands will use this section.

This section will also be used if there are no VersionOverrides -->

<Hosts>

<Host Name="Document"/>

</Hosts>

<DefaultSettings>

<SourceLocation DefaultValue="https://commandsimple.azurewebsites.net/Taskpane.html" />

</DefaultSettings>

<!--EndTaskpaneMode integration -->

<Permissions>ReadWriteDocument</Permissions>

<!--BeginAddinCommandsMode integration-->

<VersionOverrides xmlns="http://schemas.microsoft.com/office/taskpaneappversionoverrides" xsi:type="VersionOverridesV1\_0">

<Hosts>

<!--Each host can have a different set of commands. Cool huh!? -->

<!-- Workbook=Excel Document=Word Presentation=PowerPoint -->

<!-- Make sure the hosts you override match the hosts declared in the top section of the manifest -->

<Host xsi:type="Document">

<!-- Form factor. Currenly only DesktopFormFactor is supported. We will add TabletFormFactor and PhoneFormFactor in the future-->

<DesktopFormFactor>

<!--Function file is an html page that includes the javascript where functions for ExecuteAction will be called.

Think of the FunctionFile as the "code behind" ExecuteFunction-->

<FunctionFile resid="Contoso.FunctionFile.Url" />

<!--PrimaryCommandSurface==Main Office Ribbon-->

<ExtensionPoint xsi:type="PrimaryCommandSurface">

<!--Use OfficeTab to extend an existing Tab. Use CustomTab to create a new tab -->

<!-- Documentation includes all the IDs currently tested to work -->

<CustomTab id="Contoso.Tab1">

<!--Group. Ensure you provide a unique id. Recommendation for any IDs is to namespace using your company name-->

<Group id="Contoso.Tab1.Group1">

<!--Label for your group. resid must point to a ShortString resource -->

<Label resid="Contoso.Tab1.GroupLabel" />

<Icon>

<!-- Sample Todo: Each size needs its own icon resource or it will look distorted when resized -->

<!--Icons. Required sizes 16,31,80, optional 20, 24, 40, 48, 64. Strongly recommended to provide all sizes for great UX -->

<!--Use PNG icons and remember that all URLs on the resources section must use HTTPS -->

<bt:Image size="16" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="32" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="80" resid="Contoso.TaskpaneButton.Icon" />

</Icon>

<!--Control. It can be of type "Button" or "Menu" -->

<Control xsi:type="Button" id="Contoso.FunctionButton">

<!--Label for your button. resid must point to a ShortString resource -->

<Label resid="Contoso.FunctionButton.Label" />

<Supertip>

<!--ToolTip title. resid must point to a ShortString resource -->

<Title resid="Contoso.FunctionButton.Label" />

<!--ToolTip description. resid must point to a LongString resource -->

<Description resid="Contoso.FunctionButton.Tooltip" />

</Supertip>

<Icon>

<bt:Image size="16" resid="Contoso.FunctionButton.Icon" />

<bt:Image size="32" resid="Contoso.FunctionButton.Icon" />

<bt:Image size="80" resid="Contoso.FunctionButton.Icon" />

</Icon>

<!--This is what happens when the command is triggered (E.g. click on the Ribbon). Supported actions are ExecuteFuncion or ShowTaskpane-->

<!--Look at the FunctionFile.html page for reference on how to implement the function -->

- <Action xsi:type="ExecuteFunction">

<!--Name of the function to call. This function needs to exist in the global DOM namespace of the function file-->

<FunctionName>writeText</FunctionName>

</Action>

</Control>

<Control xsi:type="Button" id="Contoso.TaskpaneButton">

<Label resid="Contoso.TaskpaneButton.Label" />

<Supertip>

<Title resid="Contoso.TaskpaneButton.Label" />

<Description resid="Contoso.TaskpaneButton.Tooltip" />

</Supertip>

<Icon>

<bt:Image size="16" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="32" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="80" resid="Contoso.TaskpaneButton.Icon" />

</Icon>

<Action xsi:type="ShowTaskpane">

<TaskpaneId>Button2Id1</TaskpaneId>

<!--Provide a url resource id for the location that will be displayed on the task pane -->

<SourceLocation resid="Contoso.Taskpane1.Url" />

</Action>

</Control>

<!-- Menu example -->

<Control xsi:type="Menu" id="Contoso.Menu">

<Label resid="Contoso.Dropdown.Label" />

<Supertip>

<Title resid="Contoso.Dropdown.Label" />

<Description resid="Contoso.Dropdown.Tooltip" />

</Supertip>

<Icon>

<bt:Image size="16" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="32" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="80" resid="Contoso.TaskpaneButton.Icon" />

</Icon>

<Items>

<Item id="Contoso.Menu.Item1">

<Label resid="Contoso.Item1.Label"/>

<Supertip>

<Title resid="Contoso.Item1.Label" />

<Description resid="Contoso.Item1.Tooltip" />

</Supertip>

<Icon>

<bt:Image size="16" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="32" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="80" resid="Contoso.TaskpaneButton.Icon" />

</Icon>

<Action xsi:type="ShowTaskpane">

<TaskpaneId>MyTaskPaneID1</TaskpaneId>

<SourceLocation resid="Contoso.Taskpane1.Url" />

</Action>

</Item>

<Item id="Contoso.Menu.Item2">

<Label resid="Contoso.Item2.Label"/>

<Supertip>

<Title resid="Contoso.Item2.Label" />

<Description resid="Contoso.Item2.Tooltip" />

</Supertip>

<Icon>

<bt:Image size="16" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="32" resid="Contoso.TaskpaneButton.Icon" />

<bt:Image size="80" resid="Contoso.TaskpaneButton.Icon" />

</Icon>

<Action xsi:type="ShowTaskpane">

<TaskpaneId>MyTaskPaneID2</TaskpaneId>

<SourceLocation resid="Contoso.Taskpane2.Url" />

</Action>

</Item>

</Items>

</Control>

</Group>

<!-- Label of your tab -->

<!-- If validating with XSD it needs to be at the end, we might change this before release -->

<Label resid="Contoso.Tab1.TabLabel" />

</CustomTab>

</ExtensionPoint>

</DesktopFormFactor>

</Host>

</Hosts>

<Resources>

<bt:Images>

<bt:Image id="Contoso.TaskpaneButton.Icon" DefaultValue="https://i.imgur.com/FkSShX9.png" />

<bt:Image id="Contoso.FunctionButton.Icon" DefaultValue="https://i.imgur.com/qDujiX0.png" />

</bt:Images>

<bt:Urls>

<bt:Url id="Contoso.FunctionFile.Url" DefaultValue="https://commandsimple.azurewebsites.net/FunctionFile.html" />

<bt:Url id="Contoso.Taskpane1.Url" DefaultValue="https://commandsimple.azurewebsites.net/Taskpane.html" />

<bt:Url id="Contoso.Taskpane2.Url" DefaultValue="https://commandsimple.azurewebsites.net/Taskpane2.html" />

</bt:Urls>

<bt:ShortStrings>

<bt:String id="Contoso.FunctionButton.Label" DefaultValue="Execute Function" />

<bt:String id="Contoso.TaskpaneButton.Label" DefaultValue="Show Taskpane" />

<bt:String id="Contoso.Dropdown.Label" DefaultValue="Dropdown" />

<bt:String id="Contoso.Item1.Label" DefaultValue="Show Taskpane 1" />

<bt:String id="Contoso.Item2.Label" DefaultValue="Show Taskpane 2" />

<bt:String id="Contoso.Tab1.GroupLabel" DefaultValue="Test Group" />

<bt:String id="Contoso.Tab1.TabLabel" DefaultValue="Test Tab" />

</bt:ShortStrings>

<bt:LongStrings>

<bt:String id="Contoso.FunctionButton.Tooltip" DefaultValue="Click to Execute Function" />

<bt:String id="Contoso.TaskpaneButton.Tooltip" DefaultValue="Click to Show a Taskpane" />

<bt:String id="Contoso.Dropdown.Tooltip" DefaultValue="Click to Show Options on this Menu" />

<bt:String id="Contoso.Item1.Tooltip" DefaultValue="Click to Show Taskpane1" />

<bt:String id="Contoso.Item2.Tooltip" DefaultValue="Click to Show Taskpane2" />

</bt:LongStrings>

</Resources>

</VersionOverrides>

</OfficeApp>

### Content add-in manifest v1.1 example

[Content manifest schema](https://github.com/OfficeDev/office-js-docs/tree/master/docs/overview/schemas/content)

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp

xmlns="http://schemas.microsoft.com/office/appforoffice/1.1"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:type="ContentApp">

<Id>01eac144-e55a-45a7-b6e3-f1cc60ab0126</Id>

<AlternateId>en-US\WA123456789</AlternateId>

<Version>1.0.0.0</Version>

<ProviderName>Microsoft</ProviderName>

<DefaultLocale>en-US</DefaultLocale>

<DisplayName DefaultValue="Sample content add-in" />

<Description DefaultValue="Describe the features of this app." />

<IconUrl DefaultValue="https://contoso.com/ENUSIcon.png" />

<Hosts>

<Host Name="Workbook" />

<Host Name="Database" />

</Hosts>

<Requirements>

<Sets DefaultMinVersion="1.1">

<Set Name="TableBindings" />

</Sets>

</Requirements>

<DefaultSettings>

<SourceLocation DefaultValue="https://contoso.com/apps/content.html" />

<RequestedWidth>400</RequestedWidth>

<RequestedHeight>400</RequestedHeight>

</DefaultSettings>

<Permissions>Restricted</Permissions>

<AllowSnapshot>true</AllowSnapshot>

</OfficeApp>

### Outlook add-in manifest v1.1 example

[Content manifest schema](https://github.com/OfficeDev/office-js-docs/tree/master/docs/overview/schemas/mail)

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xmlns=

"http://schemas.microsoft.com/office/appforoffice/1.1"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:type="MailApp">

<Id>971E76EF-D73E-567F-ADAE-5A76B39052CF</Id>

<Version>1.0</Version>

<ProviderName>Microsoft</ProviderName>

<DefaultLocale>en-us</DefaultLocale>

<DisplayName DefaultValue="YouTube"/>

<Description DefaultValue=

"Watch YouTube videos referenced in the e-mails you

receive without leaving your email client.">

<Override Locale="fr-fr" Value="Visualisez les vid????os

YouTube r????f????renc????es dans vos courriers ????lectronique

directement depuis Outlook et Outlook Web App."/>

</Description>

<!-- Change the following line to specify -->

<!-- the web serverthat hosts the icon file. -->

<IconUrl DefaultValue=

"https://webserver/YouTube/YouTubeLogo.png"/>

<Hosts>

<Host Name="Mailbox" />

</Hosts>

<Requirements>

<Sets DefaultMinVersion="1.1">

<Set Name="Mailbox" />

</Sets>

</Requirements>

<FormSettings>

<Form xsi:type="ItemRead">

<DesktopSettings>

<!-- Change the following line to specify -->

<!-- the web server that hosts the HTML file. -->

<SourceLocation DefaultValue=

"https://webserver/YouTube/YouTube\_read\_desktop.htm" />

<RequestedHeight>216</RequestedHeight>

</DesktopSettings>

<TabletSettings>

<!-- Change the following line to specify -->

<!-- the web server that hosts the HTML file. -->

<SourceLocation DefaultValue=

"https://webserver/YouTube/YouTube\_read\_tablet.htm" />

<RequestedHeight>216</RequestedHeight>

</TabletSettings>

</Form>

<Form xsi:type="ItemEdit">

<DesktopSettings>

<!-- Change the following line to specify -->

<!-- the web server that hosts the HTML file. -->

<SourceLocation DefaultValue=

"https://webserver/YouTube/YouTube\_compose\_desktop.htm" />

</DesktopSettings>

<TabletSettings>

<!-- Change the following line to specify -->

<!-- the web server that hosts the HTML file. -->

<SourceLocation DefaultValue=

"https://webserver/YouTube/YouTube\_compose\_tablet.htm" />

</TabletSettings>

</Form>

</FormSettings>

<Permissions>ReadWriteItem</Permissions>

<Rule xsi:type="RuleCollection" Mode="Or">

<Rule xsi:type="RuleCollection" Mode="And">

<Rule xsi:type="RuleCollection" Mode="Or">

<Rule xsi:type="ItemIs" ItemType="Appointment" FormType="Read" />

<Rule xsi:type="ItemIs" ItemType="Message" FormType="Read" />

</Rule>

<Rule xsi:type="ItemHasRegularExpressionMatch"

PropertyName="BodyAsPlaintext" RegExName="VideoURL"

RegExValue=

"http://(((www\.)?youtube\.com/watch\?v=)|

(youtu\.be/))[a-zA-Z0-9\_-]{11}" />

</Rule>

<Rule xsi:type="RuleCollection" Mode="Or">

<Rule xsi:type="ItemIs" ItemType="Appointment" FormType="Edit" />

<Rule xsi:type="ItemIs" ItemType="Message" FormType="Edit" />

</Rule>

</Rule>

</OfficeApp>

## Validate and troubleshoot issues with your manifest

For troubleshooting issues with your manifest, see [Validate and troubleshoot issues with your manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/testing/troubleshoot-manifest.md). There, you will find information on how to validate the manifest against the [XML Schema Definition (XSD)](https://github.com/OfficeDev/office-js-docs/tree/master/docs/overview/schemas), and also how to use runtime logging to debug the manifest.

# Define add-in commands in your manifest

Add-in commands provide an easy way to customize the default Office UI with UI elements that perform actions; for example, you can add custom buttons on the ribbon. To create commands, you add a [**VersionOverrides**](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/versionoverrides.md) node to an existing manifest.

When a manifest contains the **VersionOverrides** element, versions of Word, Excel, Outlook, and PowerPoint that support add-in commands will use the information within that element to load the add-in. Earlier versions of Office products that do not support add-in commands will ignore the element.

When client applications recognize the **VersionOverrides** node, the add-in name appears in the ribbon, not in a task pane or a read/compose pane. The add-in won't appear in both places.

## VersionOverrides

The [VersionOverrides](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/versionoverrides.md) element is the root element that contains information for the add-in commands implemented by the add-in. It is supported in manifest schema v1.1 and later.

There are two versions of the **VersionOverrides** schema.

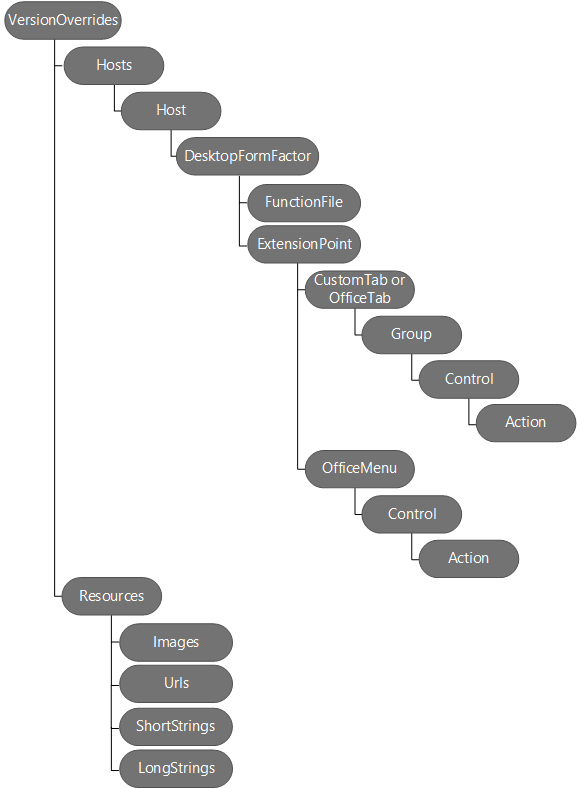
| **Schema version** | **Description** |
| --- | --- |
| 1.0 | Supports add-in commands for desktop versions of Office apps. |
| 1.1 | Adds support for [pinnable taskpanes](https://docs.microsoft.com/outlook/add-ins/pinnable-taskpane) and mobile add-ins. **Note:** Currently only supported by Outlook 2016 for Windows and Outlook for iOS |

An add-in can support multiple versions of the **VersionOverrides** schema by nesting newer versions inside of the previous version. This allows clients to support the newer versions to take advantage of the new features, while allowing older clients to load the older version. For details, see [Implementing multiple versions](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/versionoverrides.md#implementing-multiple-versions).

The **VersionOverrides** element includes the following child elements:

* [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md)
* [Requirements](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/requirements.md)
* [Hosts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/hosts.md)
* [Resources](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/resources.md)
* [VersionOverrides](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/versionoverrides.md)

The following diagram shows the hierarchy of elements used to define add-in commands.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/080da303-51c4-4882-b74a-7ba11517c0ad.png)

## Sample manifests

For a sample manifest that implements add-in commands for Word, Excel, and PowerPoint, see [Simple add-in commands sample](https://github.com/OfficeDev/Office-Add-in-Commands-Samples/tree/master/Simple).

For a sample manifest that implements add-in commands for Outlook, see [Sample manifest file for an Outlook add-in](https://github.com/OfficeDev/outlook-add-in-command-demo/blob/master/command-demo-manifest.xml).

# Specify Office hosts and API requirements

Your Office Add-in might depend on a specific Office host, a requirement set, an API member, or a version of the API in order to work as expected. For example, your add-in might:

* Run in a single Office application (Word or Excel), or several applications.
* Make use of JavaScript APIs that are only available in some versions of Office. For example, you might use the Excel JavaScript APIs in an add-in that runs in Excel 2016.
* Run only in versions of Office that support API members that your add-in uses.

This article helps you understand which options you should choose to ensure that your add-in works as expected and reaches the broadest audience possible.

**Note:** For a high-level view of where Office Add-ins are currently supported, see the [Office Add-in host and platform availability](http://dev.office.com/add-in-availability) page.

The following table lists core concepts discussed throughout this article.

| **Concept** | **Description** |
| --- | --- |
| Office application, Office host application, Office host, or host | The Office application used to run your add-in. For example, Word, Word Online, Excel, and so on. |
| Platform | Where the Office host runs, such as Office Online or Office for iPad. |
| Requirement set | A named group of related API members. Add-ins use requirement sets to determine whether the Office host supports API members used by your add-in. It's easier to test for the support of a requirement set than for the support of individual API members. Requirement set support varies by Office host and the version of the Office host.  Requirement sets are specified in the manifest file. When you specify requirement sets in the manifest, you set the minimum level of API support that the Office host must provide in order to run your add-in. Office hosts that don't support requirement sets specified in the manifest can't run your add-in, and your add-in won't display in My Add-ins. This restricts where your add-in is available.In code using runtime checks. For the complete list of requirement sets, see [Office add-in requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/office-add-in-requirement-sets.md). |
| Runtime check | A test that is performed at runtime to determine whether the Office host running your add-in supports requirement sets or methods used by your add-in. To perform a runtime check, you use an **if** statement with the **isSetSupported** method, the requirement sets, or the method names that aren't part of a requirement set.Use runtime checks to ensure that your add-in reaches the broadest number of customers. Unlike requirement sets, runtime checks don't specify the minimum level of API support that the Office host must provide for your add-in to run. Instead, you use the **if** statement to determine whether an API member is supported. If it is, you can provide additional functionality in your add-in. Your add-in will always display in **My Add-ins** when you use runtime checks. |

## Before you begin

Your add-in must use the most current version of the add-in manifest schema. If you use runtime checks in your add-in, ensure that you use the latest JavaScript API for Office (office.js) library.

### Specify the latest add-in manifest schema

Your add-in's manifest must use version 1.1 of the add-in manifest schema. Set the **OfficeApp** element in your add-in manifest as follows.

<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="TaskPaneApp">

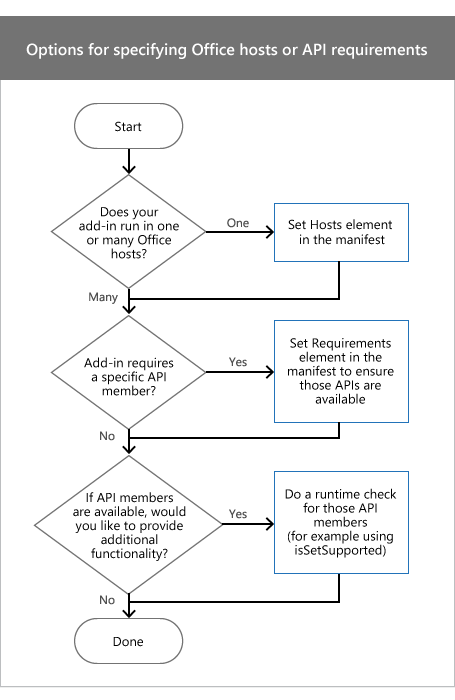
### Specify the latest JavaScript API for Office library

If you use runtime checks, reference the most current version of the JavaScript API for Office library from the content delivery network (CDN). To do this, add the following script tag to your HTML. Using /1/ in the CDN URL ensures that you reference the most recent version of Office.js.

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

## Options to specify Office hosts or API requirements

When you specify Office hosts or API requirements, there are several factors to consider. The following diagram shows how to decide which technique to use in your add-in.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/e3498f8f-7c7c-461c-84f3-b93910b088b9.png)

* If your add-in runs in one Office host, set the **Hosts** element in the manifest. For more information, see [Set the Hosts element](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md#set-the-hosts-element).
* To set the minimum requirement set or API members that an Office host must support to run your add-in, set the**Requirements** element in the manifest. For more information, see [Set the Requirements element in the manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md#set-the-requirements-element-in-the-manifest).
* If you would like to provide additional functionality if specific requirement sets or API members are available in the Office host, perform a runtime check in your add-in's JavaScript code. For example, if your add-in runs in Excel 2016, use API members from the new JavaScript API for Excel to provide additional functionality. For more information, see [Use runtime checks in your JavaScript code](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md#use-runtime-checks-in-your-javascript-code).

## Set the Hosts element

To make your add-in run in one Office host application, use the **Hosts** and **Host** elements in the manifest. If you don't specify the **Hosts** element, your add-in will run in all hosts.

For example, the following **Hosts** and **Host** declaration specifies that the add-in will work with any release of Excel, which includes Excel for Windows, Excel Online, and Excel for iPad.

<Hosts>

<Host Name="Workbook" />

</Hosts>

The **Hosts** element can contain one or more **Host** elements. The **Host** element specifies the Office host your add-in requires. The **Name** attribute is required and can be set to one of the following values.

| **Name** | **Office host applications** |
| --- | --- |
| Database | Access web apps |
| Document | Word for Windows, Mac, iPad and Online |
| Mailbox | Outlook for Windows, Mac, Web and Outlook.com |
| Presentation | PowerPoint for Windows, Mac, iPad and Online |
| Project | Project |
| Workbook | Excel Windows, Mac, iPad and Online |

**Note:** The Name attribute specifies the Office host application that can run your add-in. Office hosts are supported on different platforms and run on desktops, web browsers, tablets, and mobile devices. You can't specify which platform can be used to run your add-in. For example, if you specify Mailbox, both Outlook and Outlook Web App can be used to run your add-in.

## Set the Requirements element in the manifest

The **Requirements** element specifies the minimum requirement sets or API members that must be supported by the Office host to run your add-in. The **Requirements** element can specify both requirement sets and individual methods used in your add-in. In version 1.1 of the add-in manifest schema, the **Requirements** element is optional for all add-ins, except for Outlook add-ins.

**Caution:** Only use the **Requirements** element to specify critical requirement sets or API members that your add-in must use. If the Office host or platform doesn't support the requirement sets or API members specified in the **Requirements**element, the add-in won't run in that host or platform, and won't display in **My Add-ins**. Instead, we recommend that you make your add-in available on all platforms of an Office host, such as Excel for Windows, Excel Online, and Excel for iPad. To make your add-in available on all Office hosts and platforms, use runtime checks instead of the **Requirements**element.

The following code example shows an add-in that loads in all Office host applications that support the following:

* **TableBindings** requirement set, which has a minimum version of 1.1.
* **OOXML** requirement set, which has a minimum version of 1.1.
* **Document.getSelectedDataAsync** method.

<Requirements>

<Sets DefaultMinVersion="1.1">

<Set Name="TableBindings" MinVersion="1.1"/>

<Set Name="OOXML" MinVersion="1.1"/>

</Sets>

<Methods>

<Method Name="Document.getSelectedDataAsync"/>

</Methods>

</Requirements>

* The **Requirements** element contains the **Sets** and **Methods** child elements.
* The **Sets** element can contain one or more **Set** elements. **DefaultMinVersion** specifies the default **MinVersion** value of all child **Set** elements.
* The **Set** element specifies requirement sets that the Office host must support to run the add-in. The **Name** attribute specifies the name of the requirement set. The **MinVersion** specifies the minimum version of the requirement set. **MinVersion** overrides the value of **DefaultMinVersion**. For more information about requirement sets and requirement set versions that your API members belong to, see [Office add-in requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/office-add-in-requirement-sets.md).
* The **Methods** element can contain one or more **Method** elements. You can't use the **Methods** element with Outlook add-ins.
* The **Method** element specifies an individual method that must be supported in the Office host where your add-in runs. The **Name** attribute is required and specifies the name of the method qualified with its parent object.

## Use runtime checks in your JavaScript code

You might want to provide additional functionality in your add-in if certain requirement sets are supported by the Office host. For example, you might want to use the new Word JavaScript APIs Word in your existing add-in if your add-in runs in Word 2016. To do this, you use the **isSetSupported** method with the name of the requirement set. **isSetSupported** determines, at runtime, whether the Office host running the add-in supports the requirement set. If the requirement set is supported, **isSetSupported** returns **true** and runs the additional code that uses the API members from that requirement set. If the Office host doesn't support the requirement set, **isSetSupported** returns **false** and the additional code won't run. The following code shows the syntax to use with **isSetSupported**.

if (Office.context.requirements.isSetSupported(RequirementSetName , VersionNumber))

{

// Code that uses API members from RequirementSetName.

}

* RequirementSetName (required) is a string that represents the name of the requirement set. For more information about available requirement sets, see [Office add-in requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/office-add-in-requirement-sets.md).
* VersionNumber (optional) is the version of the requirement set.

In Excel 2016 or Word 2016, use **isSetSupported** with the **ExcelAPI** or **WordAPI** requirement sets. The **isSetSupported**method, and the **ExcelAPI** and **WordAPI** requirement sets, are available in the latest Office.js file available from the CDN. If you don't use Office.js from the CDN, your add-in might generate exceptions because **isSetSupported** will be undefined. For more information, see [Specify the latest JavaScript API for Office library](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md#specify-the-latest-javascript-api-for-office-library).

**Note:** **isSetSupported** does not work in Outlook or Outlook Web App. To use a runtime check in Outlook or Outlook Web App, use the technique described in [Runtime checks using methods not in a requirement set](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md#runtime-checks-using-methods-not-in-a-requirement-set).

The following code example shows how an add-in can provide different functionality for different Office hosts that might support different requirement sets or API members.

if (Office.context.requirements.isSetSupported('WordApi', 1.1))

{

// Run code that provides additional functionality using the JavaScript API for Word when the add-in runs in Word 2016.

}

else if (Office.context.requirements.isSetSupported('CustomXmlParts'))

{

// Run code that uses API members from the CustomXmlParts requirement set.

}

else

{

// Run additional code when the Office host is not Word 2016, and when the Office host does not support the CustomXmlParts requirement set.

}

## Runtime checks using methods not in a requirement set

Some API members don't belong to requirement sets. This only applies to API members that are part of the [JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/javascript-api-for-office.md) namespace (anything under Office.), not API members that belong to the Word JavaScript API (anything in Word.) or [Excel add-ins JavaScript API reference](https://msdn.microsoft.com/library/office/mt616490.aspx) (anything in Excel.) namespaces. When your add-in depends on a method that is not part of a requirement set, you can use the runtime check to determine whether the method is supported by the Office host, as shown in the following code example. For a complete list of methods that don't belong to a requirement set, see [Office add-in requirement sets](https://github.com/OfficeDev/office-js-docs/blob/master/reference/office-add-in-requirement-sets.md).

**Note** We recommend that you limit the use of this type of runtime check in your add-in's code.

The following code example checks whether the host supports **document.setSelectedDataAsync**.

if (Office.context.document.setSelectedDataAsync)

{

// Run code that uses document.setSelectedDataAsync.

}

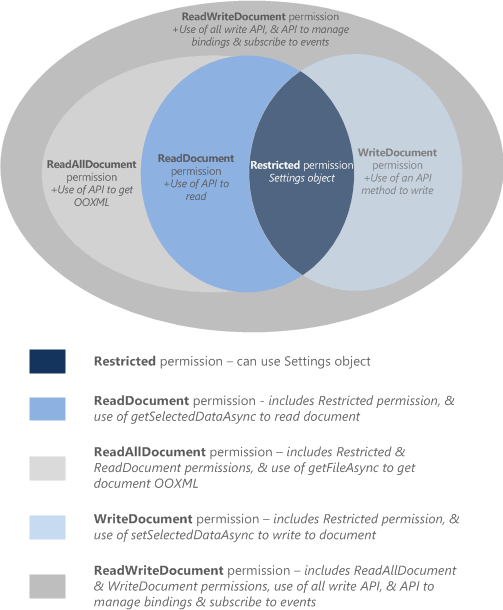
# Requesting permissions for API use in content and task pane add-ins

This article describes the different permission levels that you can declare in your content or task pane add-in's manifest to specify the level of JavaScript API access your add-in requires for its features.

## Permissions model

A five-level JavaScript API access-permissions model provides the basis for privacy and security for users of your content and task pane add-ins. Figure 1 shows the five levels of API permissions you can declare in your add-in's manifest.

**Figure 1. The five-level permission model for content and task pane add-ins**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15appsdk_TaskPaneAppPermission.gif)

These permissions specify the subset of the API that the add-in runtime will allow your content or task pane add-in to use when a user inserts, and then activates (trusts) your add-in. To declare the permission level your content or task pane add-in requires, specify one of the permission text values in the [Permissions](http://msdn.microsoft.com/en-us/library/d4cfe645-353d-8240-8495-f76fb36602fe%28Office.15%29.aspx) element of your add-in's manifest. The following example requests the **WriteDocument** permission, which will allow only methods that can write to (but not read) the document.

<Permissions>WriteDocument</Permissions>

As a best practice, you should request permissions based on the principle of least privilege. That is, you should request permission to access only the minimum subset of the API that your add-in requires to function correctly. For example, if your add-in needs only to read data in a user's document for its features, you should request no more than the **ReadDocument**permission.

The following table describes the subset of the JavaScript API that is enabled by each permission level.

| **Permission** | **Enabled subset of the API** |
| --- | --- |
| **Restricted** | The methods of the [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) object, and the [Document.getActiveViewAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getactiveviewasync.md) method.This is the minimum permission level that can be requested by a content or task pane add-in. |
| **ReadDocument** | In addition to the API allowed by the **Restricted** permission, adds access to the API members necessary to read the document and manage bindings.This includes the use of:   * The [Document.getSelectedDataAsync](http://msdn.microsoft.com/en-us/library/f85ad02c-64f0-4b73-87f6-7f521b3afd69(Office.15).aspx) method to get the selected text, HTML (Word only), or tabular data, but not the underlying Open Office XML (OOXML) code that contains all of the data in the document. * The [Document.getFileAsync](http://msdn.microsoft.com/en-us/library/78047418-89c4-4c7d-9427-4735b8559518(Office.15).aspx) method to get all of the text in the document, but not the underlying OOXML binary copy of the document. * The [Binding.getDataAsync](http://msdn.microsoft.com/en-us/library/5372ffd8-579d-4fcb-9e5b-e9a2128f3201(Office.15).aspx) method for reading bound data in the document. * The [addFromNamedItemAsync](http://msdn.microsoft.com/en-us/library/afbadac7-60c7-47cb-9477-6e9466ded44c(Office.15).aspx), [addFromPromptAsync](http://msdn.microsoft.com/en-us/library/9dc03608-b08b-4700-8be1-3c86ae236799(Office.15).aspx), [addFromSelectionAsync](http://msdn.microsoft.com/en-us/library/edc99214-e63e-43f2-9392-97ead42fc155(Office.15).aspx) methods of the Bindings object for creating bindings in the document. * The [getAllAsync](http://msdn.microsoft.com/en-us/library/ef902b73-cc4c-4551-95de-d8a51eeba82f(Office.15).aspx), [getByIdAsync](http://msdn.microsoft.com/en-us/library/2727c891-bc05-465c-9324-113fbfeb3fbb(Office.15).aspx), and [releaseByIdAsync](http://msdn.microsoft.com/en-us/library/ad285984-8b44-435d-9b84-f0ade570c896(Office.15).aspx) methods of the Bindings object for accessing and removing bindings in the document. * The [Document.getFilePropertiesAsync](http://msdn.microsoft.com/en-us/library/2533a563-95ae-4d52-b2d5-a6783e4ef5b4(Office.15).aspx) method to access document file properties, such as the URL of the document. * The [Document.goToByIdAsync](http://msdn.microsoft.com/en-us/library/35dda81c-235e-4eab-8a77-9acb3b73a380(Office.15).aspx) method to navigate to named objects and locations in the document. * For task pane add-ins for Project, all of the "get" methods of the [ProjectDocument](http://msdn.microsoft.com/en-us/library/1908af4f-93b9-4859-87e3-06942014fae1(Office.15).aspx) object. |
| **ReadAllDocument** | In addition to the API allowed by the **Restricted** and **ReadDocument** permissions, allows the following additional access to document data:   * The Document.getSelectedDataAsync and Document.getFileAsync methods can access the underlying OOXML code of the document (which in addition to the text may include formatting, links, embedded graphics, comments, revisions, and so forth). |
| **WriteDocument** | In addition to the API allowed by the **Restricted** permission, adds access to the following API members:   * The [Document.setSelectedDataAsync](http://msdn.microsoft.com/en-us/library/998f38dc-83bd-4659-a759-4758c632a6ef(Office.15).aspx) method to write to the user's selection in the document. |
| **ReadWriteDocument** | In addition to the API allowed by the **Restricted**, **ReadDocument**, **ReadAllDocument**, and**WriteDocument** permissions, includes access to all remaining API supported by content and task pane add-ins, including methods for subscribing to events.You must declare the**ReadWriteDocument** permission to access these additional API members:   * The [Binding.setDataAsync](http://msdn.microsoft.com/en-us/library/6a59bb6d-40b6-4a95-9b98-d70d4616de09(Office.15).aspx) method for writing to bound regions of the document. * The [TableBinding.addRowsAsync](http://msdn.microsoft.com/en-us/library/1cd23454-8435-4e13-98b3-d0d29ed278a8(Office.15).aspx) method for adding rows to bound tables. * The [TableBinding.addColumnsAsync](http://msdn.microsoft.com/en-us/library/8f1bfa81-3850-4ea1-ba2e-c9bcf5847a44(Office.15).aspx) method for adding columns to bound tables. * The [TableBinding.deleteAllDataValuesAsync](http://msdn.microsoft.com/en-us/library/8f5cc783-384d-4520-a218-190dfed74dd2(Office.15).aspx) method for deleting all data in a bound table. * The [setFormatsAsync](http://msdn.microsoft.com/en-us/library/49712906-f582-4055-9ef8-6edde6e97679(Office.15).aspx), [clearFormatsAsync](http://msdn.microsoft.com/en-us/library/cc56e9c0-b33c-4d9b-b676-a7e50f757c10(Office.15).aspx), and [setTableOptionsAsync](http://msdn.microsoft.com/en-us/library/2885fc57-4527-4ca4-a43d-9ee447ec27d3(Office.15).aspx) methods of the TableBinding object for setting formatting and options on bound tables. * All of the members of the [CustomXmlNode](http://msdn.microsoft.com/en-us/library/dc1518de-47fa-4108-aab7-04a022724b04(Office.15).aspx), [CustomXmlPart](http://msdn.microsoft.com/en-us/library/83f0e668-8236-4f2f-a20f-b173a9e3f65f(Office.15).aspx), [CustomXmlParts](http://msdn.microsoft.com/en-us/library/ba40cd4c-29bb-4f31-875d-6f1382fd1ee8(Office.15).aspx), and [CustomXmlPrefixMappings](http://msdn.microsoft.com/en-us/library/18b9aa8c-83e7-4c2f-8530-6a0ac8ce5535(Office.15).aspx) objects. * All of the methods for subscribing to the events supported by content and task pane add-ins, specifically the addHandlerAsync and removeHandlerAsync methods of the [Binding](http://msdn.microsoft.com/en-us/library/42882642-d22b-47d2-a8d3-3aa8c6a4435e(Office.15).aspx), [CustomXmlPart](http://msdn.microsoft.com/en-us/library/83f0e668-8236-4f2f-a20f-b173a9e3f65f(Office.15).aspx), [Document](http://msdn.microsoft.com/en-us/library/f8859516-cc1f-4b20-a8f3-cee37a983e70(Office.15).aspx), [ProjectDocument](http://msdn.microsoft.com/en-us/library/1908af4f-93b9-4859-87e3-06942014fae1(Office.15).aspx), and [Settings](http://msdn.microsoft.com/en-us/library/ad733387-a58c-4514-8fc2-53e64fad468d(Office.15).aspx) objects. |

# Update the version of your JavaScript API for Office and manifest schema files

This article describes how to update your JavaScript files (Office.js and app-specific .js files) and add-in manifest validation file in your Office Add-in project to version 1.1.

## Using the most up-to-date project files

If you use Visual Studio to develop your add-in, to use the [newest API members](https://github.com/OfficeDev/office-js-docs/blob/master/reference/what's-changed-in-the-javascript-api-for-office.md) of the JavaScript API for Office and the [v1.1 features of the add-in manifest](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-manifests.md) (which is validated against offappmanifest-1.1.xsd), you need to download and install the [Visual Studio 2015 and the latest Office Developer Tools](https://www.visualstudio.com/features/office-tools-vs).

If you use a text editor or IDE other than Visual Studio to develop your add-in, you need to update the references to the CDN for Office.js and the version of schema referenced in your add-in's manifest.

To run an add-in developed using new and updated Office.js API and add-in manifest features, your customers must be running Office 2013 SP1 or later version on-premises products, and where applicable, SharePoint Server 2013 SP1 and related server products, Exchange Server 2013 Service Pack 1 (SP1), or the equivalent online hosted products: Office 365, SharePoint Online, and Exchange Online.

To download Office, SharePoint, and Exchange SP1 products, see the following:

* [List of all Service Pack 1 (SP1) updates for Microsoft Office 2013 and related desktop products](http://support.microsoft.com/kb/2850036)
* [List of all Service Pack 1 (SP1) updates for Microsoft SharePoint Server 2013 and related server products](http://support.microsoft.com/kb/2850035)
* [Description of Exchange Server 2013 Service Pack 1](http://support.microsoft.com/kb/2926248)

## Updating an Office Add-in project created with Visual Studio to use the latest JavaScript API for Office library and version 1.1 add-in manifest schema

For projects created before the release of v1.1 of the JavaScript API for Office and add-in manifest schema, you can update a project's files using the **NuGet Package Manager**, and then update your add-in's HTML pages to reference them.

Note that the update process is applied on a per-project basis - you'll need to repeat the updating process for each add-in project in which you want to use v1.1 of Office.js and add-in manifest schema.

### To update the JavaScript API for Office library files in your project to the newest release

1. In Visual Studio 2015, open or create a new **Office Add-in** project.
   * In the left pane, choose **Update** and complete the package update process.

* Go to step 6.

1. Choose **Tools** > **NuGet Package Manager** > **Manage Nuget Packages for Solution**.
2. In the **NuGet Package Manager**, select **nuget.org** for **Package source** and **Upgrade available** for **Filter**. and select Microsoft.Office.js.
3. In the left pane, choose **Update** and complete the package update process.
4. In the **head** tag of your add-in's HTML pages, comment out or delete any existing office.js script references and reference the updated JavaScript API for Office library like as follows (:

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

Info: The /1/ in front of office.js in the CDN URL specifies to use the latest incremental release within version 1 of Office.js.

### To update the manifest file in your project to use schema version 1.1

In your Add-in's Manifest file, update the **xmlns** attribute of the **OfficeApp** element changing the version value to 1.1(leaving attributes other than the **xmlns** attribute unchanged).

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xsi:type="ContentApp"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://schemas.microsoft.com/office/appforoffice/1.1">

<!-- manifest contents -->

</OfficeApp>

**Note:** After updating the version of the add-in manifest schema to 1.1, you will need to remove the **Capabilities** and **Capability** elements, and replace them with either the [Hosts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/hosts.md) and [Host](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/hosts.md) elements or the [Requirements and Requirement elements](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md).

## Updating an Office Add-in project created with a text editor or other IDE to use the newest JavaScript API for Office library and version 1.1 add-in manifest schema

For projects created before the release of v1.1 of the JavaScript API for Office and add-in manifest schema, you need to update your add-in's HTML pages to reference CDN of the v1.1 library, and update your add-in's manifest file to use schema v1.1.

The update process is applied on a per-project basis - you'll need to repeat the updating process for each add-in project in which you want to use v1.1 of Office.js and add-in manifest schema.

You don't need local copies of the JavaScript API for Office files (Office.js and app-specific .js files) to develop anOffice Add-in (referencing the CDN for Office.js downloads the necessary files at runtime), but if you want a local copy of the library files you can use the [NuGet Command-Line Utility](http://docs.nuget.org/consume/installing-nuget) and the Install-Package Microsoft.Office.js command to download them.

**Note:** To get a copy of the XSD (XML Schema Definition) for the v1.1 add-in manifest, see the listing in [Schema reference for Office Add-ins manifests (v1.1)](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/add-in-manifests.md).

### To update the JavaScript API for Office library files in your project to use the newest release

1. Open the HTML pages for your add-in in your text editor or IDE.
2. In the **head** tag of your add-in's HTML pages, comment out or delete any existing office.js script references and reference the updated JavaScript API for Office library like as follows (:

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

Info: The /1/ in front of office.js in the CDN URL specifies to use the latest incremental release within version 1 of Office.js.

### To update the manifest file in your project to use schema version 1.1

In your Add-in's Manifest file, update the **xmlns** attribute of the **OfficeApp** element changing the version value to 1.1(leaving attributes other than the **xmlns** attribute unchanged).

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xsi:type="ContentApp"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://schemas.microsoft.com/office/appforoffice/1.1">

<!-- manifest contents -->

</OfficeApp>

**Note:** After updating the version of the add-in manifest schema to 1.1, you will need to remove the **Capabilities** and **Capability** elements, and replace them with either the [Hosts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/hosts.md) and [Host](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/hosts.md) elements or the [Requirements and Requirement elements](https://github.com/OfficeDev/office-js-docs/blob/master/docs/overview/specify-office-hosts-and-api-requirements.md).

# Understanding the JavaScript API for Office

This article provides information about the JavaScript API for Office and how to use it. For reference information, see [JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/javascript-api-for-office.md). For information about updating Visual Studio project files to the most current version of the JavaScript API for Office, see [Update the version of your JavaScript API for Office and manifest schema files](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/update-your-javascript-api-for-office-and-manifest-schema-version.md).

**Note:** When you build your add-in, if you plan to [publish](https://github.com/OfficeDev/office-js-docs/blob/master/docs/publish/publish.md) your add-in to the Office Store, make sure that you conform to the [Office Store validation policies](https://msdn.microsoft.com/en-us/library/jj220035.aspx). For example, to pass validation, your add-in must work across all platforms that support the methods that you define (for more information, see [section 4.12](https://msdn.microsoft.com/en-us/library/jj220035.aspx#Anchor_3) and the [Office Add-in host and availability page](https://dev.office.com/add-in-availability)).

## Referencing the JavaScript API for Office library in your add-in

The [JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/javascript-api-for-office.md) library consists of the Office.js file and associated host application-specific .js files, such as Excel-15.js and Outlook-15.js. The simplest method of referencing the API is using our CDN by adding the following <script> to your page's <head> tag:

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

This will download and cache the JavaScript API for Office files the first time your add-in loads to make sure that it is using the most up-to-date implementation of Office.js and its associated files for the specified version.

For more details around the Office.js CDN, including how versioning and backward compatability is handled, see [Referencing the JavaScript API for Office library from its content delivery network (CDN)](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/referencing-the-javascript-api-for-office-library-from-its-cdn.md).

## Initializing your add-in

**Applies to:** All add-in types

Office.js provides an initialization event which gets fired when the API is fully loaded and ready to begin interacting with the user. You can use the **initialize** event handler to implement common add-in initialization scenarios, such as prompting the user to select some cells in Excel, and then inserting a chart initialized with those selected values. You can also use the initialize event handler to initialize other custom logic for your add-in, such as establishing bindings, prompting for default add-in settings values, and so on.

At a minimum, the initialize event would look like the follow example:

Office.initialize = function () { };

If you are using additional JavaScript frameworks that include their own initialization handler or tests, these should be placed within the Office.initialize event. For example, [JQuery's](https://jquery.com/) $(document).ready() function would be referenced as follows:

Office.initialize = function () {

// Office is ready

$(document).ready(function () {

// The document is ready

});

};

All pages within an Office Add-ins are required to assign an event handler to the initialize event, **Office.initialize**. If you fail to assign an event handler, your add-in may raise an error when it starts. Also, if a user attempts to use your add-in with an Office Online web client, such as Excel Online, PowerPoint Online, or Outlook Web App, it will fail to run. If you don't need any initialization code, then the body of the function you assign to **Office.initialize** can be empty, as it is in the first example above.

For more detail about the sequence of events when an add-in is initialized, see [Loading the DOM and runtime environment](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/loading-the-dom-and-runtime-environment.md).

#### Initialization Reason

For task pane and content add-ins, Office.initialize provides an additional reason parameter. This parameter can be used to determine how an add-in was added to the current document. You can use this to provide different logic for when an add-in is first inserted versus when it already existed within the document.

Office.initialize = function (reason) {

$(document).ready(function () {

switch (reason) {

case 'inserted': console.log('The add-in was just inserted.');

case 'documentOpened': console.log('The add-in is already part of the document.');

}

}

For more information, see [Office.initialize Event](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.initialize.md) and [InitializationReason Enumeration](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/initializationreason-enumeration.md)

## Context Object

**Applies to:** All add-in types

When an add-in is initialized, it has many different objects that it can interact with in the runtime environment. The add-in's runtime context is reflected in the API by the [Context](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.md) object. The **Context** is the main object that provides access to the most important objects of the API, such as the [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) and [Mailbox](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.md) objects, which in turn provide access to document and mailbox content.

For example, in task pane or content add-ins, you can use the [document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.document.md) property of the **Context** object to access the properties and methods of the **Document** object to interact with the content of Word documents, Excel worksheets, or Project schedules. Similarly, in Outlook add-ins, you can use the [mailbox](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.md) property of the **Context** object to access the properties and methods of the **Mailbox** object to interact with the message, meeting request, or appointment content.

The **Context** object also provides access to the [contentLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.contentlanguage.md) and [displayLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.displaylanguage.md) properties that let you determine the locale (language) used in the document or item, or by the host application. And, the [roamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.md) property that lets you access the members of the [RoamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) object. Finally, the **Context** object provides a [ui](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/officeui.md) property that enables your add-in to launch pop-up dialogs.

## Document object

**Applies to:** Content and task pane add-in types

To interact with document data in Excel, PowerPoint, and Word, the API provides the [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) object. You can use **Document** object members to access data from the following ways:

* Read and write to active selections in the form of text, contiguous cells (matrices), or tables.
* Tabular data (matrices or tables).
* Bindings (created with the "add" methods of the **Bindings** object).
* Custom XML parts (only for Word).
* Settings or add-in state persisted per add-in on the document.

You can also use the **Document** object to interact with data in Project documents. The Project-specific functionality of the API is documented in the members [ProjectDocument](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/projectdocument.projectdocument.md) abstract class. For more information about creating task pane add-ins for Project, see [Task pane add-ins for Project](https://github.com/OfficeDev/office-js-docs/blob/master/docs/project/project-add-ins.md).

All these forms of data access start from an instance of the abstract **Document** object.

You can access an instance of the **Document** object when the task pane or content add-in is initialized by using the [document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.document.md)property of the **Context** object. The **Document** object defines common data access functions shared across Word and Excel documents, and also provides access to the **CustomXmlParts** object for Word documents.

The **Document** object supports four ways for developers to access document contents:

* Selection-based access
* Binding-based access
* Custom XML part-based access (Word only)
* Entire document-based access (PowerPoint and Word only)

To help you understand how selection- and binding-based data access methods work, we will first explain how the data-access APIs provide consistent data access across different Office applications.

### Consistent data access across Office applications

**Applies to:** Content and task pane add-in types

To create extensions that seamlessly work across different Office documents, the JavaScript API for Office abstracts away the particularities of each Office application through common data types and the ability to coerce different document contents into three common data types.

#### Common data types

In both selection-based and binding-based data access, document contents are exposed through data types that are common across all the supported Office applications. In Office 2013, three main data types are supported:

| **Data type** | **Description** | **Host application support** |
| --- | --- | --- |
| Text | Provides a string representation of the data in the selection or binding. | In Excel 2013, Project 2013, and PowerPoint 2013 only plain text is supported. In Word 2013, three text formats are supported: plain text, HTML, and Office Open XML (OOXML).When text is selected in a cell in Excel, selection-based methods read and write to the entire contents of the cell, even if only a portion of the text is selected in the cell. When text is selected in Word and PowerPoint, selection-based methods read and write only to the run of characters that are selected.Project 2013 and PowerPoint 2013 support only selection-based data access. |
| Matrix | Provides the data in the selection or binding as a two dimensional **Array**, which in JavaScript is implemented as an array of arrays.For example, two rows of **string** values in two columns would be [['a', 'b'], ['c', 'd']], and a single column of three rows would be [['a'], ['b'], ['c']]. | Matrix data access is supported only in Excel 2013 and Word 2013. |
| Table | Provides the data in the selection or binding as a [TableData](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/tabledata.md) object. The **TableData** object exposes the data through the **headers** and **rows**properties. | Table data access is supported only in Excel 2013 and Word 2013. |

#### Data type coercion

The data access methods on the **Document** and [Binding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.md) objects support specifying the desired data type using the coercionType parameter of these methods, and corresponding [CoercionType](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/coerciontype-enumeration.md) enumeration values. Regardless of the actual shape of the binding, the different Office applications support the common data types by trying to coerce the data into the requested data type. For example, if a Word table or paragraph is selected, the developer can specify to read it as plain text, HTML, Office Open XML, or a table, and the API implementation handles the necessary transformations and data conversions.

**Tip** **When should you use the matrix versus table coercionType for data access?** If you need your tabular data to grow dynamically when rows and columns are added, and you must work with table headers, you should use the table data type (by specifying the coercionType parameter of a **Document** or **Binding** object data access method as "table" or **Office.CoercionType.Table**). Adding rows and columns within the data structure is supported in both table and matrix data, but appending rows and columns is supported only for table data. If you aren't planning on adding rows and columns, and your data doesn't require header functionality, then you should use the matrix data type (by specifying thecoercionType parameter of the data access method as "matrix" or **Office.CoercionType.Matrix**), which provides a simpler model of interacting with the data.

If the data can't be coerced to the specified type, the [AsyncResult.status](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.error.md) property in the callback returns "failed", and you can use the [AsyncResult.error](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.context.md) property to access an [Error](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/error.md) object with information about why the method call failed.

## Working with selections using the Document object

The **Document** object exposes methods that let you to read and write to the user's current selection in a "get and set" fashion. To do that, the **Document** object provides the **getSelectedDataAsync** and **setSelectedDataAsync** methods.

For code examples that demonstrate how to perform tasks with selections, see [Read and write data to the active selection in a document or spreadsheet](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/read-and-write-data-to-the-active-selection-in-a-document-or-spreadsheet.md).

## Working with bindings using the Bindings and Binding objects

Binding-based data access enables content and task pane add-ins to consistently access a particular region of a document or spreadsheet through an identifier associated with a binding. The add-in first needs to establish the binding by calling one of the methods that associates a portion of the document with a unique identifier: [addFromPromptAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfrompromptasync.md), [addFromSelectionAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromselectionasync.md), or [addFromNamedItemAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromnameditemasync.md). After the binding is established, the add-in can use the provided identifier to access the data contained in the associated region of the document or spreadsheet. Creating bindings provides the following value to your add-in:

* Permits access to common data structures across supported Office applications, such as: tables, ranges, or text (a contiguous run of characters).
* Enables read/write operations without requiring the user to make a selection.
* Establishes a relationship between the add-in and the data in the document. Bindings are persisted in the document, and can be accessed at a later time.

Establishing a binding also allows you to subscribe to data and selection change events that are scoped to that particular region of the document or spreadsheet. This means that the add-in is only notified of changes that happen within the bound region as opposed to general changes across the whole document or spreadsheet.

The [Bindings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.bindings.md) object exposes a [getAllAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.getallasync.md) method that gives access to the set of all bindings established on the document or spreadsheet. An individual binding can be accessed by its ID using either the [Bindings.getBindingByIdAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.getbyidasync.md) or [Office.select](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.select.md)methods. You can establish new bindings as well as remove existing ones by using one of the following methods of the **Bindings** object: [addFromSelectionAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromselectionasync.md), [addFromPromptAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfrompromptasync.md), [addFromNamedItemAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromnameditemasync.md), or [releaseByIdAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.releasebyidasync.md).

There are three different types of bindings that you specify with the bindingType parameter when you create a binding with the **addFromSelectionAsync**, **addFromPromptAsync** or **addFromNamedItemAsync** methods:

| **Binding type** | **Description** | **Host application support** |
| --- | --- | --- |
| Text binding | Binds to a region of the document that can be represented as text. | In Word, most contiguous selections are valid, while in Excel only single cell selections can be the target of a text binding. In Excel, only plain text is supported. In Word, three formats are supported: plain text, HTML, and Open XML for Office. |
| Matrix binding | Binds to a fixed region of a document that contains tabular data without headers.Data in a matrix binding is written or read as a two dimensional **Array**, which in JavaScript is implemented as an array of arrays. For example, two rows of **string** values in two columns can be written or read as [['a', 'b'], ['c', 'd']], and a single column of three rows can be written or read as [['a'], ['b'], ['c']]. | In Excel, any contiguous selection of cells can be used to establish a matrix binding. In Word, only tables support matrix binding. |
| Table binding | Binds to a region of a document that contains a table with headers.Data in a table binding is written or read as a [TableData](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/tabledata.md)object. The **TableData** object exposes the data through the **headers** and **rows**properties. | Any Excel or Word table can be the basis for a table binding. After you establish a table binding, each new row or column a user adds to the table is automatically included in the binding. |
| After a binding is created by using one of the three "add" methods of the **Bindings**object, you can work with the binding's data and properties by using the methods of the corresponding object: [MatrixBinding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.matrixbinding.md), [TableBinding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.tablebinding.md), or [TextBinding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.textbinding.md). All three of these objects inherit the [getDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.getdataasync.md) and [setDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.setdataasync.md) methods of the **Binding**object that enable to you interact with the bound data. |  |  |

For code examples that demonstrate how to perform tasks with bindings, see [Bind to regions in a document or spreadsheet](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/bind-to-regions-in-a-document-or-spreadsheet.md).

## Working with custom XML parts using the CustomXmlParts and CustomXmlPart objects

**Applies to:** Task pane add-ins for Word

The [CustomXmlParts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.customxmlparts.md) and [CustomXmlPart](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlpart.customxmlpart.md) objects of the API provide access to custom XML parts in Word documents, which enable XML-driven manipulation of the contents of the document. For demonstrations of working with the **CustomXmlParts**and **CustomXmlPart** objects, see the [Word-Add-in-Work-with-custom-XML-parts](https://github.com/OfficeDev/Word-Add-in-Work-with-custom-XML-parts) code sample.

## Working with the entire document using the getFileAsync method

**Applies to:** Task pane add-ins for Word and PowerPoint

The [Document.getFileAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getfileasync.md) method and members of the [File](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/file.md) and [Slice](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/slice.md) objects to provide functionality for getting entire Word and PowerPoint document files in slices (chunks) of up to 4 MB at a time. For more information, see [How to: Get all file content from a document in an add-in](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/get-the-whole-document-from-an-add-in-for-powerpoint-or-word.md).

## Mailbox object

**Applies to:** Outlook add-ins

Outlook add-ins primarily use a subset of the API exposed through the [Mailbox](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.md) object. To access the objects and members specifically for use in Outlook add-ins, such as the [Item](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.item.md) object, you use the [mailbox](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.md) property of the **Context** object to access the **Mailbox** object, as shown in the following line of code.

// Access the Item object.

var item = Office.context.mailbox.item;

Additionally, Outlook add-ins can use the following objects:

* **Office** object: for initialization.
* **Context** object: for access to content and display language properties.
* **RoamingSettings** object: for saving Outlook add-in-specific custom settings to the user's mailbox where the add-in is installed.

For information about using JavaScript in Outlook add-ins, see [Outlook add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/outlook-add-ins.md) and [Overview of Outlook add-ins architecture and features](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/overview.md).

## API support matrix

This table summarizes the API and features supported across add-in types (content, task pane, and Outlook) and the Office applications that can host them when you specify the [Office host applications your add-in supports](http://msdn.microsoft.com/library/cff9fbdf-a530-4f6e-91ca-81bcacd90dcd%28Office.15%29.aspx) using the [1.1 add-in manifest schema and features supported by v1.1 JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/update-your-javascript-api-for-office-and-manifest-schema-version.md).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Host name** | Database | Workbook | Mailbox | Presentation | Document | Project |
|  | **Supported** **Host applications** | Access web apps | ExcelExcel Online | OutlookOutlook Web AppOWA for Devices | PowerPointPowerPoint Online | Word | Project |
| **Supported add-in types** | Content | Y | Y |  | Y |  |  |
|  | Task pane |  | Y |  | Y | Y | Y |
|  | Outlook |  |  | Y |  |  |  |
| **Supported API features** | Read/Write Text |  | Y |  | Y | Y | Y (Read only) |
|  | Read/Write Matrix |  | Y |  |  | Y |  |
|  | Read/Write Table |  | Y |  |  | Y |  |
|  | Read/Write HTML |  |  |  |  | Y |  |
|  | Read/WriteOffice Open XML |  |  |  |  | Y |  |
|  | Read task, resource, view, and field properties |  |  |  |  |  | Y |
|  | Selection changed events |  | Y |  |  | Y |  |
|  | Get whole document |  |  |  | Y | Y |  |
|  | Bindingsand binding events | Y (Only full and partialtable bindings) | Y |  |  | Y |  |
|  | Read/WriteCustom Xml Parts |  |  |  |  | Y |  |
|  | Persist add-in state data(settings) | Y (Per host add-in) | Y (Per document) | Y (Per mailbox) | Y (Per document) | Y (Per document) |  |
|  | Settings changed events | Y | Y |  | Y | Y |  |
|  | Get active view modeand view changed events |  |  |  | Y |  |  |
|  | Navigate to locationsin the document |  | Y |  | Y | Y |  |
|  | Activate contextuallyusing rules and RegEx |  |  | Y |  |  |  |
|  | Read Item properties |  |  | Y |  |  |  |
|  | Read User profile |  |  | Y |  |  |  |
|  | Get attachments |  |  | Y |  |  |  |
|  | Get User identity token |  |  | Y |  |  |  |
|  | Call Exchange Web Services |  |  | Y |  |  |  |

# Office JavaScript API support for content and task pane add-ins in Office 2013

You can use the [Office JavaScript API](https://github.com/OfficeDev/office-js-docs/blob/master/reference/javascript-api-for-office.md) to create task pane or content add-ins for Office 2013 host applications. The objects and methods that content and task pane add-ins support are categorized as follows:

1. **Common objects shared with other Office Add-ins.** These objects include [Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.md), [Context](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.md), and [AsyncResult](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.md). The **Office**object is the root object of the Office JavaScript API. The **Context** object represents the add-in's runtime environment. Both **Office** and **Context** are the fundamental objects for any Office Add-in. The **AsyncResult** object represents the results of an asynchronous operation, such as the data returned to the **getSelectedDataAsync** method, which reads what a user has selected in a document.
2. **The Document object.** The majority of the API available to content and task pane add-ins is exposed through the methods, properties, and events of the [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) object. A content or task pane add-in can use the [Office.context.document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.document.md) property to access the **Document** object, and through it, can access the key members of the API for working with data in documents, such as the [Bindings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.bindings.md) and [CustomXmlParts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.customxmlparts.md) objects, and the [getSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getselecteddataasync.md), [setSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.setselecteddataasync.md), and [getFileAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getfileasync.md) methods. The **Document** object also provides the [mode](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.mode.md) property for determining whether a document is read-only or in edit mode, the [url](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.url.md) property to get the URL of the current document, and access to the [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) object. The **Document** object also supports adding event handlers for the [SelectionChanged](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.selectionchanged.event.md)event, so you can detect when a user changes his or her selection in the document.

A content or task pane add-in can access the **Document** object only after the DOM and runtime environment has been loaded, typically in the event handler for the [Office.initialize](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.initialize.md) event. For information about the flow of events when an add-in is initialized, and how to check that the DOM and runtime and loaded successfully, see [Loading the DOM and runtime environment](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/loading-the-dom-and-runtime-environment.md).

1. **Objects for working with specific features.** To work with specific features of the API, use the following objects and methods:
   * The methods of the [Bindings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.bindings.md) object to create or get bindings, and the methods and properties of the [Binding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.md) object to work with data.
   * The [CustomXmlParts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.customxmlparts.md), [CustomXmlPart](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlpart.customxmlpart.md) and associated objects to create and manipulate custom XML parts in Word documents.
   * The [File](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/file.md) and [Slice](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/slice.md) objects to create a copy of the entire document, break it into chunks or "slices", and then read or transmit the data in those slices.
   * The [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) object to save custom data, such as user preferences, and add-in state.

**Important** Some of the API members aren't supported across all Office applications that can host content and task pane add-ins. To determine which members are supported, see any of the following:

For a summary of Office JavaScript API support across Office host applications, see [Understanding the JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/understanding-the-javascript-api-for-office.md).

## Reading and writing to an active selection

You can read or write to the user's current selection in a document, spreadsheet, or presentation. Depending on the host application for your add-in, you can specify the type of data structure to read or write as a parameter in the [getSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getselecteddataasync.md) and [setSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.setselecteddataasync.md) methods of the [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) object. For example, you can specify any type of data (text, HTML, tabular data, or Office Open XML) for Word, text and tabular data for Excel, and text for PowerPoint and Project. You can also create event handlers to detect changes to the user's selection. The following example gets data from the selection as text using the **getSelectedDataAsync** method.

Office.context.document.getSelectedDataAsync(

Office.CoercionType.Text, function (asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

write('Action failed. Error: ' + asyncResult.error.message);

}

else {

write('Selected data: ' + asyncResult.value);

}

});

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

For more details and examples, see [Read and write data to the active selection in a document or spreadsheet](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/read-and-write-data-to-the-active-selection-in-a-document-or-spreadsheet.md).

## Binding to a region in a document or spreadsheet

You can use the **getSelectedDataAsync** and **setSelectedDataAsync** methods to read or write to the user's current selection in a document, spreadsheet, or presentation. However, if you would like to access the same region in a document across sessions of running your add-in without requiring the user to make a selection, you should first bind to that region. You can also subscribe to data and selection change events for that bound region.

You can add a binding by using [addFromNamedItemAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromnameditemasync.md), [addFromPromptAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfrompromptasync.md), or [addFromSelectionAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.addfromselectionasync.md) methods of the [Bindings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.bindings.md) object. These methods return an identifier that you can use to access data in the binding, or to subscribe to its data change or selection change events.

The following is an example that adds a binding to the currently selected text in a document, by using the**Bindings.addFromSelectionAsync** method.

Office.context.document.bindings.addFromSelectionAsync(

Office.BindingType.Text, { id: 'myBinding' }, function (asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

write('Action failed. Error: ' + asyncResult.error.message);

} else {

write('Added new binding with type: ' +

asyncResult.value.type + ' and id: ' + asyncResult.value.id);

}

});

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

For more details and examples, see [Bind to regions in a document or spreadsheet](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/bind-to-regions-in-a-document-or-spreadsheet.md).

## Getting entire documents

If your task pane add-in runs in PowerPoint or Word, you can use the [Document.getFileAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getfileasync.md), [File.getSliceAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/file.getsliceasync.md), and [File.closeAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/file.closeasync.md) methods to get an entire presentation or document.

When you call **Document.getFileAsync**, you get a copy of the document in a [File](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/file.md) object. The **File** object provides access to the document in "chunks" represented as [Slice](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) objects. When you call **getFileAsync**, you can specify the file type (text or compressed Open Office XML format), and size of the slices (up to 4MB). To access the contents of the **File** object, you then call **File.getSliceAsync** which returns the raw data in the [Slice.data](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/slice.data.md) property. If you specified compressed format, you will get the file data as a byte array. If you are transmitting the file to a web service, you can transform the compressed raw data to a base64-encoded string before submission. Finally, when you are finished getting slices of the file, use the **File.closeAsync**method to close the document.

For more details, see how to [get the whole document from an add-in for PowerPoint or Word](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/get-the-whole-document-from-an-add-in-for-powerpoint-or-word.md).

## Reading and writing custom XML parts of a Word document

Using the Open Office XML file format and content controls, you can add custom XML parts to a Word document and bind elements in the XML parts to content controls in that document. When you open the document, Word reads and automatically populates bound content controls with data from the custom XML parts. Users can also write data into the content controls, and when the user saves the document, the data in the controls will be saved to the bound XML parts. Task pane add-ins for Word, can use the [Document.customXmlParts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.customxmlparts.md) property,[CustomXmlParts](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.customxmlparts.md), [CustomXmlPart](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlpart.customxmlpart.md), and [CustomXmlNode](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlnode.customxmlnode.md) objects to read and write data dynamically to the document.

Custom XML parts may be associated with namespaces. To get data from custom XML parts in a namespace, use the [CustomXmlParts.getByNamespaceAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.getbynamespaceasync.md) method.

You can also use the [CustomXmlParts.getByIdAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.getbyidasync.md) method to access custom XML parts by their GUIDs. After getting a custom XML part, use the [CustomXmlPart.getXmlAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlpart.getxmlasync.md) method to get the XML data.

To add a new custom XML part to a document, use the **Document.customXmlParts** property to get the custom XML parts that are in the document, and call the [CustomXmlParts.addAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlparts.addasync.md) method.

For detailed information about how to work with custom XML parts with a task pane add-in, see [Creating Better Add-ins for Word with Office Open XML](https://github.com/OfficeDev/office-js-docs/blob/master/docs/word/create-better-add-ins-for-word-with-office-open-xml.md).

## Persisting add-in settings

Often you need to save custom data for your add-in, such as a user's preferences or the add-in's state, and access that data the next time the add-in is opened. You can use common web programming techniques to save that data, such as browser cookies or HTML 5 web storage. Alternatively, if your add-in runs in Excel, PowerPoint, or Word, you can use the methods of the [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) object. Data created with the **Settings** object is stored in the spreadsheet, presentation, or document that the add-in was inserted into and saved with. This data is available to only the add-in that created it.

To avoid roundtrips to the server where the document is stored, data created with the **Settings** object is managed in memory at run time. Previously saved settings data is loaded into memory when the add-in is initialized, and changes to that data are only saved back to the document when you call the [Settings.saveAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.saveasync.md) method. Internally, the data is stored in a serialized JSON object as name/value pairs. You use the [get](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.get.md), [set](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.set.md), and [remove](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.removehandlerasync.md) methods of the **Settings** object, to read, write, and delete items from the in-memory copy of the data. The following line of code shows how to create a setting named themeColor and set its value to 'green'.

Office.context.document.settings.set('themeColor', 'green');

Because settings data created or deleted with the **set** and **remove** methods is acting on an in-memory copy of the data, you must call **saveAsync** to persist changes to settings data into the document your add-in is working with.

For more details about working with custom data using the methods of the **Settings** object, see [Persisting add-in state and settings](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/persisting-add-in-state-and-settings.md).

## Reading properties of a project document

If your task pane add-in runs in Project, your add-in can read data from some of the project fields, resource, and task fields in the active project. To do that, you use the methods and events of the [ProjectDocument](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/projectdocument.projectdocument.md) object, which extends the **Document**object to provide additional Project-specific functionality.

For examples of reading Project data, see [Create your first task pane add-in for Project 2013 by using a text editor](https://github.com/OfficeDev/office-js-docs/blob/master/docs/project/create-your-first-task-pane-add-in-for-project-by-using-a-text-editor.md).

## Permissions model and governance

Your add-in uses the **Permissions** element in its manifest to request permission to access the level of functionality it requires from the Office JavaScript API. For example, if your add-in requires read/write access to the document, its manifest must specify ReadWriteDocument as the text value in its **Permissions** element. Because permissions exist to protect a user's privacy and security, as a best practice you should request the minimum level of permissions it needs for its features. The following example shows how to request the **ReadDocument** permission in a task pane's manifest.

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:type="TaskPaneApp">

???<!-- Other manifest elements omitted. -->

<Permissions>ReadDocument</Permissions>

???

</OfficeApp>

For more information, see [Requesting permissions for API use in content and task pane add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/requesting-permissions-for-api-use-in-content-and-task-pane-add-ins.md).

# Referencing the JavaScript API for Office library from its content delivery network (CDN)

The [JavaScript API for Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/javascript-api-for-office.md) library consists of the Office.js file and associated host application-specific .js files, such as Excel-15.js and Outlook-15.js.

The simplest way to reference the API is to use our CDN by adding the following <script> to your page's <head> tag:

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

The /1/ in front of office.js in the CDN URL specifies the latest incremental release within version 1 of Office.js. Because the JavaScript API for Office maintains backward compatibility, the latest release will continue to support API members that were introduced earlier in version 1. If you need to upgrade an existing project, see [Update the version of your JavaScript API for Office and manifest schema files](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/update-your-javascript-api-for-office-and-manifest-schema-version.md).

If you plan to publish your Office Add-in from the Office Store, you must use this CDN reference. Local references are only appropriate for internal, development, and debugging scenarios.

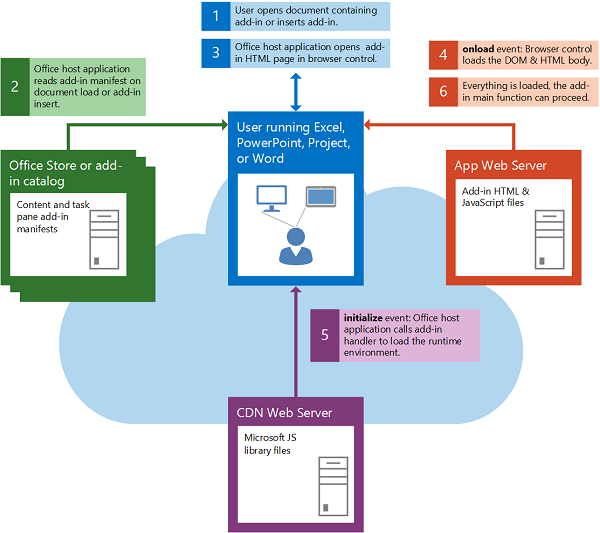
**Important:** When you develop an add-in for any Office host application, reference the JavaScript API for Office from inside the <head> section of the page. This ensures that the API is fully initialized prior to any body elements. Office hosts require that add-ins initialize within 5 seconds of activation. If your add-in doesn't activate within this threshold, it will be declared unresponsive and an error message will be displayed to the user.

# Loading the DOM and runtime environment

An add-in must ensure that both the DOM and the Office Add-ins runtime environment are loaded before running its own custom logic.

## Startup of a content or task pane add-in

The following figure shows the flow of events involved in starting a content or task pane add-in in Excel, PowerPoint, Project, Word, or Access.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15appsdk_LoadingDOMAgaveRuntime.png)

The following events occur when a content or task pane add-in starts:

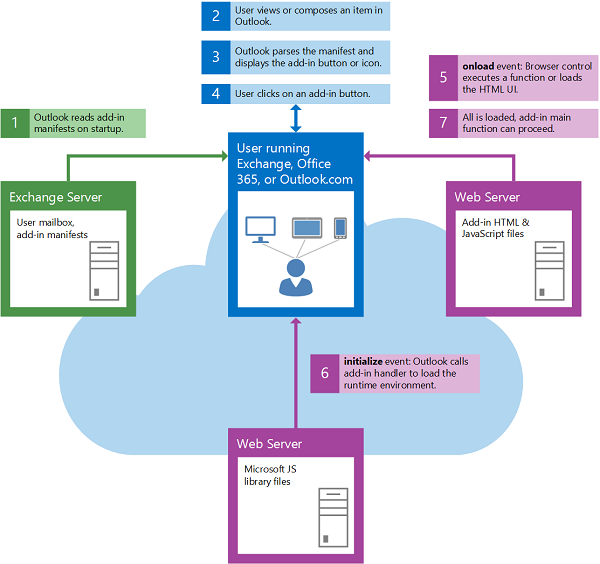
1. The user opens a document that already contains an add-in or inserts an add-in in the document.
2. The Office host application reads the add-in's XML manifest from the Office Store, add-in catalog on SharePoint, or shared folder catalog it originates from.
3. The Office host application opens the add-in's HTML page in a browser control.

The next two steps, steps 4 and 5, occur asynchronously and in parallel. For this reason, your add-in's code must make sure that both the DOM and the add-in runtime environment have finished loading before proceeding.

1. The browser control loads the DOM and HTML body, and calls the event handler for the **window.onload** event.
2. The Office host application loads the runtime environment, which downloads and caches the JavaScript API for JavaScript library files from the content distribution network (CDN) server, and then calls the add-in's event handler for the [initialize](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.initialize.md)event of the [Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.md) object.
3. When the DOM and HTML body finish loading and the add-in finishes initializing, the main function of the add-in can proceed.

## Startup of an Outlook add-in

The following figure shows the flow of events involved in starting an Outlook add-in running on the desktop, tablet, or smartphone.

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/olowawecon15_LoadingDOMAgaveRuntime.png)

The following events occur when an Outlook add-in starts:

1. When Outlook starts, Outlook reads the XML manifests for Outlook add-ins that have been installed for the user's email account.
2. The user selects an item in Outlook.
3. If the selected item satisfies the activation conditions of an Outlook add-in, Outlook activates the add-in and makes its button visible in the UI.
4. If the user clicks the button to start the Outlook add-in, Outlook opens the HTML page in a browser control. The next two steps, steps 5 and 6, occur in parallel.
5. The browser control loads the DOM and HTML body, and calls the event handler for the **onload** event.
6. Outlook calls the event handler for the [initialize](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.initialize.md) event of the [Office](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.md) object of the add-in.
7. When the DOM and HTML body finish loading and the add-in finishes initializing, the main function of the add-in can proceed.

## Checking the load status

One way to check that both the DOM and the runtime environment have finished loading is to use the jQuery [.ready()](http://api.jquery.com/ready/)function: $(document).ready(). For example, the following **initialize** event handler function makes sure the DOM is first loaded before the code specific to initializing the add-in runs. Subsequently, the **initialize** event handler proceeds to use the [mailbox.item](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.item.md) property to obtain the currently selected item in Outlook, and calls the main function of the add-in, initDialer.

Office.initialize = function () {

// Checks for the DOM to load.

$(document).ready(function () {

// After the DOM is loaded, add-in-specific code can run.

var mailbox = Office.context.mailbox;

\_Item = mailbox.item;

initDialer();

});

}

This same technique can be used in the **initialize** handler of any Office Add-in.

The phone dialer sample Outlook add-in shows a slightly different approach using only JavaScript to check these same conditions.

**Important:** Even if your add-in has no initialization tasks to perform, you must include at least a minimal **Office.initialize**event handler function like the following example.

Office.initialize = function () {

};

If you fail to include an **Office.initialize** event handler, your add-in may raise an error when it starts. Also, if a user attempts to use your add-in with an Office Online web client, such as Excel Online, PowerPoint Online, or Outlook Web App, it will fail to run.

If your add-in includes more than one page, whenever it loads a new page that page must include or call an **Office.initialize**event handler.

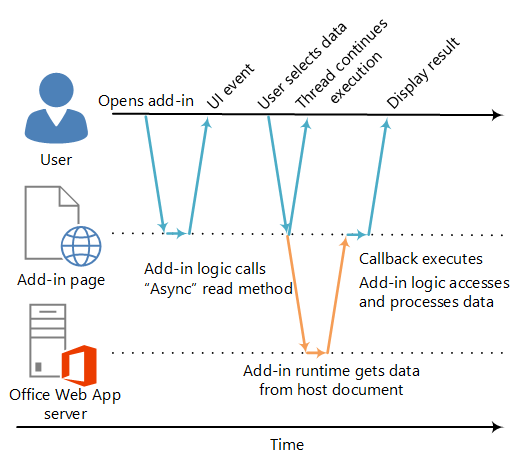
# Asynchronous programming in Office Add-ins

Why does the Office Add-ins API use asynchronous programming? Because JavaScript is a single-threaded language, if script invokes a long-running synchronous process, all subsequent script execution will be blocked until that process completes. Because certain operations against Office web clients (but rich clients as well) could block execution if they are run synchronously, most of the methods in the JavaScript API for Office are designed to execute asynchronously. This makes sure that Office Add-ins are responsive and highly performing. It also frequently requires you to write callback functions when working with these asynchronous methods.

The names of all asynchronous methods in the API end with "Async", such as the [Document.getSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.getselecteddataasync.md), [Binding.getDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.getdataasync.md), or [Item.loadCustomPropertiesAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.item.md) methods. When an "Async" method is called, it executes immediately and any subsequent script execution can continue. The optional callback function you pass to an "Async" method executes as soon as the data or requested operation is ready. This generally occurs promptly, but there can be a slight delay before it returns.

The following diagram shows the flow of execution for a call to an "Async" method that reads the data the user selected in a document open in the server-based Word Online or Excel Online. At the point when the "Async" call is made, the JavaScript execution thread is free to perform any additional client-side processing. (Although none are shown in the diagram.) When the "Async" method returns, the callback resumes execution on the thread, and the add-in can the access data, do something with it, and display the result. The same asynchronous execution pattern holds when working with the Office rich client host applications, such as Word 2013 or Excel 2013.

**Figure 1. Asynchronous programing execution flow**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15appAsyncProgFig01.png)

Support for this asynchronous design in both rich and web clients is part of the "write once-run cross-platform" design goals of the Office Add-ins development model. For example, you can create a content or task pane add-in with a single code base that will run in both Excel 2013 and Excel Online.

## Writing the callback function for an "Async" method

The callback function you pass as the callback argument to an "Async" method must declare a single parameter that the add-in runtime will use to provide access to an [AsyncResult](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.md) object when the callback function executes. You can write:

* An anonymous function that must be written and passed directly in line with the call to the "Async" method as thecallback parameter of the "Async" method.
* A named function, passing the name of that function as the callback parameter of an "Async" method.

An anonymous function is useful if you are only going to use its code once - because it has no name, you can't reference it in another part of your code. A named function is useful if you want to reuse the callback function for more than one "Async" method.

### Writing an anonymous callback function

The following anonymous callback function declares a single parameter named result that retrieves data from the [AsyncResult.value](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.status.md) property when the callback returns.

function (result) {

write('Selected data: ' + result.value);

}

The following example shows how to pass this anonymous callback function in line in the context of a full "Async" method call to the **Document.getSelectedDataAsync** method.

* The first coercionType argument, Office.CoercionType.Text, specifies to return the selected data as a string of text.
* The second callback argument is the anonymous function passed in-line to the method. When the function executes, it uses the result parameter to access the **value** property of the **AsyncResult** object to display the data selected by the user in the document.

Office.context.document.getSelectedDataAsync(Office.CoercionType.Text,

function (result) {

write('Selected data: ' + result.value);

}

});

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

You can also use the parameter of your callback function to access other properties of the **AsyncResult** object. Use the [AsyncResult.status](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.error.md) property to determine if the call succeeded or failed. If your call fails you can use the [AsyncResult.error](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.context.md)property to access an [Error](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/error.md) object for error information.

For more information about using the **getSelectedDataAsync** method, see [Read and write data to the active selection in a document or spreadsheet](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/read-and-write-data-to-the-active-selection-in-a-document-or-spreadsheet.md).

### Writing a named callback function

Alternatively, you can write a named function and pass its name to the callback parameter of an "Async" method. For example, the previous example can be rewritten to pass a function named writeDataCallback as the callback parameter like this.

Office.context.document.getSelectedDataAsync(Office.CoercionType.Text,

writeDataCallback);

// Callback to write the selected data to the add-in UI.

function writeDataCallback(result) {

write('Selected data: ' + result.value);

}

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

## Differences in what's returned to the AsyncResult.value property

The **asyncContext**, **status**, and **error** properties of the **AsyncResult** object return the same kinds of information to the callback function passed to all "Async" methods. However, what's returned to the **AsyncResult.value** property varies depending on the functionality of the "Async" method.

For example, the **addHandlerAsync** methods (of the [Binding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.md), [CustomXmlPart](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/customxmlpart.customxmlpart.md), [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md), [RoamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md), and [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md)objects) are used to add event handler functions to the items represented by these objects. You can access the **AsyncResult.value** property from the callback function you pass to any of the **addHandlerAsync** methods, but since no data or object is being accessed when you add an event handler, the **value** property always returns **undefined** if you attempt to access it.

On the other hand, if you call the **Document.getSelectedDataAsync** method, it returns the data the user selected in the document to the **AsyncResult.value** property in the callback. Or, if you call the [Bindings.getAllAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.getallasync.md) method, it returns an array of all of the **Binding** objects in the document. And, if you call the [Bindings.getByIdAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.getbyidasync.md) method, it returns a single **Binding** object.

For a description of what's returned to the **AsyncResult.value** property for an "Async" method, see the "Callback value" section of that method's reference topic. For a summary of all of the objects that provide "Async" methods, see the table at the bottom of the [AsyncResult](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.md) object topic.

## Asynchronous programming patterns

The JavaScript API for Office supports two kinds of asynchronous programming patterns:

* Using nested callbacks
* Using the promises pattern

Asynchronous programming with callback functions frequently requires you to nest the returned result of one callback within two or more callbacks. If you need to do so, you can use nested callbacks from all "Async" methods of the API.

Using nested callbacks is a programming pattern familiar to most JavaScript developers, but code with deeply nested callbacks can be difficult to read and understand. As an alternative to nested callbacks, the JavaScript API for Office also supports an implementation of the promises pattern. However, in the current version of the JavaScript API for Office, the promises pattern only works with code for [bindings in Excel spreadsheets and Word documents](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/bind-to-regions-in-a-document-or-spreadsheet.md).

### Asynchronous programming using nested callback functions

Frequently, you need to perform two or more asynchronous operations to complete a task. To accomplish that, you can nest one "Async" call inside another.

The following code example nests two asynchronous calls.

* First, the [Bindings.getByIdAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/bindings.getbyidasync.md) method is called to access a binding in the document named "MyBinding". The **AsyncResult** object returned to the result parameter of that callback provides access to the specified binding object from the **AsyncResult.value** property.
* Then, the binding object accessed from the first result parameter is used to call the [Binding.getDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.getdataasync.md) method.
* Finally, the result2 parameter of the callback passed to the **Binding.getDataAsync** method is used to display the data in the binding.

function readData() {

Office.context.document.bindings.getByIdAsync("MyBinding", function (result) {

result.value.getDataAsync({ coercionType: 'text' }, function (result2) {

write(result2.value);

});

});

}

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

This basic nested callback pattern can be used for all asynchronous methods in the JavaScript API for Office.

The following sections show how to use either anonymous or named functions for nested callbacks in asynchronous methods.

#### Using anonymous functions for nested callbacks

In the following example, two anonymous functions are declared inline and passed into the **getByIdAsync** and **getDataAsync**methods as nested callbacks. Because the functions are simple and inline, the intent of the implementation is immediately clear.

Office.context.document.bindings.getByIdAsync('myBinding', function (bindingResult) {

bindingResult.value.getDataAsync(function (getResult) {

if (getResult.status == Office.AsyncResultStatus.Failed) {

write('Action failed. Error: ' + asyncResult.error.message);

} else {

write('Data has been read successfully.');

}

});

});

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

#### Using named functions for nested callbacks

In complex implementations, it may be helpful to use named functions to make your code easier to read, maintain, and reuse. In the following example, the two anonymous functions from the example in the previous section have been rewritten as functions named deleteAllData and showResult. These named functions are then passed into the **getByIdAsync** and **deleteAllDataValuesAsync** methods as callbacks by name.

Office.context.document.bindings.getByIdAsync('myBinding', deleteAllData);

function deleteAllData(asyncResult) {

asyncResult.value.deleteAllDataValuesAsync(showResult);

}

function showResult(asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

write('Action failed. Error: ' + asyncResult.error.message);

} else {

write('Data has been deleted successfully.');

}

}

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

### Asynchronous programming using the promises pattern to access data in bindings

Instead of passing a callback function and waiting for the function to return before execution continues, the promises programming pattern immediately returns a promise object that represents its intended result. However, unlike true synchronous programming, under the covers the fulfillment of the promised result is actually deferred until the Office Add-ins runtime environment can complete the request. An onError handler is provided to cover situations when the request can't be fulfilled.

The JavaScript API for Office provides the [Office.select](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.select.md) method to support the promises pattern for working with existing binding objects. The promise object returned to the **Office.select** method supports only the four methods that you can access directly from the [Binding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.md) object: [getDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.getdataasync.md), [setDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.setdataasync.md), [addHandlerAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.value.md), and [removeHandlerAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.removehandlerasync.md).

The promises pattern for working with bindings takes this form:

**Office.select(**selectorExpression, onError**).**BindingObjectAsyncMethod

The selectorExpression parameter takes the form "bindings#bindingId", where bindingId is the name ( **id**) of a binding that you created previously in the document or spreadsheet (using one of the "addFrom" methods of the **Bindings** collection: **addFromNamedItemAsync**, **addFromPromptAsync**, or **addFromSelectionAsync**). For example, the selector expressionbindings#cities specifies that you want to access the binding with an **id** of 'cities'.

The onError parameter is an error handling function which takes a single parameter of type **AsyncResult** that can be used to access an **Error** object, if the **select** method fails to access the specified binding. The following example shows a basic error handler function that can be passed to the onError parameter.

function onError(result){

var err = result.error;

write(err.name + ": " + err.message);

}

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

Replace the BindingObjectAsyncMethod placeholder with a call to any of the four **Binding** object methods supported by the promise object: **getDataAsync**, **setDataAsync**, **addHandlerAsync**, or **removeHandlerAsync**. Calls to these methods don't support additional promises. You must call them using the [nested callback function pattern](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/asynchronous-programming-in-office-add-ins.md#AsyncProgramming_NestedCallbacks).

After a **Binding** object promise is fulfilled, it can be reused in the chained method call as if it were a binding (the add-in runtime won't asynchronously retry fulfilling the promise). If the **Binding** object promise can't be fulfilled, the add-in runtime will try again to access the binding object the next time one of its asynchronous methods is invoked.

The following code example uses the **select** method to retrieve a binding with the **id** " cities" from the **Bindings** collection, and then calls the [addHandlerAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/asyncresult.value.md) method to add an event handler for the [dataChanged](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.bindingdatachangedevent.md) event of the binding.

function addBindingDataChangedEventHandler() {

Office.select("bindings#cities", function onError(){/\* error handling code \*/}).addHandlerAsync(Office.EventType.BindingDataChanged,

function (eventArgs) {

doSomethingWithBinding(eventArgs.binding);

});

}

**Important:** The **Binding** object promise returned by the **Office.select** method provides access to only the four methods of the **Binding** object. If you need to access any of the other members of the **Binding** object, instead you must use the **Document.bindings** property and **Bindings.getByIdAsync** or **Bindings.getAllAsync** methods to retrieve the **Binding**object. For example, if you need to access any of the **Binding** object's properties (the **document**, **id**, or **type** properties), or need to access the properties of the [MatrixBinding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.matrixbinding.md) or [TableBinding](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/binding.tablebinding.md) objects, you must use the **getByIdAsync** or **getAllAsync** methods to retrieve a **Binding** object.

## Passing optional parameters to asynchronous methods

The common syntax for all "Async" methods follows this pattern:

AsyncMethod ( RequiredParameters , [ OptionalParameters ], CallbackFunction );

All asynchronous methods support optional parameters, which are passed in as a JavaScript Object Notation (JSON) object that contains one or more optional parameters. The JSON object containing the optional parameters is an unordered collection of key-value pairs with the ":" character separating the key and the value. Each pair in the object is comma-separated, and the entire set of pairs is enclosed in braces. The key is the parameter name, and value is the value to pass for that parameter.

You can create the JSON object that contains optional parameters inline, or by creating an options object and passing that in as the options parameter.

### Passing optional parameters inline

For example, the syntax for calling the [Document.setSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.setselecteddataasync.md) method with optional parameters inline looks like this:

Office.context.document.setSelectedDataAsync(data, {coercionType: 'coercionType', asyncContext:' asyncContext},callback);

In this form of the calling syntax, the two optional parameters, coercionType and asyncContext, are defined as a JSON object inline enclosed in braces.

The following example shows how to call to the **Document.setSelectedDataAsync** method by specifying optional parameters inline.

Office.context.document.setSelectedDataAsync(

"<html><body>hello world</body></html>",

{coercionType: "html", asyncContext: 42},

function(asyncResult) {

write(asyncResult.status + " " + asyncResult.asyncContext);

}

)

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

**Note:** You can specify optional parameters in any order in the JSON object as long as their names are specified correctly.

### Passing optional parameters in an options object

Alternatively, you can create an object named options that specifies the optional parameters separately from the method call, and then pass the options object as the options argument.

The following example shows one way of creating the options object, where parameter1, value1, and so on, are placeholders for the actual parameter names and values.

var options = {

parameter1: value1,

parameter2: value2,

...

parameterN: valueN

};

Which looks like the following example when used to specify the [ValueFormat](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/valueformat-enumeration.md) and [FilterType](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/filtertype-enumeration.md) parameters.

var options = {

valueFormat: "unformatted",

filterType: "all"

};

Here's another way of creating the options object.

var options = {};

options[parameter1] = value1;

options[parameter2] = value2;

...

options[parameterN] = valueN;

Which looks like the following example when used to specify the **ValueFormat** and **FilterType** parameters.:

var options = {};

options["ValueFormat"] = "unformatted";

options["FilterType"] = "all";

**Note** When using either method of creating the options object, you can specify optional parameters in any order as long as their names are specified correctly.

The following example shows how to call to the **Document.setSelectedDataAsync** method by specifying optional parameters in an options object.

var options = {

coercionType: "html",

asyncContext: 42

};

document.setSelectedDataAsync(

"<html><body>hello world</body></html>",

options,

function(asyncResult) {

write(asyncResult.status + " " + asyncResult.asyncContext);

}

)

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

In both optional parameter examples, the callback parameter is specified as the last parameter (following the inline optional parameters, or following the options argument object). Alternatively, you can specify the callback parameter inside either the inline JSON object, or in the options object. However, you can pass the callback parameter in only one location: either in the options object (inline or created externally), or as the last parameter, but not both.

# Persisting add-in state and settings

Office Add-ins are essentially web applications running in the stateless environment of a browser control. As a result, your add-in may need to persist data to maintain the continuity of certain operations or features across sessions of using your add-in. For example, your add-in may have custom settings or other values that it needs to save and reload the next time it's initialized, such as a user's preferred view or default location.

To do that, you can:

* Use members of the JavaScript API for Office that store data as name/value pairs in a property bag stored in a location that depends on add-in type.
* Use techniques provided by the underlying browser control: browser cookies, or HTML5 web storage ([localStorage](http://msdn.microsoft.com/en-us/library/cc848902%28v=vs.85%29.aspx) or [sessionStorage](http://msdn.microsoft.com/en-us/library/cc197020%28v=vs.85%29.aspx)).

This article focuses on how to use the JavaScript API for Office to persist add-in state. For examples of using browser cookies and web storage, see the [Excel-Add-in-JavaScript-PersistCustomSettings](https://github.com/OfficeDev/Excel-Add-in-JavaScript-PersistCustomSettings).

## Persisting add-in state and settings with the JavaScript API for Office

The JavaScript API for Office provides the [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md), [RoamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md), and [CustomProperties](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/CustomProperties.md) objects for saving add-in state across sessions as described in the following table. In all cases, the saved settings values are associated with the [Id](http://msdn.microsoft.com/en-us/library/67c4344a-935c-09d6-1282-55ee61a2838b%28Office.15%29.aspx) of the add-in that created them.

| **Object** | **Add-in type support** | **Storage location** | **Office host support** |
| --- | --- | --- | --- |
| [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) | content and task pane | The document, spreadsheet, or presentation the add-in is working with.Content and task pane add-in settings are available to the add-in that created them from the document where they are saved. **Important:** Don't store passwords and other sensitive personally identifiable information (PII) with the **Settings** object. The data saved isn't visible to end users, but it is stored as part of the document, which is accessible by reading the document's file format directly. You should limit your add-in's use of PII and store any PII required by your add-in only on the server hosting your add-in as a user-secured resource. | Word, Excel, or PowerPoint **Note:** Task pane add-ins for Project 2013 don't support the **Settings** API for storing add-in state or settings. However, for add-ins running in Project (as well as other Office host applications) you can use techniques such as browser cookies or web storage. For more information on these techniques, see the [Excel-Add-in-JavaScript-PersistCustomSettings](https://github.com/OfficeDev/Excel-Add-in-JavaScript-PersistCustomSettings). |
| [RoamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) | Outlook | The user's Exchange server mailbox where the add-in is installed.Because these settings are stored in the user's server mailbox, they can "roam" with the user and are available to the add-in when it is running in the context of any supported client host application or browser accessing that user's mailbox. Outlook add-in roaming settings are available only to the add-in that created them, and only from the mailbox where the add-in is installed. | Outlook |
| [CustomProperties](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/CustomProperties.md) | Outlook | The message, appointment, or meeting request item the add-in is working with. Outlook add-in item custom properties are available only to the add-in that created them, and only from the item where they are saved. | Outlook |

## Settings data is managed in memory at runtime

Internally, the data in the property bag accessed with the **Settings**, **CustomProperties**, or **RoamingSettings** objects is stored as a serialized JavaScript Object Notation (JSON) object that contains name/value pairs. The name (key) for each value must be a **string**, and the stored value can be a JavaScript **string**, **number**, **date**, or **object**, but not a **function**.

This example of the property bag structure contains three defined **string** values named firstName, location, anddefaultView.

{

"firstName":"Erik",

"location":"98052",

"defaultView":"basic"

}

After the settings property bag is saved during the previous add-in session, it can be loaded when the add-in is initialized or at any point after that during the add-in's current session. During the session, the settings are managed in entirely in memory using the **get**, **set**, and **remove** methods of the object that corresponds to the kind settings you are creating ( **Settings**,**CustomProperties**, or **RoamingSettings**).

**Important** To persist any additions, updates, or deletions made during the add-in's current session to the storage location, you must call the **saveAsync** method of the corresponding object used to work with that kind of settings. The **get**, **set**, and **remove** methods operate only on the in-memory copy of the settings property bag. If your add-in is closed without calling **saveAsync**, any changes made to settings during that session will be lost.

## How to save add-in state and settings per document for content and task pane add-ins

To persist state or custom settings of a content or task pane add-in for Word, Excel, or PowerPoint, you use the [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md)object and its methods. The property bag created with the methods of the **Settings** object are available only to the instance of the content or task pane add-in that created it, and only from the document in which it is saved.

The **Settings** object is automatically loaded as part of the [Document](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.md) object, and is available when the task pane or content add-in is activated. After the **Document** object is instantiated, you can access the **Settings** object with the [settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.settings.md) property of the **Document** object. During the lifetime of the session, you can just use the **Settings.get**, **Settings.set**, and **Settings.remove**methods to read, write, or remove persisted settings and add-in state from the in-memory copy of the property bag.

Because the set and remove methods operate against only the in-memory copy of the settings property bag, to save new or changed settings back to the document the add-in is associated with you must call the [Settings.saveAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.saveasync.md) method.

### Creating or updating a setting value

The following code example shows how to use the [Settings.set](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.set.md) method to create a setting called 'themeColor' with a value 'green'. The first parameter of the set method is the case-sensitive name (Id) of the setting to set or create. The second parameter is the value of the setting.

Office.context.document.settings.set('themeColor', 'green');

The setting with the specified name is created if it doesn't already exist, or its value is updated if it does exist. Use the **Settings.saveAsync** method to persist the new or updated settings to the document.

### Getting the value of a setting

The following example shows how use the [Settings.get](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.get.md) method to get the value of a setting called "themeColor". The only parameter of the **get** method is the case-sensitive name of the setting.

write('Current value for mySetting: ' + Office.context.document.settings.get('themeColor'));

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

The **get** method returns the value that was previously saved for the setting name that was passed in. If the setting doesn't exist, the method returns **null**.

### Removing a setting

The following example shows how to use the [Settings.remove](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.removehandlerasync.md) method to remove a setting with the name "themeColor". The only parameter of the **remove** method is the case-sensitive name of the setting.

Office.context.document.settings.remove('themeColor');

Nothing will happen if the setting does not exist. Use the **Settings.saveAsync** method to persist removal of the setting from the document.

### Saving your settings

To save any additions, changes, or deletions your add-in made to the in-memory copy of the settings property bag during the current session, you must call the [Settings.saveAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.saveasync.md) method to store them in the document. The only parameter of the **saveAsync** method is callback, which is a callback function with a single parameter.

Office.context.document.settings.saveAsync(function (asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

write('Settings save failed. Error: ' + asyncResult.error.message);

} else {

write('Settings saved.');

}

});

// Function that writes to a div with id='message' on the page.

function write(message){

document.getElementById('message').innerText += message;

}

The anonymous function passed into the **saveAsync** method as the callback parameter is executed when the operation is completed. The asyncResult parameter of the callback provides access to an **AsyncResult** object that contains the status of the operation. In the example, the function checks the **AsyncResult.status** property to see if the save operation succeeded or failed, and then displays the result in the add-in's page.

## How to save settings in the user's mailbox for Outlook add-ins as roaming settings

An Outlook add-in can use the [RoamingSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) object to save add-in state and settings data that is specific to the user's mailbox. This data is accessible only by that Outlook add-in on behalf of the user running the add-in. The data is stored on the user's Exchange Server mailbox, and is accessible when that user logs into his or her account and runs the Outlook add-in.

### Loading roaming settings

An Outlook add-in typically loads roaming settings in the [Office.initialize](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.initialize.md) event handler. The following JavaScript code example shows how to load existing roaming settings.

var \_mailbox;

var \_settings;

// The initialize function is required for all add-ins.

Office.initialize = function (reason) {

// Checks for the DOM to load using the jQuery ready function.

$(document).ready(function () {

// After the DOM is loaded, add-in-specific code can run.

// Initialize instance variables to access API objects.

\_mailbox = Office.context.mailbox;

\_settings = Office.context.roamingSettings;

});

}

### Creating or assigning a roaming setting

Continuing with the preceding example, the following setAppSetting function shows how to use the [RoamingSettings.set](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md)method to set or update a setting named cookie with today's date. Then, it saves all the roaming settings back to the Exchange Server with the [RoamingSettings.saveAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) method.

// Set an add-in setting.

function setAppSetting() {

\_settings.set("cookie", Date());

\_settings.saveAsync(saveMyAppSettingsCallback);

}

// Saves all roaming settings.

function saveMyAppSettingsCallback(asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

// Handle the failure.

}

}

The **saveAsync** method saves roaming settings asynchronously and takes an optional callback function. This code sample passes a callback function named saveMyAppSettingsCallback to the **saveAsync** method. When the asynchronous call returns, the asyncResult parameter of the saveMyAppSettingsCallback function provides access to an [AsyncResult](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/simple-types.md) object that you can use to determine the success or failure of the operation with the **AsyncResult.status** property.

### Removing a roaming setting

Also extending the preceding examples, the following removeAppSetting function, shows how to use the [RoamingSettings.remove](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) method to remove the cookie setting and save all the roaming settings back to the Exchange Server.

// Remove an application setting.

function removeAppSetting()

{

\_settings.remove("cookie");

\_settings.saveAsync(saveMyAppSettingsCallback);

}

## How to save settings per item for Outlook add-ins as custom properties

Custom properties let your Outlook add-in store information about an item it is working with. For example, if your Outlook add-in creates an appointment from a meeting suggestion in a message, you can use custom properties to store the fact that the meeting was created. This makes sure that if the message is opened again, your Outlook add-in doesn't offer to create the appointment again.

Before you can use custom properties for a particular message, appointment, or meeting request item, you must load the properties into memory by calling the [loadCustomPropertiesAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/Office.context.mailbox.item.md) method of the **Item** object. If any custom properties are already set for the current item, they are loaded from the Exchange server at this point. After you have loaded the properties, you can use the [set](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/CustomProperties.md) and [get](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/RoamingSettings.md) methods of the **CustomProperties** object to add, update, and retrieve properties in memory. To save any changes that you make to the item's custom properties, you must use the [saveAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/outlook/CustomProperties.md) method to persist the changes to the item on the Exchange server.

### Custom properties example

The following example shows a simplified set of functions for an Outlook add-in that uses custom properties. You can use this example as a starting point for your Outlook add-in that uses custom properties.

An Outlook add-in that uses these functions retrieves any custom properties by calling the **get** method on the \_customPropsvariable, as shown in the following example.

var property = \_customProps.get("propertyName");

This example includes the following functions:

| **Function name** | **Description** |
| --- | --- |
| Office.initialize | Initializes the add-in and loads the custom properties for the current item from the Exchange server. |
| customPropsCallback | Gets the custom properties that are returned from the Exchange server and saves it for later use. |
| updateProperty | Sets or updates a specific property, and then saves the change to the Exchange server. |
| removeProperty | Removes a specific property, and then persists the removal to the Exchange server. |
| saveCallback | Callback for calls to the **saveAsync** method in the updateProperty and removePropertyfunctions. |

var \_mailbox;

var \_customProps;

// The initialize function is required for all add-ins.

Office.initialize = function (reason) {

// Checks for the DOM to load using the jQuery ready function.

$(document).ready(function () {

// After the DOM is loaded, add-in-specific code can run.

\_mailbox = Office.context.mailbox;

\_mailbox.item.loadCustomPropertiesAsync(customPropsCallback);

});

}

// Get the item's custom properties from the server and save for later use.

function customPropsCallback(asyncResult) {

\_customProps = asyncResult.value;

}

// Sets or updates the specified property, and then saves the change

// to the server.

function updateProperty(name, value) {

\_customProps.set(name, value);

\_customProps.saveAsync(saveCallback);

}

// Removes the specified property, and then persists the removal

// to the server.

function removeProperty(name) {

\_customProps.remove(name);

\_customProps.saveAsync(saveCallback);

}

// Callback for calls to saveAsync method.

function saveCallback(asyncResult) {

if (asyncResult.status == Office.AsyncResultStatus.Failed) {

// Handle the failure.

}

}

# Privacy and security for Office Add-ins

## Understanding the add-in runtime

Office Add-ins are secured by an add-in runtime environment, a multiple-tier permissions model, and performance governors. This framework protects the user's experience in the following ways:

* Access to the host application's UI frame is managed.
* Only indirect access to the host application's UI thread is allowed.
* Modal interactions aren't allowed - for example, calls to JavaScript **alert**, **confirm**, and **prompt** functions aren't allowed because they're modal.

Further, the runtime framework provides the following benefits to ensure that an Office Add-in can't damage the user's environment:

* Isolates the process the add-in runs in.
* Doesn't require .dll or .exe replacement or ActiveX components.
* Makes add-ins easy to install and uninstall.

Also, the use of memory, CPU, and network resources by Office Add-ins is governable to ensure that good performance and reliability are maintained.

The following sections briefly describe how the runtime architecture supports running add-ins in Office clients on Windows-based devices, on OS X Mac devices, and in Office Online clients on the web.

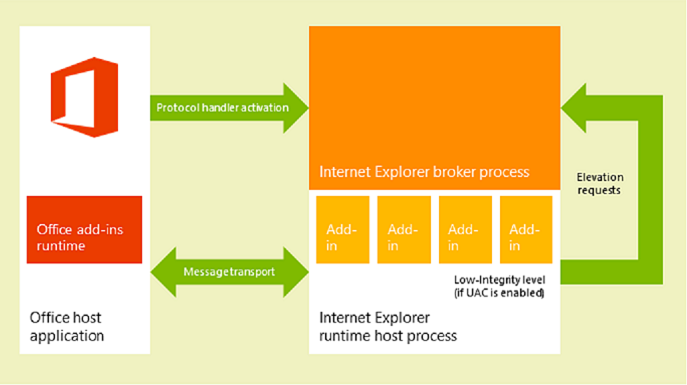
**Note:** To learn about using WIP and Intune with Office Add-ins, see [Use WIP and Intune to protect enterprise data in documents running Office Add-ins](https://docs.microsoft.com/en-us/microsoft-365-enterprise/office-add-ins-wip).

### Clients for Windows and OS X devices

In supported clients for desktop and tablet devices, such as Excel, Outlook, and Outlook for Mac, Office Add-ins are supported by integrating an in-process component, the Office Add-ins runtime, which manages the add-in lifecycle and enables interoperability between the add-in and the client application. The add-in webpage itself is hosted out-of-process. As shown in figure 1, on a Windows desktop or tablet device, the add-in webpage is hosted inside an Internet Explorer control which, in turn, is hosted inside an add-in runtime process that provides security and performance isolation.

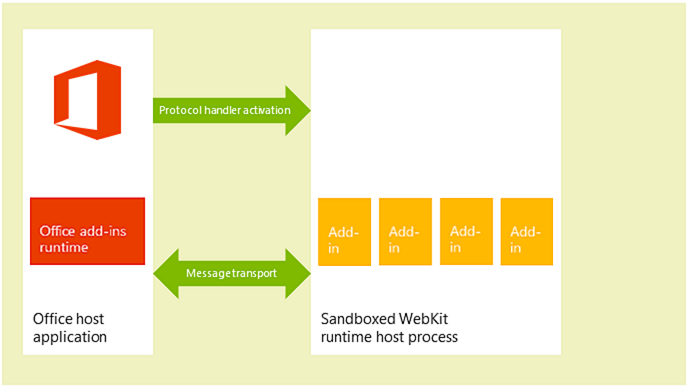
On Windows desktops, Protect Mode in Internet Explorer must be enabled for the Restricted Site Zone. This is typically enabled by default. If it is disabled, an [error will occur](https://support.microsoft.com/en-us/help/2761180/apps-for-office-don-t-start-if-you-disable-protected-mode-for-the-restricted-sites-zone-in-internet-explorer) when you try to launch an add-in.

**Figure 1. Office Add-ins runtime environment in Windows-based desktop and tablet clients**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview02.png)

As shown in figure 2, on an OS X Mac desktop, the add-in web page is hosted inside a sandboxed WebKit runtime host process which helps provide similar level of security and performance protection.

**Figure 2. Office Add-ins runtime environment in OS X Mac clients**

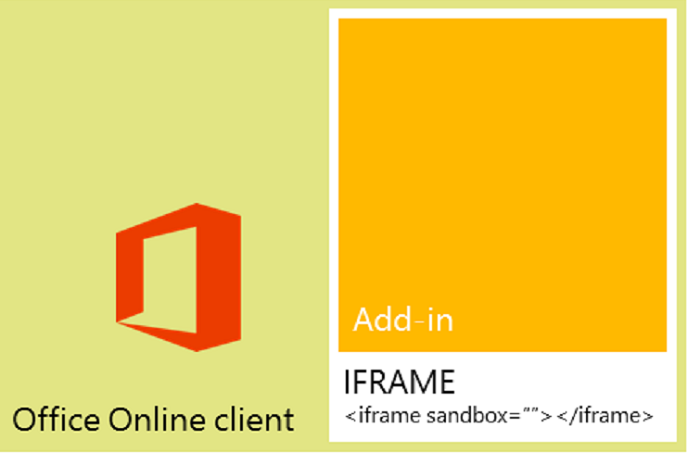
[](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview_Mac_02.png)

The Office Add-ins runtime manages interprocess communication, the translation of JavaScript API calls and events into native ones, as well as UI remoting support to enable the add-in to be rendered inside the document, in a task pane, or adjacent to an email message, meeting request, or appointment.

### Web clients

In supported Web clients, such as Excel Online and Outlook Web App, Office Add-ins are hosted in an **iframe** that runs using the HTML5 **sandbox** attribute. ActiveX components or navigating the main page of the web client are not allowed. Office Add-ins support is enabled in the web clients by the integration of the JavaScript API for Office. In a similar way to the desktop client applications, the JavaScript API manages the add-in lifecycle and interoperability between the add-in and the web client. This interoperability is implemented by using a special cross-frame post message communication infrastructure. The same JavaScript library (Office.js) that is used on desktop clients is available to interact with the web client. Figure 3 shows the infrastructure that supports Office Add-ins in the Office Online (running in the browser), and the relevant components (the web client, **iframe**, Office Add-ins runtime, and JavaScript API for Office) that are required to support them.

**Figure 3. Infrastructure that supports Office Add-ins in Office web clients**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/DK2_AgaveOverview03.png)

## Add-in integrity in the Office Store

You can make your Office Add-ins available to the public by publishing them in the Office Store. The Office Store enforces the following measures to maintain the integrity of add-ins:

* Requires the host server of an Office Add-in to always use Secure Sockets Layer (SSL) to communicate.
* Requires a developer to provide proof of identity, a contractual agreement, and a compliant privacy policy to submit add-ins.
* Ensures that the source of add-ins is accessible in read-only mode.
* Supports a user-review system for available add-ins to promote a self-policing community.

## Addressing end users' privacy concerns

This section describes the protection offered by the Office Add-ins platform from the customer's (end user's) perspective, and provides guidelines for how to support users' expectations and how to securely handle users' personally identifiable information (PII).

### End users' perspective

Office Add-ins are built using web technologies that run in a browser control or **iframe**. Because of this, using add-ins is similar to browsing to web sites on the Internet or intranet. Add-ins can be external to an organization (if you acquire the add-in from the Office Store) or internal (if you acquire the add-in from an Exchange Server add-in catalog, SharePoint add-in catalog, or file share on an organization's network). Add-ins have limited access to the network and most add-ins can read or write to the active document or mail item. The add-in platform applies certain constraints before a user or administrator installs or starts an add-in. But as with any extensibility model, users should be cautious before starting an unknown add-in.

The add-in platform addresses end users' privacy concerns in the following ways:

* Data communicated with the web server that hosts a content, Outlook or task pane add-in as well as communication between the add-in and any web services it uses must be encrypted using the Secure Socket Layer (SSL) protocol.
* Before a user installs an add-in from the Office Store, the user can view the privacy policy and requirements of that add-in. In addition, Outlook add-ins that interact with users' mailboxes surface the specific permissions that they require; the user can review the terms of use, requested permissions and privacy policy before installing an Outlook add-in.
* When sharing a document, users also share add-ins that have been inserted in or associated with that document. If a user opens a document that contains an add-in that the user hasn't used before, the host application prompts the user to grant permission for the add-in to run in the document. In an organizational environment, the Office host application also prompts the user if the document comes from an external source.
* Users can enable or disable the access to the Office Store. For content and task pane add-ins, users manage access to trusted add-ins and catalogs from the **Trust Center** on the host Office client (opened from **File** > **Options** > **Trust Center**> **Trust Center Settings** > **Trusted Add-in Catalogs**). For Outlook add-ins, uses can manage add-ins by choosing the**Manage Add-ins** button: in Outlook for Windows, choose **File** > **Manage Add-ins**. In Outlook for Mac, choose the**Manage Add-ins** button on the add-in bar. In Outlook Web App choose the **Settings** menu (gear icon) > **Manage add-ins**. Administrators can also manage this access [by using group policy](http://technet.microsoft.com/en-us/library/jj219429.aspx#BKMK_Managing).
* The design of the add-in platform provides security and performance for end users in the following ways:
  + An Office Add-in runs in a web browser control that is hosted in an add-in runtime environment separate from the Office host application. This design provides both security and performance isolation from the host application.
  + Running in a web browser control allows the add-in to do almost anything a regular web page running in a browser can do but, at the same time, restricts the add-in to observe the same-origin policy for domain isolation and security zones.

Outlook add-ins provide additional security and performance features through Outlook add-in specific resource usage monitoring. For more information, see [Privacy, permissions, and security for Outlook add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/privacy-and-security.md).

### Developer guidelines to handle PII

You can read general PII protection guidelines for IT administrators and developers in [Protecting Privacy in the Development and Testing of Human Resources Applications](http://technet.microsoft.com/en-us/library/gg447064.aspx). The following lists some specific PII protection guidelines for you as a developer of Office Add-ins:

* The [Settings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/settings.md) object is intended for persisting add-in settings and state data across sessions for a content or task pane add-in, but don't store passwords and other sensitive PII in the **Settings** object. The data in the **Settings** object isn't visible to end users, but it is stored as part of the document's file format which is readily accessible. You should limit your add-in's use of PII and store any PII required by your add-in on the server hosting your add-in as a user-secured resource.
* Using some applications can reveal PII. Make sure that you securely store data for your users' identity, location, access times, and any other credentials so that data won't become available to other users of the add-in.
* If your add-in is available in the Office Store, the Office Store requirement for HTTPS protects PII transmitted between your web server and the client computer or device. However, if you re-transmit that data to other servers, make sure you observe the same level of protection.
* If you store users' PII, make sure you reveal that fact, and provide a way for users to inspect and delete it. If you submit your add-in to the Office Store, you can outline the data you collect and how it's used in the privacy statement.

## Developers' permission choices and security practices

Follow these general guidelines to support the security model of Office Add-ins, and drill down on more details for each add-in type.

### Permissions choices

The add-in platform provides a permissions model that your add-in uses to declare the level of access to a user's data that it requires for its features. Each permission level corresponds to the subset of the JavaScript API for Office your add-in is allowed to use for its features. For example, the **WriteDocument** permission for content and task pane add-ins allows access to the [Document.setSelectedDataAsync](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/document.setselecteddataasync.md) method that lets an add-in write to the user's document, but doesn't allow access to any of the methods for reading data from the document. This permission level makes sense for add-ins that only need to write to a document, such as an add-in where the user can query for data to insert into his or her document.

As a best practice, you should request permissions based on the principle of least privilege. That is, you should request permission to access only the minimum subset of the API that your add-in requires to function correctly. For example, if your add-in needs only to read data in a user's document for its features, you should request no more than the **ReadDocument**permission. (But, keep in mind that requesting insufficient permissions will result in the add-in platform blocking your add-in's use of some APIs and will generate errors at run time.)

You specify permissions in the manifest of your add-in, as shown in the example in this section below, and end users can see the requested permission level of an add-in before they decide to install or activate the add-in for the first time. Additionally, Outlook add-ins that request the **ReadWriteMailbox** permission require explicit administrator privilege to install.

The following example shows how a task pane add-in specifies the **ReadDocument** permission in its manifest. To keep permissions as the focus, other elements in the manifest aren't displayed.

<?xml version="1.0" encoding="utf-8"?>

<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:ver="http://schemas.microsoft.com/office/appforoffice/1.0"

xsi:type="TaskPaneApp">

... <!-- To keep permissions as the focus, not displaying other elements. -->

<Permissions>ReadDocument</Permissions>

...

</OfficeApp>

For more information about this for task pane and content add-ins , see [Requesting permissions for API use in content and task pane add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/requesting-permissions-for-api-use-in-content-and-task-pane-add-ins.md).

For more information about this for Outlook add-ins, see the following topics:

* [Privacy, permissions, and security for Outlook add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/privacy-and-security.md)
* [Understanding Outlook add-in permissions](https://github.com/OfficeDev/office-js-docs/blob/master/docs/outlook/understanding-outlook-add-in-permissions.md)

### Same origin policy

Because Office Add-ins are webpages that run in a web browser control, they must follow the same-origin policy enforced by the browser: by default, a webpage in one domain can't make [XmlHttpRequest](http://www.w3.org/TR/XMLHttpRequest/) web service calls to another domain other than the one where it is hosted.

One way to overcome this limitation is to use JSON/P -- provide a proxy for the web service by including a **script** tag with a **src** attribute that points to some script hosted on another domain. You can programmatically create the **script** tags, dynamically creating the URL to which to point the **src** attribute, and passing parameters to the URL via URI query parameters. Web service providers create and host JavaScript code at specific URLs, and return different scripts depending on the URI query parameters. These scripts then execute where they are inserted and work as expected.

The following is an example of JSON/P in the Outlook add-in example.

// Dynamically create an HTML SCRIPT element that obtains the details for the specified video.

function loadVideoDetails(videoIndex) {

// Dynamically create a new HTML SCRIPT element in the webpage.

var script = document.createElement("script");

// Specify the URL to retrieve the indicated video from a feed of a current list of videos,

// as the value of the src attribute of the SCRIPT element.

script.setAttribute("src", "https://gdata.youtube.com/feeds/api/videos/" +

videos[videoIndex].Id + "?alt=json-in-script&amp;callback=videoDetailsLoaded");

// Insert the SCRIPT element at the end of the HEAD section.

document.getElementsByTagName('head')[0].appendChild(script);

}

Exchange and SharePoint provide client-side proxies to enable cross-domain access. In general, same origin policy on an intranet isn't as strict as on the Internet. For more information, see [Same Origin Policy Part 1: No Peeking](http://blogs.msdn.com/b/ieinternals/archive/2009/08/28/explaining-same-origin-policy-part-1-deny-read.aspx) and [Addressing same-origin policy limitations in Office Add-ins](https://github.com/OfficeDev/office-js-docs/blob/master/docs/develop/addressing-same-origin-policy-limitations.md).

### Tips to prevent malicious cross-site scripting

An ill-intentioned user could attack the origin of an add-in by entering malicious script through the document or fields in the add-in. A developer should process user input to avoid executing a malicious user's JavaScript within his or her domain. The following are some good practices to follow to handle user input from a document or mail message, or via fields in an add-in:

* Instead of the DOM property [innerHTML](http://msdn.microsoft.com/en-us/library/ie/ms533897.aspx), use the [innerText](https://msdn.microsoft.com/library/ms533899.aspx) and [textContent](https://developer.mozilla.org/en-US/docs/DOM/Node.textContent) properties where appropriate. Do the following for Internet Explorer and Firefox cross-browser support:

var text = x.innerText || x.textContent

For information about the differences between **innerText** and **textContent**, see [Node.textContent](https://developer.mozilla.org/en-US/docs/DOM/Node.textContent). For more information about DOM compatibility across common browsers, see [W3C DOM Compatibility - HTML](http://www.quirksmode.org/dom/w3c_html.html#t07).

* If you must use **innerHTML**, make sure the user's input doesn't contain malicious content before passing it to**innerHTML**. For more information and an example of how to use **innerHTML** safely, see [innerHTML](http://msdn.microsoft.com/en-us/library/ie/ms533897.aspx) property.
* If you are using jQuery, use the [.text()](http://api.jquery.com/text/) method instead of the [.html()](http://api.jquery.com/html/) method.
* Use the [toStaticHTML](http://msdn.microsoft.com/en-us/library/ie/cc848922.aspx) method to remove any dynamic HTML elements and attributes in users' input before passing it to **innerHTML**.
* Use the [encodeURIComponent](http://msdn.microsoft.com/en-us/library/8202bce6-1342-40dc-a5ef-ac6d210a7d15.aspx) or [encodeURI](http://msdn.microsoft.com/en-us/library/17bab5a2-bcd4-46c2-8b52-b2b5a0ed98a3.aspx) function to encode text that is intended to be a URL that comes from or contains user input.
* See [Developing secure add-ins](http://msdn.microsoft.com/en-us/library/windows/apps/hh849625.aspx) for more best practices to create more secure web solutions.

### Tips to prevent "Clickjacking"

Because Office Add-ins are rendered in an iframe when running in a browser with Office Online host applications, use the following tips to minimize the risk of [clickjacking](http://en.wikipedia.org/wiki/Clickjacking) -- a technique used by hackers to fool users into revealing confidential information.

First, identify sensitive actions that your add-in can perform. These include any actions that an unauthorized user could use with malicious intent, such as initiating a financial transaction or publishing sensitive data. For example, your add-in might let the user send a payment to a user-defined recipient.

Second, for sensitive actions, your add-in should confirm with the user before it executes the action. This confirmation should detail what effect the action will have. It should also detail how the user can prevent the action, if necessary, whether by choosing a specific button marked "Don't Allow" or by ignoring the confirmation.

Third, to ensure that no potential attacker can hide or mask the confirmation, you should display it outside the context of the add-in (that is, not in an HTML dialog box).

Here are some examples of how you could get confirmation:

* Send an email to the user that contains a confirmation link.
* Send a text message to the user that includes a confirmation code that the user can enter in the add-in.
* Open a confirmation dialog in a new browser window to a page that cannot be iframed. This is typically the pattern that is used by login pages. Use the [dialog api](https://dev.office.com/docs/add-ins/develop/dialog-api-in-office-add-ins) to create a new dialog.

Also, ensure that the address you use for contacting the user couldn't have been provided by a potential attacker. For example, for payment confirmations use the address on file for the authorized user's account.

### Other security practices

Developers should also take note of the following security practices:

* Developers shouldn't use ActiveX controls in Office Add-ins as ActiveX controls don't support the cross-platform nature of the add-in platform.
* Content and task pane add-ins assume the same SSL settings that Internet Explorer uses by default, and allows most content to be delivered only by SSL. Outlook add-ins require all content to be delivered by SSL. Developers must specify in the **SourceLocation** element of the add-in manifest a URL that uses HTTPS, to identify the location of the HTML file for the add-in.

To make sure add-ins aren't delivering content by using HTTP, when testing add-ins, developers should make sure the following settings are selected in Internet Explorer and no security warnings appear in their test scenarios:

* + Make sure the security setting, **Display mixed content**, for the **Internet** zone is set to **Prompt**. You can do that by selecting the following in Internet Explorer: on the **Security** tab of the **Internet Options** dialog box, select the **Internet** zone, select **Custom level**, scroll to look for **Display mixed content**, and select **Prompt** if it isn't already selected.
  + Make sure **Warn if Changing between Secure and not secure mode** is selected in the **Advanced** tab of the **Internet Options** dialog box.
* To make sure that add-ins don't use excessive CPU core or memory resources and cause any denial of service on a client computer, the add-in platform establishes resource usage limits. As part of testing, developers should verify whether an add-in performs within the resource usage limits.
* Before publishing an add-in, developers should make sure that any personal identifiable information that they expose in their add-in files is secure.
* Developers shouldn't embed keys that they use to access third-party APIs or services (such as Bing, Google, or Facebook) directly in the HTML pages of their add-in. Instead, they should create a custom web service or store the keys in some other form of secure web storage that they can then call to pass the key value to their add-in.
* Developers should do the following when submitting an add-in to the Office Store:
  + Host the add-in they are submitting on a web server that supports SSL.
  + Produce a statement outlining a compliant privacy policy.
  + Be ready to sign a contractual agreement upon submitting the add-in.

Other than resource usage rules, developers for Outlook add-ins should also make sure their add-ins observe limits for specifying activation rules and using the JavaScript API. For more information, see [Limits for activation and JavaScript API for Outlook add-ins](http://msdn.microsoft.com/library/e0c9e3d0-517e-4333-b8bd-e169c51a07f6.aspx).

## IT administrators' control

In a corporate setting, IT administrators have ultimate authority over enabling or disabling access to the Office Store and any private catalogs.

# Addressing same-origin policy limitations in Office Add-ins

The same-origin policy enforced by the browser prevents a script loaded from one domain from getting or manipulating properties of a webpage from another domain. This means that, by default, the domain of a requested URL must be the same as the domain of the current webpage. For example, this policy will prevent a webpage in one domain from making [XmlHttpRequest](http://www.w3.org/TR/XMLHttpRequest/) web-service calls to a domain other than the one where it is hosted.

Because Office Add-ins are hosted in a browser control, the same-origin policy applies to script running in their web pages as well.

To overcome same-origin policy enforcement when you develop add-ins, you can:

* Use JSON/P for anonymous access.
* Implement server-side script using a token-based authentication scheme.
* Using cross-origin resource sharing (CORS).
* Build your own proxy using IFRAME and POST MESSAGE.

## Using JSON/P for anonymous access

One way to overcome this limitation is to use JSON/P to provide a proxy for the web service. You do this by including a script tag with a src attribute that points to some script hosted on any domain. You can programmatically create the script tags, dynamically create the URL to point the src attribute to, and then pass parameters to the URL via URI query parameters. Web service providers create and host JavaScript code at specific URLs, and return different scripts depending on the URI query parameters. These scripts then execute where they are inserted and work as expected.

The following is an example of JSON/P that uses a technique that will work in any Office Add-in.

// Dynamically create an HTML SCRIPT element that obtains the details for the specified video.

function loadVideoDetails(videoIndex) {

// Dynamically create a new HTML SCRIPT element in the webpage.

var script = document.createElement("script");

// Specify the URL to retrieve the indicated video from a feed of a current list of videos,

// as the value of the src attribute of the SCRIPT element.

script.setAttribute("src", "https://gdata.youtube.com/feeds/api/videos/" +

videos[videoIndex].Id + "?alt=json-in-script&amp;callback=videoDetailsLoaded");

// Insert the SCRIPT element at the end of the HEAD section.

document.getElementsByTagName('head')[0].appendChild(script);

}

## Implementing server-side script using a token-based authentication scheme

Another way to address same-origin policy limitations is to implement the add-in's webpage as an ASP page that uses OAuth or caches credentials in cookies.

For an example that uses OAuth for authentication, see [Twitter SharePoint web part with OAuth](http://aidangarnish.net/post/Twitter-SharePoint-Web-Part-With-OAuth).

For an example of server-side code that shows how to use the Cookie object in System.Net to get and set cookie values, see the [Value](http://msdn2.microsoft.com/EN-US/library/4f772twc) property.

## Using cross-origin resource sharing (CORS)

For an example of using the cross-origin resource sharing feature of [XmlHttpRequest2](http://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html), see the "Cross Origin Resource Sharing (CORS)" section of [New Tricks in XMLHttpRequest2](http://www.html5rocks.com/en/tutorials/file/xhr2/).

## Building your own proxy using IFRAME and POST MESSAGE

For an example of how to build your own proxy using IFRAME and POST MESSAGE, see [Cross-Window Messaging](http://ejohn.org/blog/cross-window-messaging/).

# Localization for Office Add-ins

You can implement any localization scheme that's appropriate for your Office Add-in. The JavaScript API and manifest schema of the Office Add-ins platform provide some choices. You can use the JavaScript API for Office to determine a locale and display strings based on the locale of the host application, or to interpret or display data based on the locale of the data. You can use the manifest to specify locale-specific add-in file location and descriptive information. Alternatively, you can use Microsoft Ajax script to support globalization and localization.

## Use the JavaScript API to determine locale-specific strings

The JavaScript API for Office provides two properties that support displaying or interpreting values consistent with the locale of the host application and data:

* [Context.displayLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.displaylanguage.md) specifies the locale (or language) of the user interface of the host application. The following example verifies if the host application uses the en-US or fr-Fr locale, and displays a locale-specific greeting.

function sayHelloWithDisplayLanguage() {

var myLanguage = Office.context.displayLanguage;

switch (myLanguage) {

case 'en-US':

write('Hello!');

break;

case 'fr-FR':

write('Bonjour!');

break;

}

}

// Function that writes to a div with id='message' on the page.

function write(message) {

document.getElementById('message').innerText += message;

}

* [Context.contentLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.contentlanguage.md) specifies the locale (or language) of the data. Extending the last code sample, instead of checking the [displayLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.displaylanguage.md) property, assign myLanguage to the [contentLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.contentlanguage.md) property, and use the rest of the same code to display a greeting based on the locale of the data:

var myLanguage = Office.context.contentLanguage;

## Control localization from the manifest

Every Office Add-in specifies a [DefaultLocale](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/defaultlocale.md) element and a locale in its manifest. By default, the Office Add-in platform and Office host applications apply the values of the [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md), [DisplayName](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/displayname.md), [IconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/iconurl.md), [HighResolutionIconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/highresolutioniconurl.md), and [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) elements to all locales. You can optionally support specific values for specific locales, by specifying an [Override](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/override.md) child element for each additional locale, for any of these five elements. The value for the [DefaultLocale](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/defaultlocale.md) element and for the Locale attribute of the [Override](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/override.md) element is specified according to [RFC 3066](https://www.rfc-editor.org/info/rfc3066), "Tags for the Identification of Languages." Table 1 describes the localizing support for these elements.

**Table 1. Localization support**

| **Element** | **Localization support** |
| --- | --- |
| [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md) | Users in each locale you specify can see a localized description for the add-in in the Office Store (or private catalog). For Outlook add-ins, users can see the description in the Exchange Admin Center (EAC) after installation. |
| [DisplayName](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/displayname.md) | Users in each locale you specify can see a localized description for the add-in in the Office Store (or private catalog). For Outlook add-ins, users can see the display name as a label for the Outlook add-in button and in the EAC after installation. For content and task pane add-ins, users can see the display name in the ribbon after installing the add-in. |
| [IconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/iconurl.md) | The icon image is optional. You can use the same override technique to specify a certain image for a specific culture. If you use and localize an icon, users in each locale you specify can see a localized icon image for the add-in. For Outlook add-ins, users can see the icon in the EAC after installing the add-in. For content and task pane add-ins, users can see the icon in the ribbon after installing the add-in. |
| [HighResolutionIconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/highresolutioniconurl.md)   **Important** This element is available only when using add-in manifest version 1.1. | The high resolution icon image is optional but if it is specified, it must occur after the [IconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/iconurl.md)element. When [HighResolutionIconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/highresolutioniconurl.md) is specified, and the add-in is installed on a device that supports high dpi resolution, the [HighResolutionIconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/highresolutioniconurl.md) value is used instead of the value for [IconUrl](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/iconurl.md). You can use the same override technique to specify a certain image for a specific culture. If you use and localize an icon, users in each locale you specify can see a localized icon image for the add-in. For Outlook add-ins, users can see the icon in the EAC after installing the add-in. For content and task pane add-ins, users can see the icon in the ribbon after installing the add-in. |
| [Resources](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/resources)   **Important** This element is available only when using add-in manifest version 1.1. | Users in each locale you specify can see string and icon resources that you specifically create for the add-in for that locale. |
| [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) | Users in each locale you specify can see a webpage that you specifically design for the add-in for that locale. |

**Note:** You can localize the description and display name for only the locales that Office supports. See [Language identifiers and OptionState Id values in Office 2013](http://technet.microsoft.com/en-us/library/cc179219.aspx) for a list of languages and locales for the current release of Office.

### Examples

For example, an Office Add-in can specify the [DefaultLocale](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/defaultlocale.md) as en-us. For the [DisplayName](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/displayname.md) element, the add-in can specify an [Override](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/override.md) child element for the locale fr-fr, as shown below.

<DefaultLocale>en-us</DefaultLocale>

...

<DisplayName DefaultValue="Video player">

<Override Locale="fr-fr" Value="Lecteur vidéo" />

</DisplayName>

**Note:** If you need to localize for more than one area within a language family, such as de-de and de-at, we recommend that you use separate Override elements for each area. Using just the language name alone, in this case, de, is not supported across all combinations of Office host applications and platforms.

This means that the add-in assumes the en-us locale by default. Users see the English display name of "Video player" for all locales unless the client computer's locale is fr-fr, in which case users would see the French display name "Lecteur vidéo".

**Note:** You may only specify a single override per language, including for the default locale. For example, if your default locale is en-us you cannot not specify an override for en-us as well.

The following example applies a locale override for the [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md) element. It first specifies a default locale of en-us and an English description, and then specifies an [Override](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/override.md) statement with a French description for the fr-fr locale:

<DefaultLocale>en-us</DefaultLocale>

...

<Description DefaultValue=

"Watch YouTube videos referenced in the emails you receive

without leaving your email client.">

<Override Locale="fr-fr" Value=

"Visualisez les vidéos YouTube référencées dans vos courriers

électronique directement depuis Outlook et Outlook Web App."/>

</Description>

This means that the add-in assumes the en-us locale by default. Users would see the English description in the DefaultValueattribute for all locales unless the client computer's locale is fr-fr, in which case they would see the French description.

In the following example, the add-in specifies a separate image that's more appropriate for the fr-fr locale and culture. Users see the image DefaultLogo.png by default, except when the locale of the client computer is fr-fr. In this case, users would see the image FrenchLogo.png.

<!-- Replace "domain" with a real web server name and path. -->

<IconUrl DefaultValue="https://<domain>/DefaultLogo.png"/>

<Override Locale="fr-fr" Value="https://<domain>/FrenchLogo.png"/>

The following example shows how to localize a resource in the Resources section. It applies a locale override for an image that is more appropriate for the ja-jp culture.

<Resources>

<bt:Images>

<bt:Image id="icon1\_16x16" DefaultValue="https://www.contoso.com/icon\_default.png">

<bt:Override Locale="ja-jp" Value="https://www.contoso.com/ja-jp16-icon\_default.png" />

</bt:Image>

...

For the [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) element, supporting additional locales means providing a separate source HTML file for each of the specified locales. Users in each locale you specify can see a customized webpage that you design for that them.

For Outlook add-ins, the [SourceLocation](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/sourcelocation.md) element also aligns to the form factor. This allows you to provide a separate, localized source HTML file for each corresponding form factor. You can specify one or more [Override](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/override.md) child elements in each applicable settings element ([DesktopSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/desktopsettings.md), [TabletSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/tabletsettings.md), or [PhoneSettings](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/phonesettings.md)). The following example shows settings elements for the desktop, tablet, and smartphone form factors, each with one HTML file for the default locale and another for the French locale.

<DesktopSettings>

<SourceLocation DefaultValue="https://contoso.com/Desktop.html">

<Override Locale="fr-fr" Value="https://contoso.com/fr/Desktop.html" />

</SourceLocation>

<RequestedHeight>250</RequestedHeight>

</DesktopSettings>

<TabletSettings>

<SourceLocation DefaultValue="https://contoso.com/Tablet.html">

<Override Locale="fr-fr" Value="https://contoso.com/fr/Tablet.html" />

</SourceLocation>

<RequestedHeight>200</RequestedHeight>

</TabletSettings>

<PhoneSettings>

<SourceLocation DefaultValue="https://contoso.com/Mobile.html">

<Override Locale="fr-fr" Value="https://contoso.com/fr/Mobile.html" />

</SourceLocation>

</PhoneSettings>

## Match date/time format with client locale

You can get the locale of the user interface of the hosting application by using the [displayLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.displaylanguage.md) property. You can then display date and time values in a format consistent with the current locale of the host application. One way to do that is to prepare a resource file that specifies the date/time display format to use for each locale that your Office Add-in supports. At run time, your add-in can use the resource file and match the appropriate date/time format with the locale obtained from the [displayLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.displaylanguage.md) property.

You can get the locale of the data of the hosting application by using the [contentLanguage](https://github.com/OfficeDev/office-js-docs/blob/master/reference/shared/office.context.contentlanguage.md) property. Based on this value, you can then appropriately interpret or display date/time strings. For example, the jp-JP locale expresses data/time values as yyyy/MM/dd, and the fr-FR locale, dd/MM/yyyy.

## Use Ajax for globalization and localization

If you use Visual Studio to create Office Add-ins, the .NET Framework and Ajax provide ways to globalize and localize client script files.

You can globalize and use the [Date](http://msdn.microsoft.com/library/caf98d32-2de2-4704-8198-692350343681.aspx) and [Number](http://msdn.microsoft.com/library/c216d3a1-12ae-47d1-bca1-c3666d04572f.aspx) JavaScript type extensions and the JavaScript [Date](http://msdn.microsoft.com/library/ce2202bb-7ec9-4f5a-bf48-3a04feff283e.aspx) object in the JavaScript code for an Office Add-in to display values based on the locale settings on the current browser. For more information, see [Walkthrough: Globalizing a Date by Using Client Script](http://msdn.microsoft.com/library/69b34e6d-d590-4d03-a763-b7ae54b47d74.aspx).

You can include localized resource strings directly in standalone JavaScript files to provide client script files for different locales, which are set on the browser or provided by the user. Create a separate script file for each supported locale. In each script file, include an object in JSON format that contains the resource strings for that locale. The localized values are applied when the script runs in the browser.

## Example: Build a localized Office Add-in

This section provides examples that show you how to localize an Office Add-in description, display name, and UI.

To run the sample code provided, configure Microsoft Office 2013 on your computer to use additional languages so that you can test your add-in by switching the language used for display in menus and commands, for editing and proofing, or both.

Also, you'll need to create a Visual Studio 2015 Office Add-in project.

**Note:** To download Visual Studio 2015, see the [Office Developer Tools page](https://www.visualstudio.com/features/office-tools-vs). This page also has a link for the Office Developer Tools.

### Configure Office 2013 to use additional languages for display or editing

You can use an Office 2013 Language pack to install an additional language. For more information about Language Packs and where to get them, see [Office 2013 Language Options](http://office.microsoft.com/en-us/language-packs/).

**Note:** If you are an MSDN Subscriber, you might already have the Office 2013 Language Packs available to you. To determine whether your subscription offers Office 2013 Language Packs for download, go to [MSDN Subscriptions Home](https://msdn.microsoft.com/subscriptions/manage/), enter Office 2013 Language Pack in **Software downloads**, choose **Search**, and then select **Products available with my subscription**. Under **Language**, select the check box for the Language Pack you want to download, and then choose **Go**.

After you install the Language Pack, you can configure Office 2013 to use the installed language for display in the UI, for editing document content, or both. The example in this article uses an installation of Office 2013 that has the Spanish Language Pack applied.

### Create an Office Add-in project

1. In Visual Studio, choose **File** > **New Project**.
2. In the **New Project** dialog box, under **Templates**, expand **Visual Basic** or **Visual C#**, expand **Office/SharePoint**, and then choose **Office Add-ins**.
3. Choose **Office Add-in**, and then name your add-in, for example WorldReadyAddIn. Choose **OK**.
4. In the **Create Office Add-in** dialog box, select **Task pane** and choose **Next**. On the next page, clear the check boxes for all host applications except Word. Choose **Finish** to create the project.

### Localize the text used in your add-in

The text that you want to localize for another language appears in two areas:

* **Add-in display name and description**. This is controlled by entries in the add-in manifest file.
* **Add-in UI**. You can localize the strings that appear in your add-in UI by using JavaScript codeâ€”for example, by using a separate resource file that contains the localized strings.

To localize the add-in display name and description:

1. In **Solution Explorer**, expand **WorldReadyAddIn**, **WorldReadyAddInManifest**, and then choose **WorldReadyAddIn.xml**.
2. In WorldReadyAddInManifest.xml, replace the [DisplayName](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/displayname.md) and [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md) elements with the following block of code:.

**Note:** You can replace the Spanish language localized strings used in this example for the [DisplayName](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/displayname.md) and [Description](https://github.com/OfficeDev/office-js-docs/blob/master/reference/manifest/description.md) elements with the localized strings for any other language.

<DisplayName DefaultValue="World Ready add-in">

<Override Locale="es-es" Value="Aplicación de uso internacional"/>

</DisplayName>

<Description DefaultValue="An add-in for testing localization">

<Override Locale="es-es" Value="Una aplicación para la prueba de la localización"/>

</Description>

1. When you change the display language for Office 2013 from English to Spanish, for example, and then run the add-in, the add-in display name and description are shown with localized text.

To lay out the add-in UI:

1. In Visual Studio, in **Solution Explorer**, choose **Home.html**.
2. Replace the HTML in Home.html with the following HTML.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<meta http-equiv="X-UA-Compatible" content="IE=Edge" />

<title></title>

<script src="../../Scripts/jquery-1.8.2.js" type="text/javascript"></script>

<link href="../../Content/Office.css" rel="stylesheet" type="text/css" />

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

<!-- To enable offline debugging using a local reference to Office.js, use: -->

<!-- <script src="../../Scripts/Office/MicrosoftAjax.js" type="text/javascript"></script> -->

<!-- <script src="../../Scripts/Office/1.0/office.js" type="text/javascript"></script> -->

<link href="../App.css" rel="stylesheet" type="text/css" />

<script src="../App.js" type="text/javascript"></script>

<link href="Home.css" rel="stylesheet" type="text/css" />

<script src="Home.js" type="text/javascript"></script> <body>

<!-- Page content -->

<div id="content-header">

<div class="padding">

<h1 id="greeting"></h1>

</div>

</div>

<div id="content-main">

<div class="padding">

<div>

<p id="about"></p>

</div>

</div>

</div>

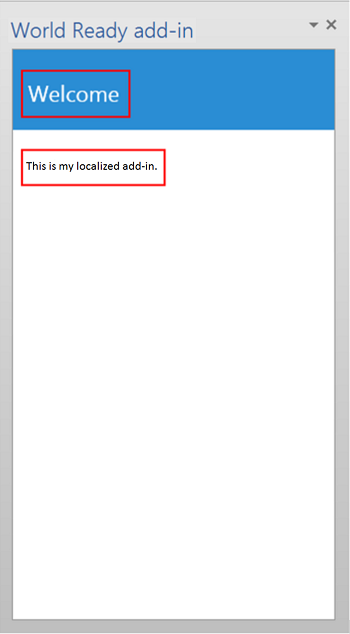
</head>

</html>

1. In Visual Studio, choose **File**, **Save AddIn\Home\Home.html**.

Figure 3 shows the heading (h1) element and the paragraph (p) element that will display localized text when your sample add-in runs.

**Figure 3. The add-in UI**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15App_HowToLocalize_fig03.png)

#### Add the resource file that contains the localized strings

The JavaScript resource file contains the strings used for the add-in UI. The sample add-in UI has an h1 element that displays a greeting, and a p element that introduces the add-in to the user.

To enable localized strings for the heading and paragraph, you place the strings in a separate resource file. The resource file creates a JavaScript object that contains a separate JavaScript Object Notation (JSON) object for each set of localized strings. The resource file also provides a method for getting back the appropriate JSON object for a given locale.

To add the resource file to the add-in project:

1. In **Solution Explorer** in Visual Studio, choose the **Add-in** folder in the web project for the sample add-in, and choose **Add**> **JavaScript file**.
2. In the **Specify Name for Item** dialog box, enterUIStrings.js.
3. Add the following code to the UIStrings.js file.

/\* Store the locale-specific strings \*/

var UIStrings = (function ()

{

"use strict";

var UIStrings = {};

// JSON object for English strings

UIStrings.EN =

{

"Greeting": "Welcome",

"Introduction": "This is my localized add-in."

};

// JSON object for Spanish strings

UIStrings.ES =

{

"Greeting": "Bienvenido",

"Introduction": "Esta es mi aplicación localizada."

};

UIStrings.getLocaleStrings = function (locale)

{

var text;

// Get the resource strings that match the language.

switch (locale)

{

case 'en-US':

text = UIStrings.EN;

break;

case 'es-ES':

text = UIStrings.ES;

break;

default:

text = UIStrings.EN;

break;

}

return text;

};

return UIStrings;

})();

The UIStrings.js resource file creates an object, **UIStrings**, which contains the localized strings for your add-in UI.

#### Localize the text used for the add-in UI

To use the resource file in your add-in, you'll need to add a script tag for it on Home.html. When Home.html is loaded, UIStrings.js executes and the **UIStrings** object that you use to get the strings is available to your code. Add the following HTML in the head tag for Home.html to make **UIStrings** available to your code.

<!-- Resource file for localized strings: -->

<script src="../UIStrings.js" type="text/javascript"></script>

Now you can use the **UIStrings** object to set the strings for the UI of your add-in.

If you want to change the localization for your add-in based on what language is used for display in menus and commands in the host application, you use the **Office.context.displayLanguage** property to get the locale for that language. For example, if the host application language uses Spanish for display in menus and commands, the **Office.context.displayLanguage**property will return the language code es-ES.

If you want to change the localization for your add-in based on what language is being used for editing document content, you use the **Office.context.contentLanguage** property to get the locale for that language. For example, if the host application language uses Spanish for editing document content, the **Office.context.contentLanguage** property will return the language code es-ES.

After you know the language the host application is using, you can use **UIStrings** to get the set of localized strings that matches the host application language.

Replace the code in the Home.js file with the following code. The code shows how you can change the strings used in the UI elements on Home.html based on either the display language of the host application or the editing language of the host application.

**Note:** To switch between changing the localization of the add-in based on the language used for editing, uncomment the line of code var myLanguage = Office.context.contentLanguage; and comment out the line of code var myLanguage = Office.context.displayLanguage;

/// <reference path="../App.js" />

/// <reference path="../UIStrings.js" />

(function () {

"use strict";

// The initialize function must be run each time a new page is loaded.

Office.initialize = function (reason)

{

$(document).ready(function () {

app.initialize();

// Get the language setting for editing document content.

// To test this, uncomment the following line and then comment out the

// line that uses Office.context.displayLanguage.

// var myLanguage = Office.context.contentLanguage;

// Get the language setting for UI display in the host application.

var myLanguage = Office.context.displayLanguage;

var UIText;

// Get the resource strings that match the language.

// Use the UIStrings object from the UIStrings.js file

// to get the JSON object with the correct localized strings.

UIText = UIStrings.getLocaleStrings(myLanguage);

// Set localized text for UI elements.

$("#greeting").text(UIText.Greeting);

$("#about").text(UIText.Instruction);

});

};

})();

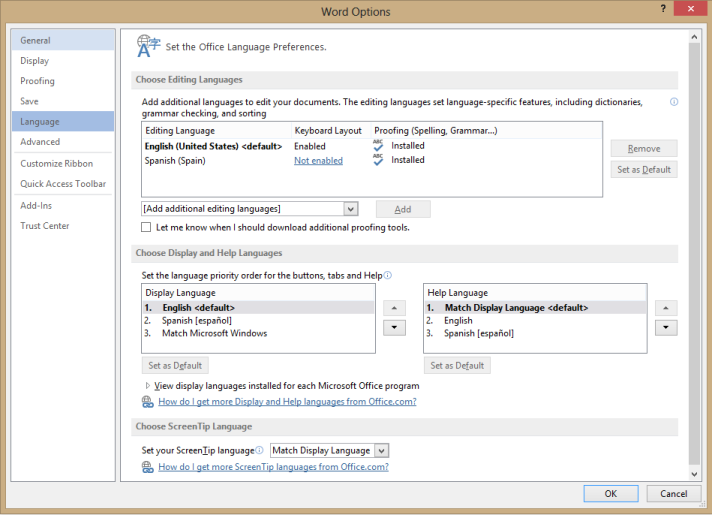
### Test your localized add-in

To test your localized add-in, change the language used for display or editing in the host application and then run your add-in.

To change the language used for display or editing in your add-in:

1. In Word 2013, choose **File**, **Options**, **Language**. Figure 4 shows the **Word Options** dialog box opened to the Language tab.

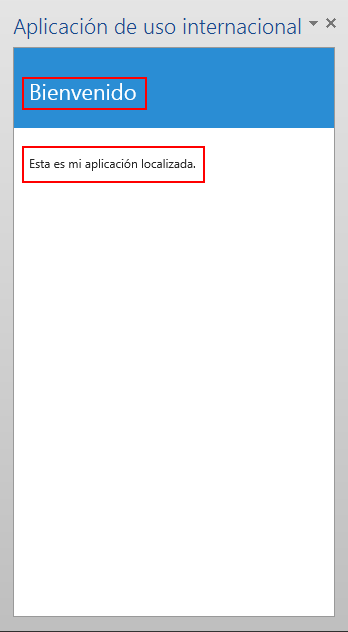
**Figure 4. Language options in the Word 2013 Options dialog box**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15App_HowToLocalize_fig04.png)

1. Under **Choose Display and Help Languages**, select the language that you want for display, for example Spanish, and then choose the up arrow to move the Spanish language to the first position in the list. Alternatively, to change the language used for editing, under **Choose editing languages**, choose the language you want to use for editing, for example, Spanish, and then choose **Set as Default**.
2. Choose **OK** to confirm your selection, and then close Word.

Run the sample add-in. The taskpane add-in loads in Word 2013, and the strings in the add-in UI change to match the language used by the host application, as shown in Figure 5.

**Figure 5. Add-in UI with localized text**

[](https://github.com/OfficeDev/office-js-docs/blob/master/images/off15App_HowToLocalize_fig05.png)

# Resource limits and performance optimization for Office Add-ins

To create the best experience for your users, ensure that your Office Add-in performs within specific limits for CPU core and memory usage, reliability and, for Outlook add-ins, the response time for evaluating regular expressions. These run-time resource usage limits apply to add-ins running on Office clients for Windows and OS X, but not Office Online, Outlook Web App,or OWA for Devices. You can also optimize the performance of your add-ins on desktop and mobile devices by optimizing the use of resources in your add-in design and implementation.

## Resource usage limits for add-ins

Run-time resource usage limits apply to all types of Office Add-ins. These limits help ensure performance for your users and mitigate denial-of-service attacks. Be sure to test your Office Add-in on your target host application using range of possible data and measure its performance against the following run-time usage limits:

* **CPU core usage** - A single CPU core usage threshold of 90%, observed three times in default 5-second intervals.

The default interval for a host rich client to check CPU core usage is every 5 seconds. If the host client detects the CPU core usage of an add-in is above the threshold value, it displays a message asking if the user wants to continue running the add-in. If the user chooses to continue, the host client does not ask the user again during that edit session. Administrators might want to use the **AlertInterval** registry key to raise the threshold to reduce the display of this warning message if users run CPU-intensive add-ins.

* **Memory usage** - A default memory usage threshold that is dynamically determined based on the available physical memory of the device.

By default, when a host rich client detects that physical memory usage on a device exceeds 80% of the available memory, the client starts monitoring the add-in's memory usage, at a document level for content and task pane add-ins, and at a mailbox level for Outlook add-ins. At a default interval of 5 seconds, the client warns the user if physical memory usage for a set of add-ins at the document or mailbox level exceeds 50%. This memory usage limit uses physical rather than virtual memory to ensure performance on devices with limited RAM, such as tablets. Administrators can override this dynamic setting with an explicit limit by using the **MemoryAlertThreshold** Windows registry key as a global setting, ir adjust the alert interval by using the **AlertInterval** key as a global setting.

* **Crash tolerance** - A default limit of four crashes for an add-in.

Administrators can adjust the threshold for crashes by using the **RestartManagerRetryLimit** registry key.

* **Application blocking** - Prolonged unresponsiveness threshold of 5 seconds for an add-in.

This affects the user's experiences of the add-in and the host application. When this occurs, the host application automatically restarts all the active add-ins for a document or mailbox (where applicable), and warns the user as to which add-in became unresponsive. Add-ins can reach this threshold when they do not regularly yield processing while performing long-running tasks. There are techniques to ensure that blocking does not occur. Administrators cannot override this threshold.

**Outlook add-ins**

If any Outlook add-in exceeds the preceding thresholds for CPU core or memory usage, or tolerance limit for crashes, Outlook disables the add-in. The Exchange Admin Center displays the disabled status of the app.

**Note** Even though only the Outlook rich clients and not Outlook Web App or OWA for Devices monitor resource usage, if a rich client disables an Outlook add-in, that add-in is also disabled for use in Outlook Web App and OWA for Devices.

In addition to the CPU core, memory, and reliability rules, Outlook add-ins should observe the following rules on activation:

* + **Regular expressions response time** - A default threshold of 1,000 milliseconds for Outlook to evaluate all regular expressions in the manifest of an Outlook add-in. Exceeding the threshold causes Outlook to retry evaluation at a later time.

Using a group policy or application-specific setting in the Windows registry, administrators can adjust this default threshold value of 1,000 milliseconds in the \*\*OutlookActivationAlertThreshold\*\* setting. For more information, see [Overriding resource usage settings for performance of Office Add-ins](http://msdn.microsoft.com/library/da14ec8c-5075-4035-a951-fc3c2b15c04b%28Office.15%29.aspx).

* **Regular expressions re-evaluation** - A default limit of three times for Outlook to reevaluate all the regular expressions in a manifest. If evaluation fails all three times by exceeding the applicable threshold (which is either the default of 1,000 milliseconds or a value specified by **OutlookActivationAlertThreshold**, if that setting exists in the Windows registry), Outlook disables the Outlook add-in. The Exchange Admin Center displays the disabled status, and the add-in is disabled for use in the Outlook rich clients, Outlook Web App and OWA for Devices.

Using a group policy or application-specific setting in the Windows registry, administrators can adjust this number of times to retry evaluation in the \*\*OutlookActivationManagerRetryLimit\*\* setting. For more information, see [Overriding resource usage settings for performance of Office Add-ins](http://msdn.microsoft.com/library/da14ec8c-5075-4035-a951-fc3c2b15c04b%28Office.15%29.aspx).

\*\*Task pane and content add-ins\*\*

If any content or task pane add-in exceeds the preceding thresholds on CPU core or memory usage, or tolerance limit for crashes, the corresponding host application displays a warning for the user. At this point, the user can do one of the following:

* Restart the add-in.
* Cancel further alerts about exceeding that threshold. Ideally, the user should then delete the add-in from the document; continuing the add-in would risk further performance and stability issues.

## Verifying resource usage issues in the Telemetry Log

Office provides a Telemetry Log that maintains a record of certain events (loading, opening, closing, and errors) of Office solutions running on the local computer, including resource usage issues in an Office Add-in. If you have the Telemetry Log set up, you can use Excel to open the Telemetry Log in the following default location on your local drive:

%Users%\ <Current user > \AppData\Local\Microsoft\Office\15.0\Telemetry

For each event that the Telemetry Log tracks for an add-in, there is a date/time of the occurrence, event ID, severity, and short descriptive title for the event, the friendly name and unique ID of the add-in, and the application that logged the event. You can refresh the Telemetry Log to see the current tracked events. The following table shows examples of Outlook add-ins that were tracked in the Telemetry log.

| **Date/Time** | **Event ID** | **Severity** | **Title** | **File** | **ID** | **Application** |
| --- | --- | --- | --- | --- | --- | --- |
| 10/8/2012 5:57:10 PM | 7 |  | add-in manifest downloaded successfully | Who's Who | 69cc567c-6737-4c49-88dd-123334943a22 | Outlook |
| 10/8/2012 5:57:01 PM | 7 |  | add-in manifest downloaded successfully | LinkedIn | 333bf46d-7dad-4f2b-8cf4-c19ddc78b723 | Outlook |
| The following table lists the events that the Telemetry Log tracks for Office Add-ins in general. |  |  |  |  |  |  |

| **Event ID** | **Title** | **Severity** | **Description** |
| --- | --- | --- | --- |
| 7 | Add-in manifest downloaded successfully |  | The manifest of the Office Add-in was successfully loaded and read by the host application. |
| 8 | Add-in manifest did not download | Critical | The host application was unable to load the manifest file for the Office Add-in from the SharePoint catalog, corporate catalog, or the Office Store. |
| 9 | Add-in markup could not be parsed | Critical | The host application loaded the Office Add-in manifest, but could not read the HTML markup of the app. |
| 10 | Add-in used too much CPU | Critical | The Office Add-in used more than 90% of the CPU resources over a finite period of time. |
| 15 | Add-in disabled due to string search time-out |  | Outlook add-ins search the subject line and message of an e-mail to determine whether they should be displayed by using a regular expression. The Outlook add-in listed in the **File** column was disabled by Outlook because it timed out repeatedly while trying to match a regular expression. |
| 18 | Add-in closed successfully |  | The host application was able to close the Office Add-in successfully. |
| 19 | Add-in encountered runtime error | Critical | The Office Add-in had a problem that caused it to fail. For more details, look at the **Microsoft Office Alerts** log using the Windows Event Viewer on the computer that encountered the error. |
| 20 | Add-in failed to verify licensing | Critical | The licensing information for the Office Add-in could not be verified and may have expired. For more details, look at the **Microsoft Office Alerts** log using the Windows Event Viewer on the computer that encountered the error. |
| For more information, see [Deploying Telemetry Dashboard](http://msdn.microsoft.com/en-us/library/f69cde72-689d-421f-99b8-c51676c77717%28Office.15%29.aspx) and [Troubleshooting Office files and custom solutions with the telemetry log](http://msdn.microsoft.com/library/ef88e30e-7537-488e-bc72-8da29810f7aa%28Office.15%29.aspx) |  |  |  |

## Design and implementation techniques

While the resources limits on CPU and memory usage, crash tolerance, UI responsiveness apply to Office Add-ins running only on the rich clients, optimizing the usage of these resources and battery should be a priority if you want your add-in to perform satisfactorily on all supporting clients and devices. Optimization is particularly important if your add-in carries out long-running operations or handles large data sets. The following list suggests some techniques to break up CPU-intensive or data-intensive operations into smaller chunks so that your add-in can avoid excessive resource consumption and the host application can remain responsive:

* In a scenario where your add-in needs to read a large volume of data from an unbounded dataset, you can apply paging when reading the data from a table, or reduce the size of data in each shorter read operation, rather than attempting to complete the read in one single operation.

For a JavaScript and jQuery code sample that shows breaking up a potentially long-running and CPU-intensive series of inputting and outputting operations on unbounded data, see [How can I give control back (briefly) to the browser during intensive JavaScript processing?](http://stackoverflow.com/questions/210821/how-can-i-give-control-back-briefly-to-the-browser-during-intensive-javascript). This example uses the [setTimeout](http://msdn.microsoft.com/en-us/library/ie/ms536753%28v=vs.85%29.aspx) method of the global object to limit the duration of input and output. It also handles the data in defined chunks instead of randomly unbounded data.

* If your add-in uses a CPU-intensive algorithm to process a large volume of data, you can use web workers to perform the long-running task in the background while running a separate script in the foreground, such as displaying progress in the user interface. Web workers do not block user activities and allow the HTML page to remain responsive. For an example of web workers, see [The Basics of Web Workers](https://www.html5rocks.com/en/tutorials/workers/basics/). See [Web Workers](http://msdn.microsoft.com/en-us/library/IE/hh772807%28v=vs.85%29.aspx) for more information about the Internet Explorer Web Workers API.
* If your add-in uses a CPU-intensive algorithm but you can divide the data input or output into smaller sets, consider creating a web service, passing the data to the web service to off-load the CPU, and wait for an asynchronous callback.
* Test your add-in against the highest volume of data you expect, and restrict your add-in to process up to that limit.