Office JavaScript API reference documentation

06/13/2025

An add-in can use the Office JavaScript APIs to interact with objects in Office client applications.

- **Application-specific** APIs provide strongly-typed objects that can be used to interact with objects that are native to a specific Office application.
- **Common** APIs can be used to access features such as UI, dialogs, and client settings that are common across multiple Office applications.

You should use application-specific APIs whenever feasible, and use Common APIs only for scenarios that aren't supported by application-specific APIs. For more detailed information about these two API models, see Develop Office Add-ins.

API reference

Excel API reference

JavaScript APIs for building Excel add-ins

Outlook API reference

JavaScript APIs for building Outlook add-ins

Word API reference

JavaScript APIs for building Word add-ins

PowerPoint API reference

JavaScript APIs for building PowerPoint add-ins

OneNote API reference

JavaScript APIs for building OneNote add-ins

Common API reference

JavaScript APIs that can be used by any Office Add-in

Note: There's currently no application-specific JavaScript API for Project; you'll use Common APIs to create Project add-ins.

Understanding the Office JavaScript API

Article • 05/18/2023

An Office Add-in can use the Office JavaScript APIs to interact with content in the Office document where the add-in is running.

Accessing the Office JavaScript API library

The Office JavaScript API library can be accessed via the Office JS content delivery network (CDN) at: `https://appsforoffice.microsoft.com/lib/1/hosted/office.js`. To use Office JavaScript APIs within any of your add-in's web pages, you must reference the CDN in a `<script>` tag in the `<head>` tag of the page.

HTML

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

> **Note**
>
> To use preview APIs, reference the preview version of the Office JavaScript API library on the CDN: `https://appsforoffice.microsoft.com/lib/beta/hosted/office.js`.

For more information about accessing the Office JavaScript API library, including how to get IntelliSense, see Referencing the Office JavaScript API library from its content delivery network (CDN).

API models

The Office JavaScript API includes two distinct models:

- **Application-specific** APIs provide strongly-typed objects that can be used to interact with objects that are native to a specific Office application. For example, you can use the Excel JavaScript APIs to access worksheets, ranges, tables, charts, and more. Application-specific APIs are currently available for the following Office applications.

    - Excel
    - OneNote
    - PowerPoint
    - Word

    This API model uses promises and allows you to specify multiple operations in each request you send to the Office application. Batching operations in this manner can significantly improve add-in performance in Office applications on the web. Application-specific APIs were introduced with Office 2016.

    > **Note**
    >
    > There's also an application-specific API for **Visio**, but you can use it only in SharePoint Online pages to interact with Visio diagrams that have been embedded in the page. Office Web Add-ins are not supported in Visio.

    See Using the application-specific API model to learn more about this API model.

- **Common** APIs can be used to access features such as UI, dialogs, and client settings that are common across multiple types of Office applications. This API model uses callbacks, which allow you to specify only one operation in each request sent to the Office application. Common APIs were introduced with Office 2013 and can be used to interact with any supported Office applications. For details about the Common API object model, which includes APIs for interacting with Outlook, PowerPoint, and Project, see Common JavaScript API object model.

> **Note**
>
> Custom functions without a **shared runtime** run in a **JavaScript-only runtime** that prioritizes execution of calculations. These functions use a slightly different programming model.

API requirement sets

Requirement sets are named groups of API members. Requirement sets can be specific to Office applications, such as the `ExcelApi 1.7` requirement set (a set of APIs that can only be used in Excel), or common to multiple applications, such as the `DialogApi 1.1` requirement set (a set of APIs that can be used in any Office application that supports the Dialog API).

Your add-in can use requirement sets to determine whether the Office application supports the API members that it needs to use. For more information about this, see Specify Office applications and API requirements.

Requirement set support varies by Office application, version, and platform. For detailed information about the platforms, requirement sets, and Common APIs that each Office application supports, see Office client application and platform availability for Office Add-ins.

> **Note**
>
> If you plan to **publish** your add-in to AppSource and make it available within the Office experience, make sure that you conform to the **Commercial marketplace certification policies**. 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 1120.3** and the **Office Add-in application and availability page**).

See also

- Office JavaScript API reference
- Loading the DOM and runtime environment
- Referencing the Office JavaScript API library
- Initialize your Office Add-in
- Runtimes in Office Add-ins

Application-specific API model

Article • 02/12/2025

This article describes how to use the API model for building add-ins in Excel, OneNote, PowerPoint, Visio, and Word. It introduces core concepts that are fundamental to using the promise-based APIs.

> **Note**
>
> This model isn't supported by Outlook or Project clients. Use the Common API model to work with those applications. For full platform availability notes, see Office client application and platform availability for Office Add-ins.

> **Tip**
>
> The examples in this page use the Excel JavaScript APIs, but the concepts also apply to OneNote, PowerPoint, Visio, and Word JavaScript APIs. For complete code samples that show how you could use these and other concepts in various Office applications, see Office Add-in code samples.

Asynchronous nature of the promise-based APIs

Office Add-ins are websites which appear inside a webview control within Office applications, such as Excel. This control is embedded within the Office application on desktop-based platforms, such as Office on Windows, and runs inside an HTML iframe in Office on the web. Due to performance considerations, the Office.js APIs cannot interact synchronously with the Office applications across all platforms. Therefore, the `sync()` API call in Office.js returns a Promise that is resolved when the Office application completes the requested read or write actions. Also, you can queue up multiple actions, such as setting properties or invoking methods, and run them as a batch of commands with a single call to `sync()`, rather than sending a separate request for each action. The following sections describe how to accomplish this using the `run()` and `sync()` APIs.

*.run function

`Excel.run`, `OneNote.run`, `PowerPoint.run`, and `Word.run` execute a function that specifies the actions to perform against Excel, Word, and OneNote. `*.run` automatically creates a request context that you can use to interact with Office objects. When `*.run` completes, a promise is resolved, and any objects that were allocated at runtime are automatically released.

The following example shows how to use `Excel.run`. The same pattern is also used with OneNote, PowerPoint, Visio, and Word.

JavaScript

```
Excel.run(function (context) {
    // Add your Excel JS API calls here that will be batched and sent to the
workbook.
    console.log('Your code goes here.');
}).catch(function (error) {
    // Catch and log any errors that occur within `Excel.run`.
    console.log('error: ' + error);
    if (error instanceof OfficeExtension.Error) {
        console.log('Debug info: ' + JSON.stringify(error.debugInfo));
    }
});
```

Request context

The Office application and your add-in run in different processes. Since they use different runtime environments, add-ins require a `RequestContext` object in order to connect your add-in to objects in Office such as worksheets, ranges, paragraphs, and tables. This `RequestContext` object is provided as an argument when calling `*.run`.

Proxy objects

The Office JavaScript objects that you declare and use with the promise-based APIs are proxy objects. Any methods that you invoke or properties that you set or load on proxy objects are simply added to a queue of pending commands. When you call the `sync()` method on the request context (for example, `context.sync()`), the queued commands are dispatched to the Office application and run. These APIs are fundamentally batch-centric. You can queue up as many changes as you wish on the request context, and then call the `sync()` method to run the batch of queued commands.

For example, the following code snippet declares the local JavaScript Excel.Range object, `selectedRange`, to reference a selected range in the Excel workbook, and then sets some

properties on that object. The `selectedRange` object is a proxy object, so the properties that are set and the method that is invoked on that object will not be reflected in the Excel document until your add-in calls `context.sync()`.

JavaScript

```
const selectedRange = context.workbook.getSelectedRange();
selectedRange.format.fill.color = "#4472C4";
selectedRange.format.font.color = "white";
selectedRange.format.autofitColumns();
```

Performance tip: Minimize the number of proxy objects created

Avoid repeatedly creating the same proxy object. Instead, if you need the same proxy object for more than one operation, create it once and assign it to a variable, then use that variable in your code.

JavaScript

```
// BAD: Repeated calls to .getRange() to create the same proxy object.
worksheet.getRange("A1").format.fill.color = "red";
worksheet.getRange("A1").numberFormat = "0.00%";
worksheet.getRange("A1").values = [[1]];

// GOOD: Create the range proxy object once and assign to a variable.
const range = worksheet.getRange("A1");
range.format.fill.color = "red";
range.numberFormat = "0.00%";
range.values = [[1]];

// ALSO GOOD: Use a "set" method to immediately set all the properties
// without even needing to create a variable!
worksheet.getRange("A1").set({
    numberFormat: [["0.00%"]],
    values: [[1]],
    format: {
        fill: {
            color: "red"
        }
    }
});
```

sync()

Calling the `sync()` method on the request context synchronizes the state between proxy objects and objects in the Office document. The `sync()` method runs any commands that are queued on the request context and retrieves values for any properties that should be loaded on the proxy objects. The `sync()` method executes asynchronously and returns a Promise, which is resolved when the `sync()` method completes.

The following example shows a batch function that defines a local JavaScript proxy object (`selectedRange`), loads a property of that object, and then uses the JavaScript promises pattern to call `context.sync()` to synchronize the state between proxy objects and objects in the Excel document.

JavaScript

```
await Excel.run(async (context) => {
    const selectedRange = context.workbook.getSelectedRange();
    selectedRange.load('address');
    await context.sync();
    console.log('The selected range is: ' + selectedRange.address);
});
```

In the previous example, `selectedRange` is set and its `address` property is loaded when `context.sync()` is called.

Since `sync()` is an asynchronous operation, you should always return the `Promise` object to ensure the `sync()` operation completes before the script continues to run. If you're using TypeScript or ES6+ JavaScript, you can `await` the `context.sync()` call instead of returning the promise.

Performance tip: Minimize the number of sync calls

In the Excel JavaScript API, `sync()` is the only asynchronous operation, and it can be slow under some circumstances, especially for Excel on the web. To optimize performance, minimize the number of calls to `sync()` by queueing up as many changes as possible before calling it. For more information about optimizing performance with `sync()`, see Avoid using the context.sync method in loops.

load()

Before you can read the properties of a proxy object, you must explicitly load the properties to populate the proxy object with data from the Office document, and then call `context.sync()`. For example, if you create a proxy object to reference a selected range, and then want to read the selected range's `address` property, you need to load

the `address` property before you can read it. To request properties of a proxy object be loaded, call the `load()` method on the object and specify the properties to load. The following example shows the `Range.address` property being loaded for `myRange`.

JavaScript

```
await Excel.run(async (context) => {
    const sheetName = 'Sheet1';
    const rangeAddress = 'A1:B2';
    const myRange =
context.workbook.worksheets.getItem(sheetName).getRange(rangeAddress);

    myRange.load('address');
    await context.sync();

    console.log (myRange.address);   // ok
    //console.log (myRange.values);  // not ok as it was not loaded

    console.log('done');
});
```

> **Note**
>
> If you're only calling methods or setting properties on a proxy object, you don't need to call the `load()` method. The `load()` method is only required when you want to read properties on a proxy object.

Just like requests to set properties or invoke methods on proxy objects, requests to load properties on proxy objects get added to the queue of pending commands on the request context, which will run the next time you call the `sync()` method. You can queue up as many `load()` calls on the request context as necessary.

Scalar and navigation properties

There are two categories of properties: **scalar** and **navigational**. Scalar properties are assignable types such as strings, integers, and JSON structs. Navigation properties are read-only objects and collections of objects that have their fields assigned, instead of directly assigning the property. For example, `name` and `position` members on the Excel.Worksheet object are scalar properties, whereas `protection` and `tables` are navigation properties.

Your add-in can use navigational properties as a path to load specific scalar properties. The following code queues up a `load` command for the name of the font used by an

`Excel.Range` object, without loading any other information.

JavaScript

```
someRange.load("format/font/name")
```

You can also set the scalar properties of a navigation property by traversing the path. For example, you could set the font size for an `Excel.Range` by using `someRange.format.font.size = 10;`. You don't need to load the property before you set it.

Please be aware that some of the properties under an object may have the same name as another object. For example, `format` is a property under the `Excel.Range` object, but `format` itself is an object as well. So, if you make a call such as `range.load("format")`, this is equivalent to `range.format.load()` (an undesirable empty `load()` statement). To avoid this, your code should only load the "leaf nodes" in an object tree.

Load from a collection

When working with a collection, use `load` on the collection to load properties for every object in the collection. Use `load` exactly as you would for an individual object in that collection.

The following sample code shows the `name` property being loaded and logged for every chart in the "Sample" worksheet.

JavaScript

```
await Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getItem("Sample");
    const chartCollection = sheet.charts;

    // Load the name property on every chart in the chart collection.
    chartCollection.load("name");
    await context.sync();

    chartCollection.items.forEach((chart) => {
        console.log(chart.name);
    });
});
```

You normally don't include the `items` property of the collection in the `load` arguments. All the items are loaded if you load any item properties. However, if you will be looping

over the items in the collection, but don't need to load any particular property of the items, you need to `load` the `items` property.

The following sample code shows the `name` property being set for every chart in the "Sample" worksheet.

JavaScript

```
await Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getItem("Sample");
    const chartCollection = sheet.charts;

    // Load the items property from the chart collection to set properties
on individual charts.
    chartCollection.load("items");
    await context.sync();

    chartCollection.items.forEach((chart, index) => {
        chart.name = `Sample chart ${index}`;
    });
});
```

Calling `load` without parameters (not recommended)

If you call the `load()` method on an object (or collection) without specifying any parameters, all scalar properties of the object or the collection's objects will be loaded. Loading unneeded data will slow down your add-in. You should always explicitly specify which properties to load.

> **ⓘ Important**
>
> The amount of data returned by a parameter-less `load` statement can exceed the size limits of the service. To reduce the risks to older add-ins, some properties are not returned by `load` without explicitly requesting them. The following properties are excluded from such load operations.
>
> - `Excel.Range.numberFormatCategories`

ClientResult

Methods in the promise-based APIs that return primitive types have a similar pattern to the `load`/`sync` paradigm. As an example, `Excel.TableCollection.getCount` gets the number of tables in the collection. `getCount` returns a `ClientResult<number>`, meaning

the `value` property in the returned ClientResult is a number. Your script can't access that value until `context.sync()` is called.

The following code gets the total number of tables in an Excel workbook and logs that number to the console.

JavaScript

```
const tableCount = context.workbook.tables.getCount();

// This sync call implicitly loads tableCount.value.
// Any other ClientResult values are loaded too.
await context.sync();

// Trying to log the value before calling sync would throw an error.
console.log (tableCount.value);
```

set()

Setting properties on an object with nested navigation properties can be cumbersome. As an alternative to setting individual properties using navigation paths as described above, you can use the `object.set()` method that is available on objects in the promise-based JavaScript APIs. With this method, you can set multiple properties of an object at once by passing either another object of the same Office.js type or a JavaScript object with properties that are structured like the properties of the object on which the method is called.

The following code sample sets several format properties of a range by calling the `set()` method and passing in a JavaScript object with property names and types that mirror the structure of properties in the `Range` object. This example assumes that there is data in range **B2:E2**.

JavaScript

```
await Excel.run(async (context) => {
    const sheet = context.workbook.worksheets.getItem("Sample");
    const range = sheet.getRange("B2:E2");
    range.set({
        format: {
            fill: {
                color: '#4472C4'
            },
            font: {
                name: 'Verdana',
                color: 'white'
            }
```

```
        }
    });
    range.format.autofitColumns();

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

Some properties cannot be set directly

Some properties cannot be set, despite being writable. These properties are part of a parent property that must be set as a single object. This is because that parent property relies on the subproperties having specific, logical relationships. These parent properties must be set using object literal notation to set the entire object, instead of setting that object's individual subproperties. One example of this is found in PageLayout. The `zoom` property must be set with a single PageLayoutZoomOptions object, as shown here.

JavaScript

```
// PageLayout.zoom.scale must be set by assigning PageLayout.zoom to a
PageLayoutZoomOptions object.
sheet.pageLayout.zoom = { scale: 200 };
```

In the previous example, you would ***not*** be able to directly assign `zoom` a value: `sheet.pageLayout.zoom.scale = 200;`. That statement throws an error because `zoom` is not loaded. Even if `zoom` were to be loaded, the set of scale will not take effect. All context operations happen on `zoom`, refreshing the proxy object in the add-in and overwriting locally set values.

This behavior differs from navigational properties like Range.format. Properties of `format` can be set using object navigation, as shown here.

JavaScript

```
// This will set the font size on the range during the next
`content.sync()`.
range.format.font.size = 10;
```

You can identify a property that cannot have its subproperties directly set by checking its read-only modifier. All read-only properties can have their non-read-only subproperties directly set. Writeable properties like `PageLayout.zoom` must be set with an object at that level. In summary:

- Read-only property: Subproperties can be set through navigation.

- Writable property: Subproperties cannot be set through navigation (must be set as part of the initial parent object assignment).

*OrNullObject methods and properties

Some accessor methods and properties throw an exception when the desired object doesn't exist. For example, if you attempt to get an Excel worksheet by specifying a worksheet name that isn't in the workbook, the `getItem()` method throws an `ItemNotFound` exception. The application-specific libraries provide a way for your code to test for the existence of document entities without requiring exception handling code. This is accomplished by using the `*OrNullObject` variations of methods and properties. These variations return an object whose `isNullObject` property is set to `true`, if the specified item doesn't exist, rather than throwing an exception.

For example, you can call the `getItemOrNullObject()` method on a collection such as **Worksheets** to retrieve an item from the collection. The `getItemOrNullObject()` method returns the specified item if it exists; otherwise, it returns an object whose `isNullObject` property is set to `true`. Your code can then evaluate this property to determine whether the object exists.

> **Note**
>
> The `*OrNullObject` variations do not ever return the JavaScript value `null`. They return ordinary Office proxy objects. If the the entity that the object represents does not exist then the `isNullObject` property of the object is set to `true`. Do not test the returned object for nullity or falsity. It is never `null`, `false`, or `undefined`.

The following code sample attempts to retrieve an Excel worksheet named "Data" by using the `getItemOrNullObject()` method. If a worksheet with that name does not exist, a new sheet is created. Note that the code does not load the `isNullObject` property. Office automatically loads this property when `context.sync` is called, so you do not need to explicitly load it with something like `dataSheet.load('isNullObject')`.

JavaScript

```
await Excel.run(async (context) => {
    let dataSheet = context.workbook.worksheets.getItemOrNullObject("Data");

    await context.sync();

    if (dataSheet.isNullObject) {
```

```
        dataSheet = context.workbook.worksheets.add("Data");
    }

    // Set `dataSheet` to be the second worksheet in the workbook.
    dataSheet.position = 1;
});
```

Undo support

Undo is partially supported by the application-specific Office JavaScript APIs. This means that users may be able to revert changes made by add-ins through the undo command. If a particular API doesn't support undo, the application's undo stack is cleared. This means that you won't be able to undo the effects of that API or anything prior to calling that API.

API support for undo is continuing to expand but is currently incomplete. We advise against designing your add-in in such a way that it relies on undo support.

See also

- Common JavaScript API object model
- Resource limits and performance optimization for Office Add-ins

Common JavaScript API object model

Article • 03/21/2023

> **Important**
>
> This article applies to the **Common APIs**, the Office JavaScript API model that was introduced with Office 2013. These APIs include features such as UI, dialogs, and client settings that are common across multiple types of Office applications. Outlook add-ins exclusively use Common APIs, especially the subset of APIs exposed through the Mailbox object.
>
> You should only use Common APIs for scenarios that aren't supported by **application-specific APIs**. To learn when to use Common APIs instead of application-specific APIs, see Understanding the Office JavaScript API.

Office JavaScript APIs give access to the Office client application's underlying functionality. Most of this access goes through a few important objects. The Context object gives access to the runtime environment after initialization. The Document object gives the user control over an Excel, PowerPoint, or Word document. The Mailbox object gives an Outlook add-in access to messages, appointments, and user profiles. Understanding the relationships between these high-level objects is the foundation of an Office Add-in.

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 object. The **Context** is the main object that provides access to the most important objects of the API, such as the Document and Mailbox 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 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 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 and displayLanguage properties that let you determine the locale (language) used in the document or item, or by the Office application. The roamingSettings property lets you access the members of the RoamingSettings object, which stores settings specific to your add-in for individual users' mailboxes. Finally, the **Context** object provides a ui 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 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 abstract class. For more information about creating task pane add-ins for Project, see Task pane add-ins for Project.

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 property of the `Context` object. The `Document` object defines common data access methods 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 Office JavaScript API 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. Three main data types are supported.

Expand table

Data type	Description	Host application support
Text	Provides a string representation of the data in the selection or binding.	In Excel, Project, and PowerPoint, only plain text is supported. In Word, 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 and PowerPoint 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**	Matrix data access is supported only in Excel and Word.

Data type	Description	Host application support
	values in two columns would be `[['a', 'b'], ['c', 'd']]`, and a single column of three rows would be `[['a'], ['b'], ['c']]`.	
Table	Provides the data in the selection or binding as a TableData object. The `TableData` object exposes the data through the `headers` and `rows` properties.	Table data access is supported only in Excel and Word.

Data type coercion

The data access methods on the `Document` and Binding objects support specifying the desired data type using the *coercionType* parameter of these methods, and corresponding CoercionType 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 the *coercionType* 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 property in the callback returns `"failed"`, and you can use the AsyncResult.error property to access an

Error object with information about why the method call failed.

Work 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.

Work 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, addFromSelectionAsync, or addFromNamedItemAsync. 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 object exposes a getAllAsync 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 method or Office.select function. You can establish new bindings as well as remove existing ones by

using one of the following methods of the `Bindings` object: addFromSelectionAsync, addFromPromptAsync, addFromNamedItemAsync, or releaseByIdAsync.

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.

Expand table

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 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, TableBinding, or TextBinding. All three of these objects inherit the getDataAsync and setDataAsync 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.

Work with custom XML parts using the CustomXmlParts and CustomXmlPart objects

Applies to: Task pane add-ins for Word

The CustomXmlParts and CustomXmlPart 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 code sample.

Work with the entire document using the getFileAsync method

Applies to: Task pane add-ins for Word and PowerPoint

The Document.getFileAsync method and members of the File and Slice 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 Get the whole document from an add-in for PowerPoint or Word.

Mailbox object

Applies to: Outlook add-ins

Outlook add-ins primarily use a subset of the API exposed through the Mailbox object. To access the objects and members specifically for use in Outlook add-ins, such as the Item object, use the mailbox property of the **Context** object to access the **Mailbox** object, as shown in the following line of code.

JavaScript

```
// Access the Item object.
const 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.

Asynchronous programming in Office Add-ins

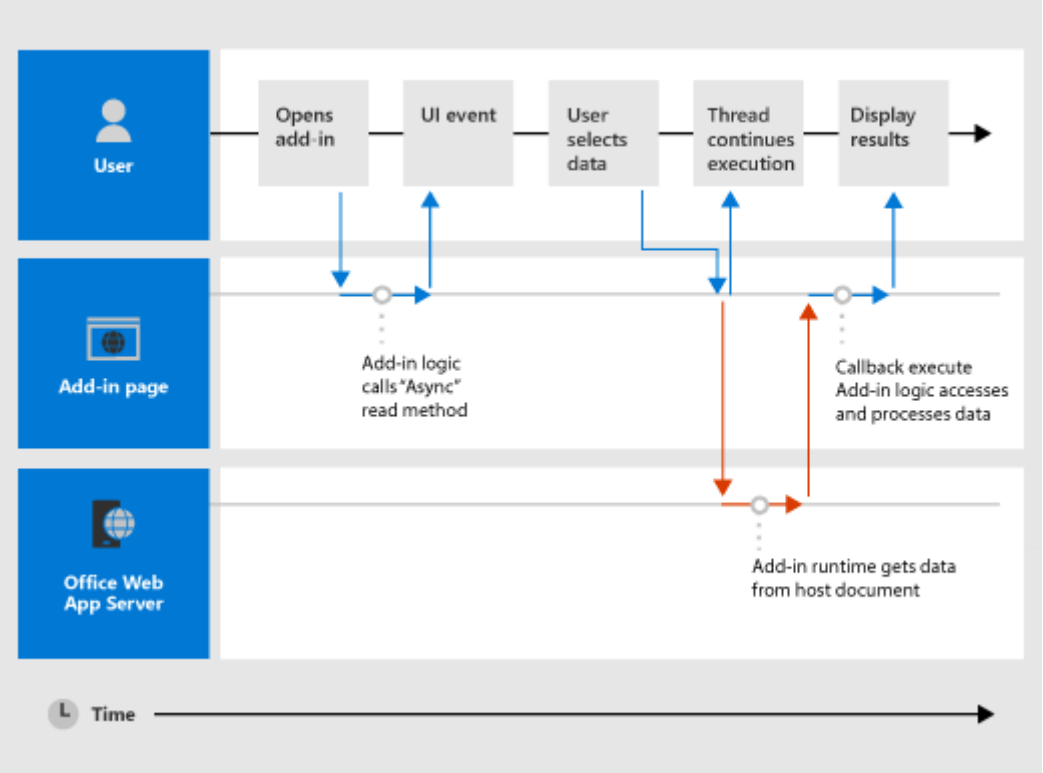
Article • 12/31/2024

> **Important**
>
> This article applies to the **Common APIs**, the Office JavaScript API model that was introduced with Office 2013. These APIs include features such as UI, dialogs, and client settings that are common across multiple types of Office applications. Outlook add-ins exclusively use Common APIs, especially the subset of APIs exposed through the Mailbox object.
>
> You should only use Common APIs for scenarios that aren't supported by **application-specific APIs**. To learn when to use Common APIs instead of application-specific APIs, see Understanding the Office JavaScript API.

Why does the Office Add-ins API use asynchronous programming? JavaScript is a single-threaded language. If a script invokes a long-running synchronous process of the Office client, all subsequent scripts are blocked until that process completes. Being asynchronous makes sure that Office Add-ins are responsive and fast.

The names of all asynchronous methods in the Common APIs end with "Async", such as the `Document.getSelectedDataAsync`, `Binding.getDataAsync`, or `Item.loadCustomPropertiesAsync` methods. When an "Async" method is called, it runs immediately. The rest of the script continues while the operation completes on the client-side. The optional callback function that you pass to an "Async" method runs as soon as the data or requested operation is ready. This generally occurs promptly, but there can be a slight delay.

The following diagram shows the flow of an "Async" method that reads the data the user selected in a document. When the "Async" call is made, the JavaScript thread is free to perform any additional client-side processing (although none is shown in the diagram). When the "Async" method returns, the callback resumes on the thread. The add-in can then access data, do something with it, and display the result. The pattern is the same across platforms.



Write 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. The add-in runtime uses that parameter to provide access to an AsyncResult object for the callback function.

The callback function can either be an anonymous function or a named function. 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.

Write an anonymous callback function

The following anonymous callback function declares a single parameter named `result` for the data returned by the client. It retrieves and writes that data from the AsyncResult.value property when the callback returns.

JavaScript

```
function (result) {
    write('Selected data: ' + result.value);
}
```

The following example shows this anonymous callback function in the context of a full "Async" method call to the `Document.getSelectedDataAsync(coercionType, callback)` 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 inline to the method. When the function runs, it uses the *result* parameter to access the `value` property of the `AsyncResult` object. It then displays the data selected by the user in the document.

JavaScript

```
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 property to determine if the call succeeded or failed. If your call failed, use the AsyncResult.error property to access an Error object to help decide what to do.

For more information on the `getSelectedDataAsync` method, see Read and write data to the active selection in a document or spreadsheet.

Write a named callback function

Alternatively, you can write a named function and pass its name to the *callback* parameter of an "Async" method. Here, the previous example is rewritten to pass a function named `writeDataCallback` as the *callback* parameter.

JavaScript

```
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 functions 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, CustomXmlPart, Document, RoamingSettings, and Settings objects) are used to add event handler functions. The `AsyncResult.value` property in those callback functions always returns **undefined**, since no data or object is accessed when you add an event handler.

On the other hand, if you call the `Document.getSelectedDataAsync` method, it returns the data the user selected in the document as the `AsyncResult.value` property in the callback. Or, if you call the Bindings.getAllAsync method, it returns an array of all of the `Binding` objects in the document.

For a description of what's returned to the `AsyncResult.value` property for an `Async` method, see the **`callback`** section of that method's reference topic.

Asynchronous programming patterns

The Common APIs in the Office JavaScript API support two kinds of asynchronous programming patterns.

- Nested callbacks
- Promises

> ⓘ **Note**
>
> In the current version of the Office JavaScript API, *built-in* support for the promises pattern only works with code for bindings in Excel spreadsheets and Word

documents. However, you can wrap other functions that have callbacks inside your own custom `Promise`-returning function. For more information, see Wrap Common APIs in Promise-returning functions.

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 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 method.
- Finally, the `result2` parameter of the callback passed to the `Binding.getDataAsync` method is used to display the data in the binding.

JavaScript

```
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 Common APIs.

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 the script 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 completes the request. An *onError* handler is provided to cover situations when the request can't be fulfilled.

The Common APIs provide the Office.select function to support the promises pattern when working with existing binding objects. The promise object returned to the `Office.select` function only supports the four methods directly accessible from the Binding object.

- getDataAsync
- setDataAsync
- addHandlerAsync
- removeHandlerAsync

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 in the document or spreadsheet (using one of the "addFrom" methods of the `Bindings` collection: `addFromNamedItemAsync`, `addFromPromptAsync`, or `addFromSelectionAsync`). The example *selectorExpression* of `bindings#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`. This is used to access an `Error` object if the `select` function fails to access the specified binding. The following example shows a basic error handler function that can be passed to the *onError* parameter.

JavaScript

```
function onError(result){
    const 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. In that case, you must use the nested callback function pattern.

After a `Binding` object promise is fulfilled, it can be reused in the chained method call as if it were a binding. If it's successful, 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 example uses the `select` function to retrieve a binding with the `id` "`cities`" from the `Bindings` collection, and then calls the addHandlerAsync method to add an event handler for the dataChanged event of the binding.

JavaScript

```
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` function 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.

Pass 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. These are passed in as a JavaScript object. The object that contains the optional parameters is an *unordered* collection of key-value pairs. You can create the object that contains optional parameters inline, or by creating an `options` object and passing that in as the *options* parameter.

Pass optional parameters inline

Here is an example of the Document.setSelectedDataAsync method with optional parameters defined inline. The two optional parameters, *coercionType* and *asyncContext*, are defined as an anonymous JavaScript object.

JavaScript

```
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;
}
```

Pass optional parameters in a named object

Alternatively, you can create a named object that specifies the optional parameters separately from the method call and then pass the object as the *options* argument. The following example shows one way of creating an `options` object, where `parameter1`, `value1`, and so on, are placeholders for the actual parameter names and values.

JavaScript

```
const options = {
    parameter1: value1,
    parameter2: value2,
    ...
    parameterN: valueN
};
```

Which looks like the following example when used to specify the ValueFormat and FilterType parameters.

JavaScript

```
const options = {
    valueFormat: "unformatted",
    filterType: "all"
};
```

Here's another way of creating the `options` object.

JavaScript

```
const 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:

JavaScript

```
const options = {};
options["ValueFormat"] = "unformatted";
options["FilterType"] = "all";
```

The following example shows how to call to the `Document.setSelectedDataAsync` method by specifying optional parameters in an `options` object.

JavaScript

```
const 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 JavaScript 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.

Wrap Common APIs in `Promise`-returning functions

The Common API (and Outlook API) methods do not return Promises. Therefore, you cannot use await to pause the execution until the asynchronous operation completes. If you need `await` behavior, wrap the method call in an explicitly created `Promise`.

The basic pattern is to create an asynchronous method that returns a Promise object immediately and *resolves* that Promise object when the inner method completes, or *rejects* the object if the method fails. The following is a simple example.

JavaScript

```
function getDocumentFilePath() {
    return new OfficeExtension.Promise(function (resolve, reject) {
        try {
            Office.context.document.getFilePropertiesAsync(function
(asyncResult) {
                resolve(asyncResult.value.url);
            });
        }
        catch (error) {
            reject(WordMarkdownConversion.errorHandler(error));
        }
    })
}
```

When this function needs to be awaited, it can be either called with the `await` keyword or passed to a `then` function.

> ⓘ Note

This technique is especially useful when you need to call a Common API inside a call of the `run` function in an application-specific object model. For an example of the `getDocumentFilePath` function being used in this way, see the file Home.js in the sample Word-Add-in-JavaScript-MDConversion.

See also

- Understanding the Office JavaScript API
- Office JavaScript API
- Runtimes in Office Add-ins

Office JavaScript API support for content and task pane add-ins

Article • 02/12/2025

> **ⓘ Important**
>
> This article applies to the **Common APIs**, the Office JavaScript API model that was introduced with Office 2013. These APIs include features such as UI, dialogs, and client settings that are common across multiple types of Office applications. Outlook add-ins exclusively use Common APIs, especially the subset of APIs exposed through the Mailbox object.
>
> You should only use Common APIs for scenarios that aren't supported by **application-specific APIs**. To learn when to use Common APIs instead of application-specific APIs, see Understanding the Office JavaScript API.

You can use the Office JavaScript API to create task pane or content add-ins for Office client 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, Context, and AsyncResult. 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 object. A content or task pane add-in can use the Office.context.document 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 and CustomXmlParts objects, and the getSelectedDataAsync, setSelectedDataAsync, and getFileAsync methods. The `Document` object also provides the mode property for determining whether a document is read-only or in edit mode, the url property to get the URL of the current document, and access to the Settings object. The `Document` object also supports adding event handlers for the SelectionChanged event, so you can detect when a user changes their 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 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.

3. **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 object to create or get bindings, and the methods and properties of the Binding object to work with data.
    - The CustomXmlParts, CustomXmlPart and associated objects to create and manipulate custom XML parts in Word documents.
    - The File and Slice 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 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 client applications, see Understanding the Office JavaScript API.

Read and write to an active selection in a document, spreadsheet, or presentation

You can read or write to the user's current selection in a document, spreadsheet, or presentation. Depending on the Office application for your add-in, you can specify the type of data structure to read or write as a parameter in the getSelectedDataAsync and setSelectedDataAsync methods of the Document 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.

JavaScript

```
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.

Bind 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, addFromPromptAsync, or addFromSelectionAsync methods of the Bindings 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.

JavaScript

```
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.

Get entire documents

If your task pane add-in runs in PowerPoint or Word, you can use the Document.getFileAsync, File.getSliceAsync, and File.closeAsync methods to get an entire presentation or document.

When you call `Document.getFileAsync` you get a copy of the document in a File object. The `File` object provides access to the document in "chunks" represented as Slice 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 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.

Read and write 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 property,CustomXmlParts, CustomXmlPart, and CustomXmlNode 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 method.

You can also use the CustomXmlParts.getByIdAsync method to access custom XML parts by their GUIDs. After getting a custom XML part, use the CustomXmlPart.getXmlAsync 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 method.

For detailed information about how to manage custom XML parts with a task pane add-in, see Understand when and how to use Office Open XML in your Word add-in.

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 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 method. Internally, the data is stored in a serialized JSON object as name/value pairs. You use the get, set, and remove 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'.

JavaScript

```
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.

Permissions model and governance

Your add-in uses the app manifest to request permission to access the level of functionality it requires from the Office JavaScript API. The method varies depending on the type of manifest.

- **Unified manifest for Microsoft 365**: Use the "authorization.permissions.resourceSpecific" property. For example, if your add-in requires read/write access to the document, its manifest must specify `Document.ReadWrite.User` as the value in its "authorization.permissions.resourceSpecific.name" property. The following example requests the **read document** permission, which allows only methods that can read (but not write to) the document.

  JSON

  ```
  "authorization": {
      "permissions": {
        "resourceSpecific": [
          ...
          {
            "name": "Document.Read.User",
            "type": "Delegated"
          },
        ]
      }
  },
  ```

  > **Note**
  >
  > The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

- **Add-in only manifest**: Use the `Permissions` element in the manifest 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 **read document** permission in a task pane's manifest.

XML

```xml
<?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 add-ins.

See also

- Office JavaScript API
- Schema reference for Office Add-ins manifests
- Troubleshoot user errors with Office Add-ins
- Runtimes in Office Add-ins

Bind to regions in a document or spreadsheet

08/01/2025

> **Important**
>
> This article applies to the **Common APIs**, the Office JavaScript API model that was introduced with Office 2013. These APIs include features such as UI, dialogs, and client settings that are common across multiple types of Office applications. Outlook add-ins exclusively use Common APIs, especially the subset of APIs exposed through the Mailbox object.
>
> You should only use Common APIs for scenarios that aren't supported by **application-specific APIs**. To learn when to use Common APIs instead of application-specific APIs, see Understanding the Office JavaScript API.

Bindings let your add-in consistently access specific regions of a document or spreadsheet. Think of a binding as a bookmark that remembers a specific location, even if users change their selection or navigate elsewhere in the document. Specifically, here are what bindings offer your add-in.

- **Access common data structures** across supported Office applications, such as tables, ranges, or text.
- **Read and write data** without requiring users to make a selection first.
- **Create persistent relationships** between your add-in and document data. Bindings are saved with the document and work across sessions.

To create a binding, call one of these Bindings object methods to associate a document region with a unique identifier: addFromPromptAsync, addFromSelectionAsync, or addFromNamedItemAsync. Once you've established the binding, use its identifier to read from or write to that region anytime.

You can also subscribe to data and selection change events for specific bound regions. This means your add-in only gets notified about changes within the bound area, not the entire document.

Choose the right binding type

Office supports three different types of bindings. You specify the type with the *bindingType* parameter when creating a binding using addFromSelectionAsync, addFromPromptAsync, or

addFromNamedItemAsync.

Text Binding

Text Binding - Binds to a document region that can be represented as text.

In Word, most contiguous selections work. In Excel, only single cell selections can use text binding. Excel supports only plain text, while Word supports three formats: plain text, HTML, and Open XML for Office.

Matrix Binding

Matrix Binding - Binds to a fixed region containing tabular data without headers.

Data in a matrix binding is read or written as a two-dimensional **Array** (an array of arrays in JavaScript). For example, two rows of **string** values in two columns would look like `[['a', 'b'], ['c', 'd']]`, and a single column of three rows would be `[['a'], ['b'], ['c']]`.

In Excel, any contiguous selection of cells works for matrix binding. In Word, only tables support matrix binding.

Table Binding

Table Binding - Binds to a document region containing a table with headers.

Data in a table binding is read or written as a TableData object. The `TableData` object exposes 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, new rows or columns that users add to the table are automatically included in the binding.

After creating a binding with one of the three "addFrom" methods, you can work with the binding's data and properties using the corresponding object: MatrixBinding, TableBinding, or TextBinding. All three objects inherit the getDataAsync and setDataAsync methods from the `Binding` object for interacting with bound data.

> ⓘ **Note**
>
> **Should you use matrix or table bindings?** When working with tabular data that includes a total row, use matrix binding if your add-in needs to access values in the total row or detect when a user selects the total row. Table bindings don't include total rows in their

TableBinding.rowCount property or in the `rowCount` and `startRow` properties of BindingSelectionChangedEventArgs in event handlers. To work with total rows, you must use matrix binding.

Create a binding from the current selection

The following example adds a text binding called `myBinding` to the current selection using the addFromSelectionAsync method.

JavaScript

```
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;
}
```

In this example, the binding type is text, so a TextBinding is created for the selection. Different binding types expose different data and operations. Office.BindingType is an enumeration of available binding types.

The second optional parameter specifies the ID of the new binding. If you don't specify an ID, one is generated automatically.

The anonymous function passed as the final *callback* parameter runs when the binding creation is complete. The function receives a single parameter, `asyncResult`, which provides access to an AsyncResult object with the call's status. The `AsyncResult.value` property contains a reference to a Binding object of the specified type for the newly created binding. You can use this Binding object to get and set data.

Create a binding from a prompt

The following function adds a text binding called `myBinding` using the addFromPromptAsync method. This method lets users specify the range for the binding using the application's built-

in range selection prompt.

JavaScript

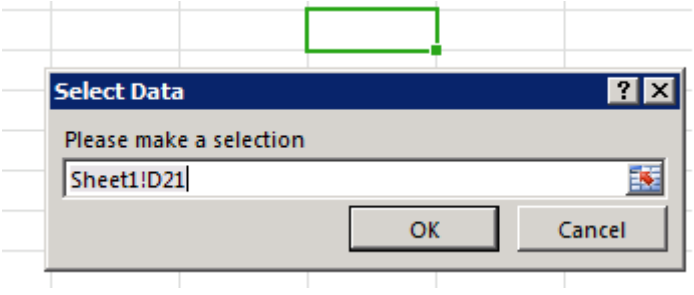
```
function bindFromPrompt() {
    Office.context.document.bindings.addFromPromptAsync(Office.BindingType.Text, {
id: 'myBinding' }, function (asyncResult) {
        if (asyncResult.status == Office.AsyncResultStatus.Failed) {
            write('Action failed. Error: ' + asyncResult.error.message);
        } 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;
}
```

In this example, the binding type is text, so a TextBinding is created for the user's selection in the prompt.

The second parameter contains the ID of the new binding. If you don't specify an ID, one is generated automatically.

The anonymous function passed as the third *callback* parameter runs when the binding creation is complete. When the callback function runs, the AsyncResult object contains the call's status and the newly created binding.

The following screenshot shows the built-in range selection prompt in Excel.



Add a binding to a named item

The following function adds a binding to the existing `myRange` named item as a "matrix" binding using the addFromNamedItemAsync method and assigns the binding's `id` as "myMatrix".

JavaScript

```
function bindNamedItem() {
    Office.context.document.bindings.addFromNamedItemAsync("myRange", "matrix",
{id:'myMatrix'}, function (result) {
        if (result.status == 'succeeded'){
            write('Added new binding with type: ' + result.value.type + ' and id:
' + result.value.id);
            }
        else
            write('Error: ' + result.error.message);
    });
}

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

For Excel, the `itemName` parameter of addFromNamedItemAsync refers to an existing named range, a range specified with A1 reference style (`"A1:A3"`), or a table. By default, Excel assigns the names "Table1" for the first table, "Table2" for the second table, and so on. To assign a meaningful name to a table in the Excel UI, use the **Table Name** property on the **Table Tools | Design** tab.

> ⓘ **Note**
>
> In Excel, when specifying a table as a named item, you must fully qualify the name to include the worksheet name in this format (e.g., `"Sheet1!Table1"`).

The following function creates a binding in Excel to the first three cells in column A (`"A1:A3"`), assigns the ID `"MyCities"`, and then writes three city names to that binding.

JavaScript

```
 function bindingFromA1Range() {
    Office.context.document.bindings.addFromNamedItemAsync("A1:A3", "matrix", {
id: "MyCities" },
        function (asyncResult) {
            if (asyncResult.status == "failed") {
                write('Error: ' + asyncResult.error.message);
            } else {
                // Write data to the new binding.
                Office.select("bindings#MyCities").setDataAsync([['Berlin'],
['Munich'], ['Duisburg']], { coercionType: "matrix" },
                    function (asyncResult) {
                        if (asyncResult.status == "failed") {
```

```
                            write('Error: ' + asyncResult.error.message);
                        }
                    });
            }
        });
}
// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

For Word, the `itemName` parameter of addFromNamedItemAsync refers to the `Title` property of a `Rich Text` content control. (You can't bind to content controls other than the `Rich Text` content control.)

By default, a content control has no `Title` value assigned. To assign a meaningful name in the Word UI, after inserting a **Rich Text** content control from the **Controls** group on the **Developer** tab, use the **Properties** command in the **Controls** group to display the **Content Control Properties** dialog. Then set the `Title` property of the content control to the name you want to reference from your code.

The following function creates a text binding in Word to a rich text content control named `"FirstName"`, assigns the **id** `"firstName"`, and then displays that information.

JavaScript

```
function bindContentControl() {
    Office.context.document.bindings.addFromNamedItemAsync('FirstName',
        Office.BindingType.Text, {id:'firstName'},
        function (result) {
            if (result.status === Office.AsyncResultStatus.Succeeded) {
                write('Control bound. Binding.id: '
                    + result.value.id + ' Binding.type: ' + result.value.type);
            } else {
                write('Error:', result.error.message);
            }
    });
}
// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

Get all bindings

The following example gets all bindings in a document using the getAllAsync method.

JavaScript

```
Office.context.document.bindings.getAllAsync(function (asyncResult) {
    let bindingString = '';
    for (let i in asyncResult.value) {
        bindingString += asyncResult.value[i].id + '\n';
    }
    write('Existing bindings: ' + bindingString);
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

The anonymous function passed as the `callback` parameter runs when the operation is complete. The function is called with a single parameter, `asyncResult`, which contains an array of the bindings in the document. The array is iterated to build a string that contains the IDs of the bindings. The string is then displayed in a message box.

Get a binding by ID using getByIdAsync

The following example uses the getByIdAsync method to get a binding in a document by specifying its ID. This example assumes that a binding named `'myBinding'` was added to the document using one of the methods described earlier in this article.

JavaScript

```
Office.context.document.bindings.getByIdAsync('myBinding', function (asyncResult)
{
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
        write('Action failed. Error: ' + asyncResult.error.message);
    }
    else {
        write('Retrieved 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;
}
```

In this example, the first `id` parameter is the ID of the binding to retrieve.

The anonymous function passed as the second *callback* parameter runs when the operation is completed. The function is called with a single parameter, *asyncResult*, which contains the call's status and the binding with the ID "myBinding".

Get a binding by ID using `Office.select`

The following example uses the Office.select function to get a Binding object promise in a document by specifying its ID in a selector string. It then calls the getDataAsync method to get data from the specified binding. This example assumes that a binding named `'myBinding'` was added to the document using one of the methods described earlier in this article.

JavaScript

```
Office.select("bindings#myBinding", function onError(){}).getDataAsync(function
(asyncResult) {
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
        write('Action failed. Error: ' + asyncResult.error.message);
    } else {
        write(asyncResult.value);
    }
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

If the `select` function promise successfully returns a Binding object, that object exposes only the following four methods: getDataAsync, setDataAsync, addHandlerAsync, and removeHandlerAsync. If the promise can't return a Binding object, the `onError` callback can be used to access an asyncResult.error object to get more information. If you need to call a member of the Binding object other than the four methods exposed by the Binding object promise returned by the `select` function, instead use the getByIdAsync method by using the Document.bindings property and getByIdAsync method to retrieve the Binding object.

Release a binding by ID

The following example uses the releaseByIdAsync method to release a binding in a document by specifying its ID.

JavaScript

```
Office.context.document.bindings.releaseByIdAsync('myBinding', function
(asyncResult) {
```

```
    write('Released myBinding!');
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

In this example, the first `id` parameter is the ID of the binding to release.

The anonymous function passed as the second parameter is a callback that runs when the operation is complete. The function is called with a single parameter, asyncResult, which contains the call's status.

Read data from a binding

The following example uses the getDataAsync method to get data from an existing binding.

JavaScript

```
myBinding.getDataAsync(function (asyncResult) {
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
        write('Action failed. Error: ' + asyncResult.error.message);
    } else {
        write(asyncResult.value);
    }
});

// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

`myBinding` is a variable that contains an existing text binding in the document. Alternatively, you could use Office.select to access the binding by its ID, and start your call to the getDataAsync method, like this:

JavaScript

```
Office.select("bindings#myBindingID").getDataAsync
```

The anonymous function passed into the method is a callback that runs when the operation is complete. The AsyncResult.value property contains the data within `myBinding`. The type of the value depends on the binding type. The binding in this example is a text binding, so the value

will contain a string. For additional examples of working with matrix and table bindings, see the getDataAsync method topic.

Write data to a binding

The following example uses the setDataAsync method to set data in an existing binding.

JavaScript

```
myBinding.setDataAsync('Hello World!', function (asyncResult) { });
```

`myBinding` is a variable that contains an existing text binding in the document.

In this example, the first parameter is the value to set on `myBinding`. Because this is a text binding, the value is a `string`. Different binding types accept different types of data.

The anonymous function passed into the method is a callback that runs when the operation is complete. The function is called with a single parameter, `asyncResult`, which contains the result's status.

Detect changes to data or selection in a binding

The following function attaches an event handler to the DataChanged event of a binding with an ID of "MyBinding".

JavaScript

```
function addHandler() {
Office.select("bindings#MyBinding").addHandlerAsync(
    Office.EventType.BindingDataChanged, dataChanged);
}
function dataChanged(eventArgs) {
    write('Bound data changed in binding: ' + eventArgs.binding.id);
}
// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

The `myBinding` is a variable that contains an existing text binding in the document.

The first *eventType* parameter of addHandlerAsync specifies the name of the event to subscribe to. Office.EventType is an enumeration of available event type values. `Office.EventType.BindingDataChanged` evaluates to the string "bindingDataChanged".

The `dataChanged` function passed as the second *handler* parameter is an event handler that runs when the data in the binding is changed. The function is called with a single parameter, *eventArgs*, which contains a reference to the binding. This binding can be used to retrieve the updated data.

Similarly, you can detect when a user changes selection in a binding by attaching an event handler to the SelectionChanged event of a binding. To do that, specify the `eventType` parameter of addHandlerAsync as `Office.EventType.BindingSelectionChanged` or `"bindingSelectionChanged"`.

You can add multiple event handlers for a given event by calling addHandlerAsync again and passing in an additional event handler function for the `handler` parameter. The name of each event handler function must be unique.

Remove an event handler

To remove an event handler for an event, call removeHandlerAsync passing in the event type as the first *eventType* parameter, and the name of the event handler function to remove as the second *handler* parameter. For example, the following function removes the `dataChanged` event handler function added in the previous section's example.

JavaScript

```
function removeEventHandlerFromBinding() {
    Office.select("bindings#MyBinding").removeHandlerAsync(
        Office.EventType.BindingDataChanged, {handler:dataChanged});
}
```

> **ⓘ Important**
>
> If the optional *handler* parameter is omitted when **removeHandlerAsync** is called, all event handlers for the specified `eventType` will be removed.

See also

- Understanding the Office JavaScript API
- Asynchronous programming in Office Add-ins
- Read and write data to the active selection in a document or spreadsheet

Referencing the Office JavaScript API library

Article • 01/16/2025

The Office JavaScript API library provides the APIs that your add-in can use to interact with the Office application. The simplest way to reference the library is to use the content delivery network (CDN) by adding the following `<script>` tag within the `<head>` section of your HTML page.

HTML

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

This will download and cache the Office JavaScript API 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.

> **Important**
>
> You must reference the Office JavaScript API from inside the `<head>` section of the page to ensure that the API is fully initialized prior to any body elements.

Office.js-specific web API behavior

Office.js replaces the default Window.history methods of `replaceState` and `pushState` with `null`. This is done to support older Microsoft webviews and Office versions. If your add-in relies on these methods and doesn't need to run on Office versions that use the Internet Explorer 11 browser control, replace the Office.js library reference with the following workaround.

HTML

```
<script type="text/javascript">
    // Cache the history method values.
    window._historyCache = {
        replaceState: window.history.replaceState,
        pushState: window.history.pushState
    };
```

```
</script>

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

<script type="text/javascript">
    // Restore the history method values after loading Office.js
    window.history.replaceState = window._historyCache.replaceState;
    window.history.pushState = window._historyCache.pushState;
</script>
```

Thank you to @stepper and the Stack Overflow community for suggesting and verifying this workaround.

API versioning and backward compatibility

In the previous HTML snippet, the `/1/` in front of `office.js` in the CDN URL specifies the latest incremental release within version 1 of Office.js. Because the Office JavaScript API maintains backward compatibility, the latest release will continue to support API members that were introduced earlier in version 1.

If you plan to publish your Office Add-in from AppSource, you must use this CDN reference. Local references are only appropriate for internal, development, and debugging scenarios.

> **Note**
>
> To use preview APIs, reference the preview version of the Office JavaScript API library on the CDN: `https://appsforoffice.microsoft.com/lib/beta/hosted/office.js`.

Enabling IntelliSense for a TypeScript project

In addition to referencing the Office JavaScript API as described previously, you can also enable IntelliSense for TypeScript add-in project by using the type definitions from DefinitelyTyped. To do so, run the following command in a Node-enabled system prompt (or git bash window) from the root of your project folder. You must have Node.js installed (which includes npm).

command line

```
npm install --save-dev @types/office-js
```

Preview APIs

New JavaScript APIs are first introduced in "preview" and later become part of a specific numbered requirement set after sufficient testing occurs and user feedback is acquired.

> **Note**
>
> Preview APIs are subject to change and are not intended for use in a production environment. We recommend that you try them out in test and development environments only. Do not use preview APIs in a production environment or within business-critical documents.
>
> To use preview APIs:
>
> - You must use the preview version of the Office JavaScript API library from the Office.js content delivery network (CDN). The type definition file for TypeScript compilation and IntelliSense is found at the CDN and DefinitelyTyped. You can install these types with `npm install --save-dev @types/office-js-preview`.
> - You may need to join the Microsoft 365 Insider program for access to more recent Office builds.

CDN references for other Microsoft 365 environments

21Vianet operates and manages an Office 365 service powered by licensed Microsoft technologies to provide Office 365 services for China compliant with local laws and regulations. Add-ins developed for use within this cloud environment should use corresponding CDN. Use `https://appsforoffice.cdn.partner.office365.cn/appsforoffice/lib/1/hosted/office.js` instead of the standard CDN reference. This ensures continued compliance and provides better add-in performance.

See also

- Understanding the Office JavaScript API
- Office JavaScript API
- Guidance for deploying Office Add-ins on government clouds
- Microsoft software license terms for the Microsoft Office JavaScript (Office.js) API library

Specify Office applications and API requirements with the unified manifest

Article • 02/12/2025

> **Note**
>
> For information about specifying requirements with the add-in only manifest, see Specify Office hosts and API requirements with the add-in only manifest.

Your Office Add-in might depend on a specific Office application (also called an Office host) or on specific members of the Office JavaScript Library (office.js). For example, your add-in might:

- Run in a single Office application (e.g., Word or Excel), or several applications.
- Make use of Office JavaScript APIs that are only available in some versions of Office. For example, the volume-licensed perpetual version of Excel 2016 doesn't support all Excel-related APIs in the Office JavaScript library.
- Be designed for use only in a mobile form factor.

In these situations, you need to ensure that your add-in is never installed on Office applications or Office versions in which it cannot run.

There are also scenarios in which you want to control which features of your add-in are visible to users based on their Office application and Office version. Three examples are:

- Your add-in has features that are useful in both Word and PowerPoint, such as text manipulation, but it has some additional features that only make sense in PowerPoint, such as slide management features. You need to hide the PowerPoint-only features when the add-in is running in Word.
- Your add-in has a feature that requires an Office JavaScript API method that is supported in some versions of an Office application, such as Microsoft 365 subscription Excel, but isn't supported in others, such as volume-licensed perpetual Excel 2016. But your add-in has other features that require only Office JavaScript API methods that *are* supported in volume-licensed perpetual Excel 2016. In this scenario, you need the add-in to be installable on that version of Excel 2016, but the feature that requires the unsupported method should be hidden from those users.
- Your add-in has features that are supported in desktop Office, but not in mobile Office.

This article helps you understand how 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 client application and platform availability for Office Add-ins page.

> **Tip**
>
> Many of the tasks described in this article are done for you, in whole or in part, when you create your add-in project with a tool, such as the Yeoman generator for Office Add-ins or one of the Office Add-in templates in Visual Studio. In such cases, please interpret the task as meaning that you should verify that it has been done.

Use the latest Office JavaScript API library

Your add-in should load the most current version of the Office JavaScript API library from the content delivery network (CDN). To do this, be sure you have the following `<script>` tag in the first HTML file your add-in opens. Using `/1/` in the CDN URL ensures that you reference the most recent version of Office.js.

HTML

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

Specify which Office applications can host your add-in

To specify the Office applications on which your add-in can be installed, use the "extensions.requirements.scopes" array. Specify any subset of "mail", "workbook", "document", and "presentation". The following table shows which Office application and platform combinations correspond to these values. It also shows what kind of add-in can be installed for each scope.

Expand table

Name	Office client applications	Available add-in types
document	Word on the web, Windows, Mac, iPad	Task pane
mail	Outlook on the web, Windows (new and classic), Android, iOS	Mail
presentation	PowerPoint on the web, Windows, Mac, iPad	Task pane, Content
workbook	Excel on the web, Windows, Mac, iPad	Task pane, Content

> **Note**
>
> Content add-ins have an "extensions.contentRuntimes" property. They can't have an "extensions.runtimes" property so they can't be combined with a Task pane or Mail add-in. For more information about Content add-ins, see Content Office Add-ins.

For example, the following JSON specifies that the add-in can install on any release of Excel, which includes Excel on the web, Windows, and iPad, but can't be installed on any other Office application.

JSON

```
"extensions": [
    {
        "requirements": {
            "scopes": [ "workbook" ],
        },
        ...
    }
]
```

> **Note**
>
> Office applications are supported on different platforms and run on desktops, web browsers, tablets, and mobile devices. You usually can't specify which platform can be used to run your add-in. For example, if you specify "workbook", both Excel on the web and on Windows can be used to run your add-in. However, if you specify "mail", your add-in won't run on Outlook mobile clients unless you define the mobile extension point.

Specify which Office APIs your add-in needs

You can't explicitly specify the Office versions and builds or the platforms on which your add-in should be installable, and you wouldn't want to because you would have to revise your manifest whenever support for the add-in features that your add-in uses is extended to a new version or platform. Instead, specify in the manifest the APIs that your add-in needs. Office prevents the add-in from being installed on combinations of Office version and platform that don't support the APIs and ensures that the add-in won't appear in **My Add-ins**.

> **ⓘ Important**
>
> Only use the "requirements" property that is a direct child of "extensions" to specify the API members that your add-in must have to be of any significant value at all. If your add-in uses an API for some features, but has other useful features that don't require the API, you should design the add-in so that it's installable on platform and Office version combinations that don't support the API but provides a diminished experience on those combinations. For this purpose, use "requirements" properties that aren't direct children of "extensions". For more information, see Design for alternate experiences.

Requirement sets

To simplify the process of specifying the APIs that your add-in needs, Office groups most APIs together in requirement sets. The APIs in the Common API Object Model are grouped by the development feature that they support. For example, all the APIs connected to table bindings are in the requirement set called "TableBindings 1.1". The APIs in the Application specific object models are grouped by when they were released for use in production add-ins.

Requirement sets are versioned. For example, the APIs that support Dialog Boxes are in the requirement set DialogApi 1.1. When additional APIs that enable messaging from a task pane to a dialog were released, they were grouped into DialogApi 1.2, along with all the APIs in DialogApi 1.1. *Each version of a requirement set is a superset of all earlier versions.*

Requirement set support varies by Office application, the version of the Office application, and the platform on which it is running. For example, DialogApi 1.2 isn't supported on volume-licensed perpetual versions of Office before Office 2021, but DialogApi 1.1 is supported on all perpetual versions back to Office 2016. You want your add-in to be installable on every combination of platform and Office version that

supports the APIs that it uses, so you should always specify in the manifest the *minimum* version of each requirement set that your add-in requires. Details about how to do this are later in this article.

> **Tip**
>
> For more information about requirement set versioning, see **Office requirement sets availability**, and for the complete lists of requirement sets and information about the APIs in each, start with **Office Add-in requirement sets**. The reference topics for most Office.js APIs also specify the requirement set they belong to (if any).

extensions.requirements.capabilities property

Use the "requirements.capabilities" property to specify the minimum requirement sets that must be supported by the Office application to install your add-in. If the Office application or platform doesn't support the requirement sets or API members specified in the "requirements.capabilities" property, the add-in won't run in that application or platform, and won't display in **My Add-ins**.

> **Tip**
>
> All APIs in the application-specific models are in requirement sets, but some of those in the Common API model aren't. If your add-in requires an API that isn't in any requirement set, you can implement a runtime check for the availability of the API and display a message to the add-in's users if it isn't supported. For more information, see **Check for API availability at runtime**.

The following code example shows how to configure an add-in that is installable in all Office application and platform combinations 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".

JSON

```
"extensions": [
    {
        "requirements": {
            "capabilities": [
                {
                    "name": "TableBindings",
```

```
                    "minVersion": "1.1"
                },
                {
                    "name": "OOXML",
                    "minVersion": "1.1"
                }
            ],
        },
        ...
    }
]
```

> **Tip**
>
> For more information and another example of using the "extensions.requirements" property, see the "extensions.requirements" section in Specify Office Add-in requirements in the unified manifest for Microsoft 365.

Specify the form factors on which your add-in can be installed

For an Outlook add-in, you can specify whether the add-in should be installable on desktop (includes tablets) or mobile form factors. To configure this, use the "extensions.requirements.formFactors" property. The following example show how to make the Outlook add-in installable on both form factors.

JSON

```
"extensions": [
    {
        "requirements": {
            ...
            "formFactors": [
                "desktop",
                "mobile"
            ]
        },
        ...
    }
]
```

Design for alternate experiences

The extensibility features that the Office Add-in platform provides can be usefully divided into three kinds:

- Extensibility features that are available immediately after the add-in is installed. An example of this kind of feature is Add-in Commands, which are custom ribbon buttons and menus.
- Extensibility features that are available only when the add-in is running and that are implemented with Office.js JavaScript APIs; for example, Dialog Boxes.
- Extensibility features that are available only at runtime but are implemented with a combination of Office.js JavaScript and manifest configuration. Examples of these are Excel custom functions, single sign-on, and custom contextual tabs.

If your add-in uses a specific extensibility feature for some of its functionality but has other useful functionality that doesn't require the extensibility feature, you should design the add-in so that it's installable on platform and Office version combinations that don't support the extensibility feature. It can provide a valuable, albeit diminished, experience on those combinations.

You implement this design differently depending on how the extensibility feature is implemented:

- For features implemented entirely with JavaScript, see Check for API availability at runtime.
- For features that require you to configure the manifest, see the "Filter features" section of Specify Office Add-in requirements in the unified manifest for Microsoft 365.

See also

- Office Add-ins manifest
- Office Add-in requirement sets
- Specify Office Add-in requirements in the unified manifest for Microsoft 365

Specify Office applications and API requirements with the add-in only manifest

Article • 02/12/2025

> **Note**
>
> For information about specifying requirements with the unified manifest for Microsoft 365, see Specify Office hosts and API requirements with the unified manifest.

Your Office Add-in might depend on a specific Office application (also called an Office host) or on specific members of the Office JavaScript Library (office.js). For example, your add-in might:

- Run in a single Office application (for example, Word or Excel), or several applications.
- Make use of Office JavaScript APIs that are only available in some versions of Office. For example, the volume-licensed perpetual version of Excel 2016 doesn't support all Excel-related APIs in the Office JavaScript library.

In these situations, you need to ensure that your add-in is never installed on Office applications or Office versions in which it cannot run.

There are also scenarios in which you want to control which features of your add-in are visible to users based on their Office application and Office version. Two examples are:

- Your add-in has features that are useful in both Word and PowerPoint, such as text manipulation, but it has some additional features that only make sense in PowerPoint, such as slide management features. You need to hide the PowerPoint-only features when the add-in is running in Word.
- Your add-in has a feature that requires an Office JavaScript API method that is supported in some versions of an Office application, such as Microsoft 365 subscription Excel, but is not supported in others, such as volume-licensed perpetual Excel 2016. But your add-in has other features that require only Office JavaScript API methods that *are* supported in volume-licensed perpetual Excel 2016. In this scenario, you need the add-in to be installable on that version of Excel 2016, but the feature that requires the unsupported method should be hidden from those users.

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 client application and platform availability for Office Add-ins page.

> **Tip**
>
> Many of the tasks described in this article are done for you, in whole or in part, when you create your add-in project with a tool, such as the Yeoman generator for Office Add-ins or one of the Office Add-in templates in Visual Studio. In such cases, please interpret the task as meaning that you should verify that it has been done.

Use the latest Office JavaScript API library

Your add-in should load the most current version of the Office JavaScript API library from the content delivery network (CDN). To do this, be sure you have the following `<script>` tag in the first HTML file your add-in opens. Using `/1/` in the CDN URL ensures that you reference the most recent version of Office.js.

HTML

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

Specify which Office applications can host your add-in

By default, an add-in is installable in all Office applications supported by the specified add-in type (that is, Mail, Task pane, or Content). For example, a task pane add-in is installable by default on Access, Excel, OneNote, PowerPoint, Project, and Word.

To ensure that your add-in is installable in only a subset of Office applications, use the Hosts and Host elements in the add-in only manifest.

For example, the following **<Hosts>** and **<Host>** declaration specifies that the add-in can install on any release of Excel, which includes Excel on the web, Windows, and iPad, but can't be installed on any other Office application.

XML

```xml
<Hosts>
  <Host Name="Workbook" />
</Hosts>
```

The **<Hosts>** element can contain one or more **<Host>** elements. There should be a separate **<Host>** element for each Office application on which the add-in should be installable. The `Name` attribute is required and can be set to one of the following values.

Expand table

Name	Office client applications	Available add-in types
Document	Word on the web, Windows, Mac, iPad	Task pane
Mailbox	Outlook on the web, Windows (new and classic), Mac, Android, iOS	Mail
Notebook	OneNote on the web	Task pane, Content
Presentation	PowerPoint on the web, Windows, Mac, iPad	Task pane, Content
Project	Project on Windows	Task pane
Workbook	Excel on the web, Windows, Mac, iPad	Task pane, Content
Database	Access (obsolete)	Task pane

Note

Office applications are supported on different platforms and run on desktops, web browsers, tablets, and mobile devices. You usually can't specify which platform can be used to run your add-in. For example, if you specify `Workbook`, both Excel on the web and on Windows can be used to run your add-in. However, if you specify `Mailbox`, your add-in won't run on Outlook mobile clients unless you define the mobile extension point.

Note

It isn't possible for an add-in only manifest to apply to more than one type: Mail, Task pane, or Content. This means that if you want your add-in to be installable on Outlook and on one of the other Office applications, you must create *two* add-ins, one with a Mail type manifest and the other with a Task pane or Content type manifest.

Specify which Office versions and platforms can host your add-in

You can't explicitly specify the Office versions and builds or the platforms on which your add-in should be installable, and you wouldn't want to because you would have to revise your manifest whenever support for the add-in features that your add-in uses is extended to a new version or platform. Instead, specify in the manifest the APIs that your add-in needs. Office prevents the add-in from being installed on combinations of Office version and platform that don't support the APIs and ensures that the add-in won't appear in **My Add-ins**.

> **Important**
>
> Only use the base manifest to specify the API members that your add-in must have to be of any significant value at all. If your add-in uses an API for some features but has other useful features that don't require the API, you should design the add-in so that it's installable on platform and Office version combinations that don't support the API but provides a diminished experience on those combinations. For more information, see Design for alternate experiences.

Requirement sets

To simplify the process of specifying the APIs that your add-in needs, Office groups most APIs together in requirement sets. The APIs in the Common API Object Model are grouped by the development feature that they support. For example, all the APIs connected to table bindings are in the requirement set called "TableBindings 1.1". The APIs in the Application specific object models are grouped by when they were released for use in production add-ins.

Requirement sets are versioned. For example, the APIs that support Dialog Boxes are in the requirement set DialogApi 1.1. When additional APIs that enable messaging from a task pane to a dialog were released, they were grouped into DialogApi 1.2, along with

all the APIs in DialogApi 1.1. *Each version of a requirement set is a superset of all earlier versions.*

Requirement set support varies by Office application, the version of the Office application, and the platform on which it is running. For example, DialogApi 1.2 isn't supported on volume-licensed perpetual versions of Office before Office 2021, but DialogApi 1.1 is supported on all perpetual versions back to Office 2016. You want your add-in to be installable on every combination of platform and Office version that supports the APIs that it uses, so you should always specify in the manifest the *minimum* version of each requirement set that your add-in requires. Details about how to do this are later in this article.

> **Tip**
>
> For more information about requirement set versioning, see **Office requirement sets availability**, and for the complete lists of requirement sets and information about the APIs in each, start with **Office Add-in requirement sets**. The reference topics for most Office.js APIs also specify the requirement set they belong to (if any).

> **Note**
>
> Some requirement sets also have manifest elements associated with them. See **Specifying requirements in a VersionOverrides element** for information about when this fact is relevant to your add-in design.

Requirements element

Use the Requirements element and its child element Sets to specify the minimum requirement sets that must be supported by the Office application to install your add-in.

All APIs in the application specific models are in requirement sets, but some of those in the Common API model are not. Use the Methods to specify the setless API members that your add-in requires. You can't use the **<Methods>** element with Outlook add-ins.

If the Office application or platform doesn't support the requirement sets or API members specified in the **<Requirements>** element, the add-in won't run in that application or platform, and won't display in **My Add-ins**.

> **Note**

The <**Requirements**> element is optional for all add-ins, except for Outlook add-ins. When the `xsi:type` attribute of the root `OfficeApp` element is `MailApp`, there must be a <**Requirements**> element that specifies the minimum version of the Mailbox requirement set that the add-in requires. For more information, see Outlook JavaScript API requirement sets.

The following code example shows how to configure an add-in that is installable in all Office 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.

XML

```
<OfficeApp ... >
  ...
  <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>
    ...
</OfficeApp>
```

Note the following about this example.

- 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.
- A Set element specifies a requirement set that the Office application must support to make the add-in installable. 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 the `DefaultMinVersion` attribute in the parent <**Sets**>.
- 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 the Office application must support to make the add-in installable. The `Name` attribute is required and

specifies the name of the method qualified with its parent object.

Design for alternate experiences

The extensibility features that the Office Add-in platform provides can be usefully divided into three kinds:

- Extensibility features that are available immediately after the add-in is installed. You can make use of this kind of feature by configuring a VersionOverrides element in the manifest. An example of this kind of feature is Add-in Commands, which are custom ribbon buttons and menus.
- Extensibility features that are available only when the add-in is running and that are implemented with Office.js JavaScript APIs; for example, Dialog Boxes.
- Extensibility features that are available only at runtime but are implemented with a combination of Office.js JavaScript and configuration in a **<VersionOverrides>** element. Examples of these are Excel custom functions, single sign-on, and custom contextual tabs.

If your add-in uses a specific extensibility feature for some of its functionality but has other useful functionality that doesn't require the extensibility feature, you should design the add-in so that it's installable on platform and Office version combinations that don't support the extensibility feature. It can provide a valuable, albeit diminished, experience on those combinations.

You implement this design differently depending on how the extensibility feature is implemented:

- For features implemented entirely with JavaScript, see Check for API availability at runtime.
- For features that require you to configure a **<VersionOverrides>** element, see Specifying requirements in a VersionOverrides element.

Specify requirements in a VersionOverrides element

The VersionOverrides element was added to the manifest schema primarily, but not exclusively, to support features that must be available immediately after an add-in is installed, such as add-in commands (custom ribbon buttons and menus). Office must know about these features when it parses the add-in manifest.

Suppose your add-in uses one of these features, but the add-in is valuable, and should be installable, even on Office versions that don't support the feature. In this scenario, identify the feature using a Requirements element (and its child Sets and Methods

elements) that you include as a child of the <**VersionOverrides**> element itself instead of as a child of the base `OfficeApp` element. The effect of doing this is that Office will allow the add-in to be installed, but Office will ignore certain of the child elements of the <**VersionOverrides**> element on Office versions where the feature isn't supported.

Specifically, the child elements of the <**VersionOverrides**> that override elements in the base manifest, such as a <**Hosts**> element, are ignored and the corresponding elements of the base manifest are used instead. However, there can be child elements in a <**VersionOverrides**> that actually implement additional features rather than override settings in the base manifest. Two examples are the `WebApplicationInfo` and `EquivalentAddins`. These parts of the <**VersionOverrides**> will *not* be ignored, assuming the platform and version of Office support the corresponding feature.

For information about the descendent elements of the <**Requirements**> element, see Requirements element earlier in this article.

The following is an example.

XML

```
<VersionOverrides ... >
   ...
   <Requirements>
      <Sets DefaultMinVersion="1.1">
         <Set Name="WordApi" MinVersion="1.2"/>
      </Sets>
   </Requirements>
   <Hosts>

      <!-- ALL MARKUP INSIDE THE HOSTS ELEMENT IS IGNORED WHEREVER WordApi
1.2 IS NOT SUPPORTED -->

      <Host xsi:type="Workbook">
         <!-- markup for custom add-in commands -->
      </Host>
   </Hosts>
</VersionOverrides>
```

> **⚠ Warning**
>
> If your add-in includes add-in commands, use great care before including a <**Requirements**> element in a <**VersionOverrides**>. On platform and version combinations that don't support the requirement, *none* of the add-in commands will be installed, *even those that invoke functionality that doesn't need the requirement*. Consider, for example, an add-in that has two custom ribbon buttons. One of them calls Office JavaScript APIs that are available in requirement set

ExcelApi 1.4 (and later). The other calls APIs that are only available in **ExcelApi 1.9** (and later). If you put a requirement for **ExcelApi 1.9** in the <**VersionOverrides**>, then when 1.9 isn't supported, *neither* button will appear on the ribbon. A better strategy in this scenario would be to use the technique described in **Check for API availability at runtime**. The code invoked by the second button first uses `isSetSupported` to check for support of **ExcelApi 1.9**. If it isn't supported, the code gives the user a message saying that this feature of the add-in isn't available on their version of Office.

Tip

There's no point to repeating a <**Requirement**> element in a <**VersionOverrides**> that already appears in the base manifest. If the requirement is specified in the base manifest, then the add-in can't install where the requirement isn't supported so Office doesn't even parse the <**VersionOverrides**> element.

See also

- Office Add-ins manifest
- Office Add-in requirement sets
- Word-Add-in-Get-Set-EditOpen-XML

Understanding platform-specific requirement sets

Article • 11/19/2024

The Office Add-ins platform allows you to build solutions that extend Office applications and interact with content in Office documents. Your solution can run in Office across several platforms, including Windows, Mac, iPad, and in a browser. We've provided requirement sets that help you declare which APIs and platforms your add-in supports. Requirement sets are named groups of API members which are usually supported on all available platforms. However, with platform-specific requirement sets, APIs are implemented and made available first in the target platforms.

Each application that supports Office Add-ins has its usual set of available platforms. For a comprehensive listing, see Office client application and platform availability for Office Add-ins. For the purpose of this discussion, we'll focus on Excel, Outlook, PowerPoint, and Word.

Cross-platform requirement sets are available on Windows, Mac, and in a browser. Depending on the features being made available, a requirement set may also be supported on iPad or mobile platforms.

However, platform-specific requirement sets provide support for a subset of the usual platforms. For example, online-only requirement sets provide APIs that are only available when the add-in runs in a web browser. Similarly, desktop-only requirement sets provide APIs that may only be available when the add-in runs in Windows and Mac. See the specific requirement set page for actual platform support.

Current platform-specific requirement sets

At present, platform-specific requirement sets are available in Excel and Word. Excel includes an online-only requirement set. Word includes online-only and desktop-only requirement sets. For the full list, see Special requirement sets.

Note that in Outlook, platform-specific behavior may be found in extension points. For example, MobileOnlineMeetingCommandSurface and MobileLogEventAppointmentAttendee are only available to add-ins running in Outlook on Android and on iOS.

Why platform-specific requirement sets?

We're providing platform-specific requirement sets for a few reasons.

1. **Feature availability.** Some features aren't implemented in the Office applications UI on a particular platform. As such, the API can only be made available on supported platforms. Having these types of APIs in a platform-specific requirement set means that developers can use those APIs in their add-ins. This is especially useful for cases where the feature may never be implemented in other platforms.
2. **Platform-specific add-ins.** Developers who have add-ins focused on a particular platform don't need to wait for those APIs to be implemented in other platforms. These developers are able to incorporate those APIs into their solutions and ship to their customers much sooner.
3. **Tailored experiences.** Customers can use an Office application differently depending on the platform for several reasons, like feature availability or comfort level, for example. Let's say that on the Windows version, a customer completes one set of tasks but on an iPad, they complete a different set of tasks. You can have your add-in provide a tailored experience based on your users' usual scenarios per platform.

To help you decide if platform-specific requirement sets can work for you, consider the following.

API promotion to cross-platform requirement set

When APIs in a platform-specific requirement set are supported cross-platform, they're added to the next requirement set targeted for release. Even after the new requirement is made generally available, those APIs *still remain* in the platform-specific requirement set.

How to use a platform-specific requirement set

The following sections describe where you can specify your minimum requirement set. For more information about these options, see Specify which Office versions and platforms can host your add-in.

Manifest

When you note a requirement set in the Set element of your add-in manifest, you're indicating the minimum set of APIs that your add-in needs. Combined with supported

Office host applications and other information, this determines whether or not your add-in activates in an Office client.

When you declare a platform-specific requirement set, your add-in activates only when it's run in Office on that platform. For example, if you have the WordApiDesktop 1.1 requirement set in your manifest, your add-in will only activate in Word on Windows and on Mac.

Keep in mind that in the case where the APIs become supported cross-platform, you'll need to update your add-in manifest to add a cross-platform requirement set and remove the platform-specific requirement set. If your add-in is available in AppSource or the Office store, you'll need to resubmit it for validation.

Code

Another option is to implement a runtime check in your code. This way, you can make new features available to your customers on those platforms. The runtime check also ensures that the platform-specific code doesn't run on unsupported platforms and cross-platform features continue to work for your customers. The following code is an example that checks for a specific requirement set.

JavaScript

```
if (Office.context.requirements.isSetSupported("WordApiDesktop", "1.1")) {
   // Any API exclusive to this WordApiDesktop requirement set.
}
```

Whenever platform-specific APIs become available cross-platform, enable your customers on all supported platforms to use those features by implementing one of the following options.

- Remove the runtime check. But note that customers on older Office clients, especially on Windows, may hit errors if their client doesn't support the new APIs yet.
- Update the runtime code to check for the cross-platform requirement set.

A variation is to do a runtime check for a particular API. This means that the encapsulated code should run on any platforms that support that API. If the API was first released in a platform-specific requirement set then promoted to a cross-platform one, you shouldn't need to update your code unless you made assumptions about the supported platforms. The following code is an example.

JavaScript

```
if (Office.context.document.setSelectedDataAsync)
{
    // Run code that uses document.setSelectedDataAsync.
}
```

Notify customers on AppSource

If your add-in is in AppSource or the Office store, be sure to notify customers about any platform-specific behavior.

Details + support > **Products supported** on your add-in's AppSource page should automatically show the appropriate supported platforms based on the requirements you declared in the manifest.

However, if your add-in is supported cross-platform but you also implemented platform-specific behaviors, you should point out those feature differences in the **Overview** section on your add-in's AppSource page.

Exceptions

The following are exceptions to the approach described.

Online-only requirement sets

An online-only requirement set is a superset of the latest numbered requirement set. For each Office application with an online-only requirement set, `1.1` is the only version. It's invalid to specify an online-only requirement set in the Set element of your add-in manifest.

To check for APIs that are only supported in these requirement sets and to prevent your add-in from trying to run the code on unsupported platforms, add code similar to the following:

JavaScript

```
if (Office.context.requirements.isSetSupported("ExcelApiOnline", "1.1")) {
   // Any API exclusive to the ExcelApiOnline requirement set.
}
```

JavaScript

```
if (Office.context.requirements.isSetSupported("WordApiOnline", "1.1")) {
   // Any API exclusive to the WordApiOnline requirement set.
}
```

When APIs in an online-only requirement set are supported cross-platform, they're added to the next released requirement set. After the new requirement set is made generally available, those APIs are *removed* from the online-only requirement set.

Follow the guidance in the earlier Code section to adjust your add-in implementation accordingly.

Desktop-only HiddenDocument requirement sets in Word

It's important to note that while the HiddenDocument requirement sets in Word are desktop-only, it's invalid to specify a HiddenDocument requirement set in the Set element of your add-in manifest.

To check for APIs that are only supported in these requirement sets and to prevent your add-in from trying to run the code on unsupported platforms, add code similar to the following:

JavaScript

```
if (Office.context.requirements.isSetSupported("WordApiHiddenDocument",
"1.5")) {
   // Any API exclusive to this WordApiHiddenDocument requirement set.
}
```

See also

- Understanding the Office JavaScript API
- Office versions and requirement sets
- Specify Office applications and API requirements
- Office client application and platform availability for Office Add-ins

Check for API availability at runtime

Article • 02/12/2025

If your add-in uses a specific extensibility feature for some of its functionality, but has other useful functionality that doesn't require the extensibility feature, you should design the add-in so that it's installable on platform and Office version combinations that don't support the extensibility feature. It can provide a valuable, albeit diminished, experience on those combinations.

When the difference in the two experiences consists entirely of differences in the Office JavaScript Library APIs that are called, and not in any features that are configured in the manifest, then you test at runtime to discover whether the user's Office client supports an API requirement set. You can also test at runtime whether APIs that aren't in a set are supported.

> **Note**
>
> To provide alternate experiences with features that require manifest configuration, follow the guidance in Specify Office hosts and API requirements with the unified manifest or Specify Office applications and API requirements with the add-in only manifest.

Check for requirement set support

The isSetSupported method is used to check for requirement set support. Pass the requirement set's name and the minimum version as parameters. If the requirement set is supported, `isSetSupported` returns `true`. The following code shows an example.

JavaScript

```
if (Office.context.requirements.isSetSupported("WordApi", "1.2")) {
   // Code that uses API members from the WordApi 1.2 requirement set.
} else {
   // Provide diminished experience here.
   // For example, run alternate code when the user's Word is
   // volume-licensed perpetual Word 2016 (which doesn't support WordApi
1.2).
}
```

About this code, note:

- The first parameter is required. It's a string that represents the name of the requirement set. For more information about available requirement sets, see Office Add-in requirement sets.
- The second parameter is optional. It's a string that specifies the minimum requirement set version that the Office application must support in order for the code within the `if` statement to run (for example, "1.9"). If not used, version "1.1" is assumed.

> **⚠ Warning**
>
> When calling the `isSetSupported` method, the value of the second parameter (if specified) should be a string, not a number. The JavaScript parser can't differentiate between numeric values such as 1.1 and 1.10, whereas it can for string values such as "1.1" and "1.10".

The following table shows the requirement set names for the application-specific API models.

Expand table

Office application	RequirementSetName
Excel	ExcelApi
OneNote	OneNoteApi
Outlook	Mailbox
PowerPoint	PowerPointApi
Word	WordApi

The following is an example of using the method with one of the Common API model requirement sets.

JavaScript

```
if (Office.context.requirements.isSetSupported('CustomXmlParts')) {
    // Run code that uses API members from the CustomXmlParts requirement
set.
} else {
    // Run alternate code when the user's Office application doesn't support
the CustomXmlParts requirement set.
}
```

> **Note**
>
> The `isSetSupported` method and the requirement sets for these applications are available in the latest Office.js file on the CDN. If you don't use Office.js from the CDN, your add-in might generate exceptions if you are using an old version of the library in which `isSetSupported` is undefined. For more information, see Use the latest Office JavaScript API library.

Check for setless API support

When your add-in depends on a method that isn't part of a requirement set, called a setless API, use a runtime check to determine whether the method is supported by the Office application, 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.

> **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 Office application supports `document.setSelectedDataAsync`.

JavaScript

```
if (Office.context.document.setSelectedDataAsync) {
    // Run code that uses `document.setSelectedDataAsync`.
}
```

See also

- Office requirement sets availability
- Specify Office hosts and API requirements with the unified manifest
- Specify Office hosts and API requirements with the add-in only manifest

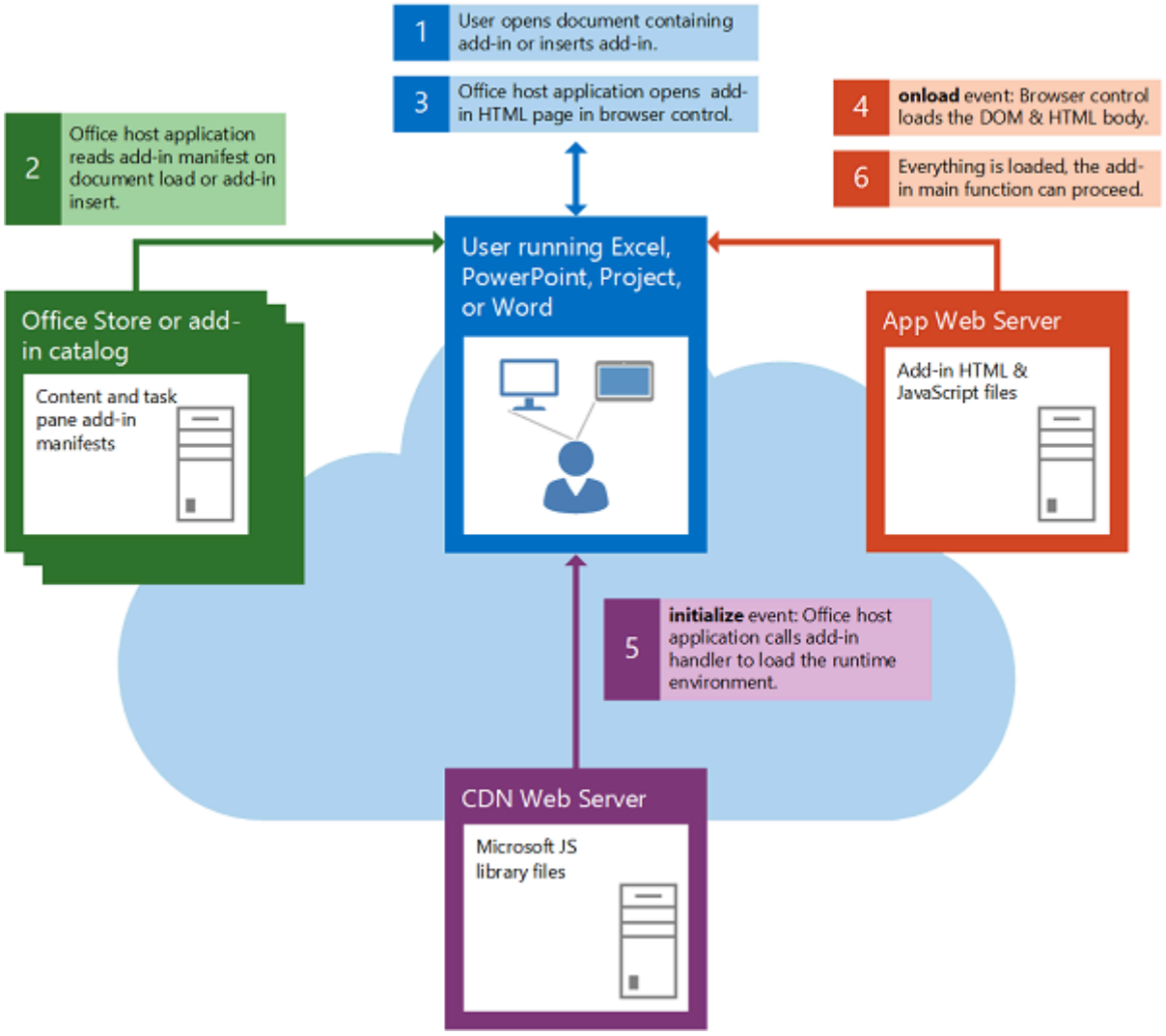
Load the DOM and runtime environment

Article • 05/20/2023

Before running its own custom logic, an add-in must ensure that both the DOM and the Office Add-ins runtime environment are loaded.

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, or Word.



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 client application reads the add-in's manifest from AppSource, an app catalog on SharePoint, or the shared folder catalog it originates from.

3. The Office client application opens the add-in's HTML page in a webview 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.

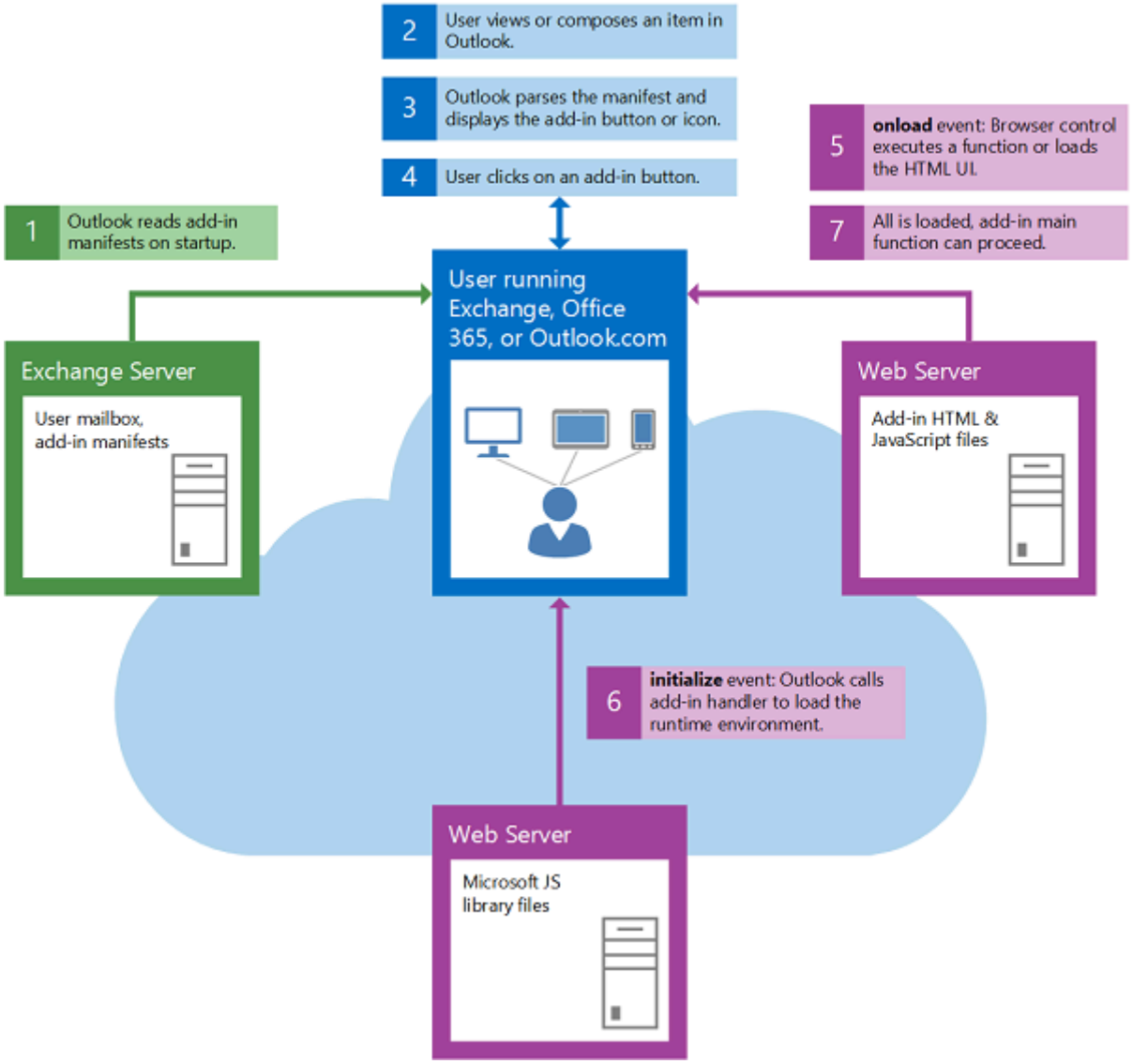
4. The webview control loads the DOM and HTML body, and calls the event handler for the `window.onload` event.

5. The Office client application loads the runtime environment, which downloads and caches the Office JavaScript API library files from the content distribution network (CDN) server, and then calls the add-in's event handler for the initialize event of the Office object, if a handler has been assigned to it. At this time it also checks to see if any callbacks (or chained `then()` method) have been passed (or chained) to the `Office.onReady` handler. For more information about the distinction between `Office.initialize` and `Office.onReady`, see Initialize your add-in.

6. 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.



The following events occur when an Outlook add-in starts.

1. When Outlook starts, Outlook reads the 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 webview control. The next two steps, steps 5 and 6, occur in parallel.
5. The webview control loads the DOM and HTML body, and calls the event handler for the `onload` event.
6. Outlook 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 event handler for the initialize event of the Office object of the add-in, if a handler has been assigned to it. At this time it also checks

to see if any callbacks (or chained `then()` methods) have been passed (or chained) to the `Office.onReady` handler. For more information about the distinction between `Office.initialize` and `Office.onReady`, see Initialize your 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.

See also

- Understanding the Office JavaScript API
- Initialize your Office Add-in
- Runtimes in Office Add-ins

Initialize your Office Add-in

Article • 08/18/2023

Office Add-ins often have start-up logic to do things such as:

- Check that the user's version of Office supports all the Office APIs that your code calls.
- Ensure the existence of certain artifacts, such as a worksheet with a specific name.
- Prompt the user to select some cells in Excel, and then insert a chart initialized with those selected values.
- Establish bindings.
- Use the Office Dialog API to prompt the user for default add-in settings values.

However, an Office Add-in can't successfully call any Office JavaScript APIs until the library has been loaded. This article describes the two ways your code can ensure that the library has been loaded.

- Initialize with `Office.onReady()`.
- Initialize with `Office.initialize`.

> **Tip**
>
> We recommend that you use `Office.onReady()` instead of `Office.initialize`. Although `Office.initialize` is still supported, `Office.onReady()` provides more flexibility. You can assign only one handler to `Office.initialize` and it's called only once by the Office infrastructure. You can call `Office.onReady()` in different places in your code and use different callbacks.
>
> For information about the differences in these techniques, see **Major differences between Office.initialize and Office.onReady()**.

For more details about the sequence of events when an add-in is initialized, see Loading the DOM and runtime environment.

Initialize with Office.onReady()

`Office.onReady()` is an asynchronous function that returns a Promise object while it checks to see if the Office.js library is loaded. When the library is loaded, it resolves the Promise as an object that specifies the Office client application with an `Office.HostType` enum value (`Excel`, `Word`, etc.) and the platform with an `Office.PlatformType` enum value (`PC`, `Mac`, `OfficeOnline`, etc.). The Promise resolves immediately if the library is already loaded when `Office.onReady()` is called.

One way to call `Office.onReady()` is to pass it a callback function. Here's an example.

JavaScript

```
Office.onReady(function(info) {
    if (info.host === Office.HostType.Excel) {
        // Do Excel-specific initialization (for example, make add-in task
pane's
        // appearance compatible with Excel "green").
    }
    if (info.platform === Office.PlatformType.PC) {
        // Make minor layout changes in the task pane.
    }
    console.log(`Office.js is now ready in ${info.host} on
${info.platform}`);
});
```

Alternatively, you can chain a `then()` method to the call of `Office.onReady()`, instead of passing a callback. For example, the following code checks to see that the user's version of Excel supports all the APIs that the add-in might call.

JavaScript

```
Office.onReady()
    .then(function() {
        if (!Office.context.requirements.isSetSupported('ExcelApi', '1.7'))
{
            console.log("Sorry, this add-in only works with newer versions
of Excel.");
        }
    });
```

Here's the same example using the `async` and `await` keywords in TypeScript.

TypeScript

```
(async () => {
    await Office.onReady();
    if (!Office.context.requirements.isSetSupported('ExcelApi', '1.7')) {
        console.log("Sorry, this add-in only works with newer versions of
```

```
Excel.");
    }
})();
```

If you're using additional JavaScript frameworks that include their own initialization handler or tests, these should *usually* be placed within the response to `Office.onReady()`. For example, JQuery's `$(document).ready()` method would be referenced as follows:

JavaScript

```
Office.onReady(function() {
    // Office is ready.
    $(document).ready(function () {
        // The document is ready.
    });
});
```

However, there are exceptions to this practice. For example, suppose you want to open your add-in in a browser (instead of sideload it in an Office application) in order to debug your UI with browser tools. In this scenario, once Office.js determines that it is running outside of an Office host application, it will call the callback and resolve the promise with `null` for both the host and platform.

Another exception would be if you want a progress indicator to appear in the task pane while the add-in is loading. In this scenario, your code should call the jQuery `ready` and use its callback to render the progress indicator. Then the `Office.onReady` callback can replace the progress indicator with the final UI.

Initialize with Office.initialize

An initialize event fires when the Office.js library is loaded and ready for user interaction. You can assign a handler to `Office.initialize` that implements your initialization logic. The following is an example that checks to see that the user's version of Excel supports all the APIs that the add-in might call.

JavaScript

```
Office.initialize = function () {
    if (!Office.context.requirements.isSetSupported('ExcelApi', '1.7')) {
        console.log("Sorry, this add-in only works with newer versions of
Excel.");
    }
};
```

If you're using additional JavaScript frameworks that include their own initialization handler or tests, these should *usually* be placed within the `Office.initialize` event (the exceptions described in the **Initialize with Office.onReady()** section earlier apply in this case also). For example, JQuery's `$(document).ready()` method would be referenced as follows:

JavaScript

```
Office.initialize = function () {
    // Office is ready.
    $(document).ready(function () {
        // The document is ready.
    });
  };
```

For task pane and content add-ins, `Office.initialize` provides an additional *reason* parameter. This parameter specifies 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.

JavaScript

```
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 and InitializationReason Enumeration.

Major differences between Office.initialize and Office.onReady

- You can assign only one handler to `Office.initialize` and it's called only once by the Office infrastructure; but you can call `Office.onReady()` in different places in your code and use different callbacks. For example, your code could call `Office.onReady()` as soon as your custom script loads with a callback that runs initialization logic; and your code could also have a button in the task pane, whose

script calls `Office.onReady()` with a different callback. If so, the second callback runs when the button is clicked.

- The `Office.initialize` event fires at the end of the internal process in which Office.js initializes itself. And it fires *immediately* after the internal process ends. If the code in which you assign a handler to the event executes too long after the event fires, then your handler doesn't run. For example, if you are using the WebPack task manager, it might configure the add-in's home page to load polyfill files after it loads Office.js but before it loads your custom JavaScript. By the time your script loads and assigns the handler, the initialize event has already happened. But it's never "too late" to call `Office.onReady()`. If the initialize event has already happened, the callback runs immediately.

> **Note**
>
> Even if you have no start-up logic, you should either call `Office.onReady()` or assign an empty function to `Office.initialize` when your add-in JavaScript loads. Some Office application and platform combinations won't load the task pane until one of these happens. The following examples show these two approaches.

JavaScript

```
Office.onReady();
```

JavaScript

```
Office.initialize = function () {};
```

Debug initialization

For information about debugging the `Office.initialize` and `Office.onReady()` functions, see Debug the initialize and onReady functions.

See also

- Understanding the Office JavaScript API
- Loading the DOM and runtime environment

Automatically open a task pane with a document

Article • 02/12/2025

You can use add-in commands in your Office Add-in to extend the Office UI by adding buttons to the Office app 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 task pane feature, introduced in the AddInCommands 1.1 requirement set, to automatically open a task pane when your scenario requires it.

> **Note**
>
> To configure a task pane to open immediately when the add-in is installed, but not necessarily whenever the document is opened later, see Automatically open a task pane when an add-in is installed.

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

When a user launches add-ins that don't use add-in commands, the add-ins are inserted into the document, and persist in that document. As a result, when other users open the document, they're 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.

Expand table

Products	Platforms
• Word • Excel • PowerPoint	Supported platforms for all supported products: • Office on the web • Office on Windows (Version 1705 (Build 8121.1000) or later) • Office on Mac (Version 15.34 (17051500) or later)

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 AppSource.

- 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.

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.

Step 1: Specify the task pane to open

Configure the manifest to specify the task pane page that should open automatically. The process depends on what type of manifest the add-in uses.

Unified manifest for Microsoft 365

> **Note**
>
> The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

To specify the task pane to open automatically, find the runtime object in the "runtimes" array whose "code.page" property is set to the URL of the page that you want to open automatically. Ensure that the "actions" array in this same runtime object has at least one action whose "type" value is "openPage". Add a "view" property to this action object and set it to "Office.AutoShowTaskpaneWithDocument". You can only set this value on one action object and it must be an action of type "openPage". If you set this value on multiple actions, the first occurrence of the value will be recognized and the others will be ignored.

The following example shows a "view" value set to "Office.AutoShowTaskpaneWithDocument".

JSON

```
"runtimes": [
    {
        ...
        "code": {
            "page": "https://contoso.com/taskpane.html"
        },
        "actions": [
            {
                "id": "ShowTaskPane",
                "type": "openPage",
                "view": "Office.AutoShowTaskpaneWithDocument"
            }
        ]
    }
]
```

Step 2: Tag the document to automatically open the task pane

You can tag the document to trigger the autoopen feature in one of two ways. Pick the alternative that works best for your scenario.

Tag the document on the client side

Use the Office.js settings.set method to set **Office.AutoShowTaskpaneWithDocument** to `true`, as shown in the following example.

JavaScript

```
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.

Add two Open XML parts to the document.

- A `webextension` part
- A `taskpane` part

The following example shows how to add the `webextension` part.

XML

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

Expand table

`storeType` value	`id` value	`store` value	`version` value
OMEX (AppSource)	The AppSource asset ID of the add-in (see Note).	The locale of AppSource; for example, "en-us".	The version in the AppSource catalog (see Note).

`storeType` value	`id` value	`store` value	`version` value
WOPICatalog (partner WOPI hosts)	The AppSource asset ID of the add-in (see Note).	"wopicatalog". Use this value for add-ins that are published in App Source and are installed in WOPI hosts. For more information, see Integrating with Office Online.	The version in the add-in manifest.
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". EXCatalog row is the row to use with add-ins that use Centralized Deployment in the Microsoft 365 admin center.	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.

> **Note**
>
> To find the asset ID and version of an add-in in AppSource, go to the AppSource 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.

The following example shows how to add the `taskpane` part.

XML

```
<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.)

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 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 Productivity Tool to generate C# code to programmatically add the markup based on the XML you generate.

Test and verify opening task panes

You can deploy a test version of your add-in that will automatically open a task pane using Centralized Deployment via the Microsoft 365 admin center. The following example shows how add-ins are inserted from the Centralized Deployment catalog using the EXCatalog store version.

```
XML
```

```
<we:webextension
xmlns:we="http://schemas.microsoft.com/office/webextensions/webextension/2010/11" id="{52811C31-4593-43B8-A697-EB873422D156}">
    <we:reference id="af8fa5ba-4010-4bcc-9e03-a91ddadf6dd3"
version="1.0.0.0" store="EXCatalog" storeType="EXCatalog"/>
    <we:alternateReferences/>
    <we:properties/>
    <we:bindings/>
    <we:snapshot
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships
"/>
</we:webextension>
```

You can test the previous example by using your Microsoft 365 subscription to try out Centralized Deployment and verify that your add-in works as expected. If you don't already have a Microsoft 365 subscription, you might qualify for a Microsoft 365 E5 developer subscription through the Microsoft 365 Developer Program; for details, see the FAQ. Alternatively, you can sign up for a 1-month free trial or purchase a Microsoft 365 plan.

See also

- For a sample that shows you how to use the autoopen feature, see Auto-open a task pane with a document.
- Automatically open a task pane when an add-in is installed
- Learn about the Microsoft 365 Developer Program

Automatically open a task pane when an add-in is installed

Article • 02/12/2025

You can configure your add-in's task pane to launch immediately after it's installed. This feature increases usage.

By default, task pane add-ins that do *not* include any add-in commands open the task pane immediately upon installation. However, when an add-in has one or more add-in commands, then the user is notified of new add-in, but the add-in doesn't launch automatically. This historic default behavior is changing so add-ins that do have add-in commands will launch automatically in some situations. In addition, if the add-in has more than one task pane page, it's possible for you to control whether the add-in launches upon installation and, if so, which page opens in the task pane.

> **Note**
>
> - This feature applies only to add-ins installed by an end-user, not to centrally deployed add-ins.
> - This feature doesn't apply to Content add-ins or Mail (Outlook) add-ins.
> - This feature applies only to add-ins that have at least one add-in command of the type "task pane command".

New behavior

The new behavior is as follows:

- If the add-in has just one task pane command, then the add-in's ribbon tab is selected and the task pane opens automatically upon installation. You don't need to configure anything.
- If the add-in has multiple task pane commands, and one is configured to be the default (see Configure default task pane), then the add-in's ribbon tab is selected and the default task pane opens automatically upon installation.
- If the add-in has multiple task pane commands, but none is configured to be the default, then the add-in's ribbon tab is selected automatically upon installation and a callout appears near it notifying the user of the new add-in, but no task pane is opened. This is the same as the historic default behavior.

> **Note**
>
> If for any reason, the add-in command that launches the task pane cannot be manually selected by a user at start up, such as when it's configured to be disabled at start up, then it won't be automatically opened regardless of configuration.

Configure default task pane in the manifest

The process for specifying the default task pane depends on the type of manifest the add-in uses.

Unified manifest for Microsoft 365

> **Note**
>
> The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

To specify the default task pane, find the runtime object in the "runtimes" array whose "code.page" property is set to the URL of the page that you want to be the default. Ensure that the "actions" array in this same runtime object has at least one action whose "type" value is "openPage". Add a "view" property to this action object and set it to "Office.AutoShowTaskpaneWithDocument". You can only set this value on one action object and it must be an action of type "openPage". If you set this value on multiple actions, the first occurrence of the value will be recognized as the default and the others will be ignored.

The following example shows a "view" value set to "Office.AutoShowTaskpaneWithDocument".

JSON

```
"runtimes": [
    {
        ...
        "code": {
            "page": "https://contoso.com/taskpane.html"
        },
        "actions": [
            {
                "id": "ShowTaskPane",
```

```
                "type": "openPage",
                "view": "Office.AutoShowTaskpaneWithDocument"
            }
        ]
    }
]
```

> **Tip**
>
> If you want your add-in to automatically launch whenever the user reopens the document, you need to take further configuration steps. For details and advice about when to use this feature, see **Automatically open a task pane with a document**.

See also

- Automatically open a task pane with a document

Configure your Office Add-in to use a shared runtime

Article • 03/12/2025

> **Important**
>
> The shared runtime is only supported in some Office applications. For more information, see Shared runtime requirement sets.

You can configure your Office Add-in to run all of its code in a single shared runtime. With a shared runtime, you'll have better coordination across your add-in and access to the DOM and CORS from all parts of your add-in. You'll also have access to additional features, such as running code when the document opens or activating ribbon buttons in certain contexts. To configure your add-in to use a shared runtime, follow the instructions in this article.

Create the add-in project

If you're starting a new project, use the Yeoman generator for Office Add-ins to create an Excel, PowerPoint, or Word add-in project.

> **Tip**
>
> If you're using the Yeoman generator to create custom functions in Excel, select the following options:
>
> - **Project type**: `Excel Custom Functions using a Shared Runtime`
> - **Script type**: `JavaScript`

If your add-in uses an add-in only manifest, you can also use the steps in this article to update a Visual Studio project to use the shared runtime. However, you may need to update the XML schemas for the manifest. For more information, see Troubleshoot development errors with Office Add-ins.

Configure the manifest

Follow these steps to configure a new or existing project to use a shared runtime. These steps assume you have generated your project using the Yeoman generator for Office Add-ins. Select the tab for the type of manifest your add-in is using.

Unified manifest for Microsoft 365

> **Note**
>
> Implementing a shared runtime with the unified manifest for Microsoft 365 is in public developer preview. This shouldn't be used in production add-ins. We invite you to try it out in test or development environments. For more information, see the Public developer preview app manifest schema.

1. Open your add-in project in Visual Studio Code.
2. Open the **manifest.json** file.
3. Add the following object to the "extensions.runtimes" array. Note the following about this markup.
    - The SharedRuntime 1.1 requirement set is specified in the "requirements.capabilities" object. This configures your add-in to run in a shared runtime on supported clients. For a list of clients that support the SharedRuntime 1.1 requirement set, see Shared runtime requirement sets.
    - The "id" of the runtime is set to the descriptive name "SharedRuntime".
    - The "lifetime" property is set to "long", so that your add-in can take advantage of features, such as starting your add-in when the document opens, continuing to run code after the task pane is closed, or using CORS and DOM from custom functions. If you set the property to "short" in this example, your add-in will start when one of your ribbon buttons is pressed, but it may shut down after your ribbon handler is done running. Similarly, your add-in will start when the task pane is opened, but it may shut down when the task pane is closed.

JSON

```
"runtimes": [
    "requirements": {
        "capabilities": [
            {
                "name": "SharedRuntime",
                "minVersion": "1.1"
```

```
            }
        ]
    },
    "id": "SharedRuntime",
    "type": "general",
    "code": {
        "page": "https://localhost:3000/taskpane.html"
    },
    "lifetime": "long",
    "actions": [
        ...
    ]
]
```

4. Save your changes.

Configure the webpack.config.js file

The **webpack.config.js** will build multiple runtime loaders. You need to modify it to load only the shared runtime via the **taskpane.html** file.

1. Start Visual Studio Code and open the add-in project you generated.
2. Open the **webpack.config.js** file.
3. If your **webpack.config.js** file has the following **functions.html** plugin code, remove it.

```
JavaScript

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

4. If your **webpack.config.js** file has the following **commands.html** plugin code, remove it.

```
JavaScript

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

5. If your project used either the **functions** or **commands** chunks, add them to the chunks list as shown next (the following code is for if your project used both chunks).

```
JavaScript

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

6. Save your changes and rebuild the project.

```
command line

npm run build
```

> **Note**
>
> If your project has a **functions.html** file or **commands.html** file, they can be removed. The **taskpane.html** will load the **functions.js** and **commands.js** code into the shared runtime via the webpack updates you just made.

Test your Office Add-in changes

Confirm that you're using the shared runtime correctly by using the following instructions.

1. Open the **taskpane.js** file.

2. Replace the entire contents of the file with the following code. This will display a count of how many times the task pane has been opened. Adding the `onVisibilityModeChanged` event is only supported in a shared runtime.

```
JavaScript

/*global document, Office*/

let _count = 0;

Office.onReady(() => {
  document.getElementById("sideload-msg").style.display = "none";
  document.getElementById("app-body").style.display = "flex";
```

```
  updateCount(); // Update count on first open.
  Office.addin.onVisibilityModeChanged((args) => {
    if (args.visibilityMode === Office.VisibilityMode.taskpane) {
      updateCount(); // Update count on subsequent opens.
    }
  });
});

function updateCount() {
  _count++;
  document.getElementById("run").textContent = "Task pane opened " +
_count + " times.";
}
```

3. Save your changes and run the project.

command line

```
npm start
```

Each time you open the task pane, the count of how many times it has been opened will be incremented. The value of **_count** won't be lost because the shared runtime keeps your code running even when the task pane is closed.

When you're ready to stop the dev server and uninstall the add-in, run the following command.

command line

```
npm stop
```

About the shared runtime

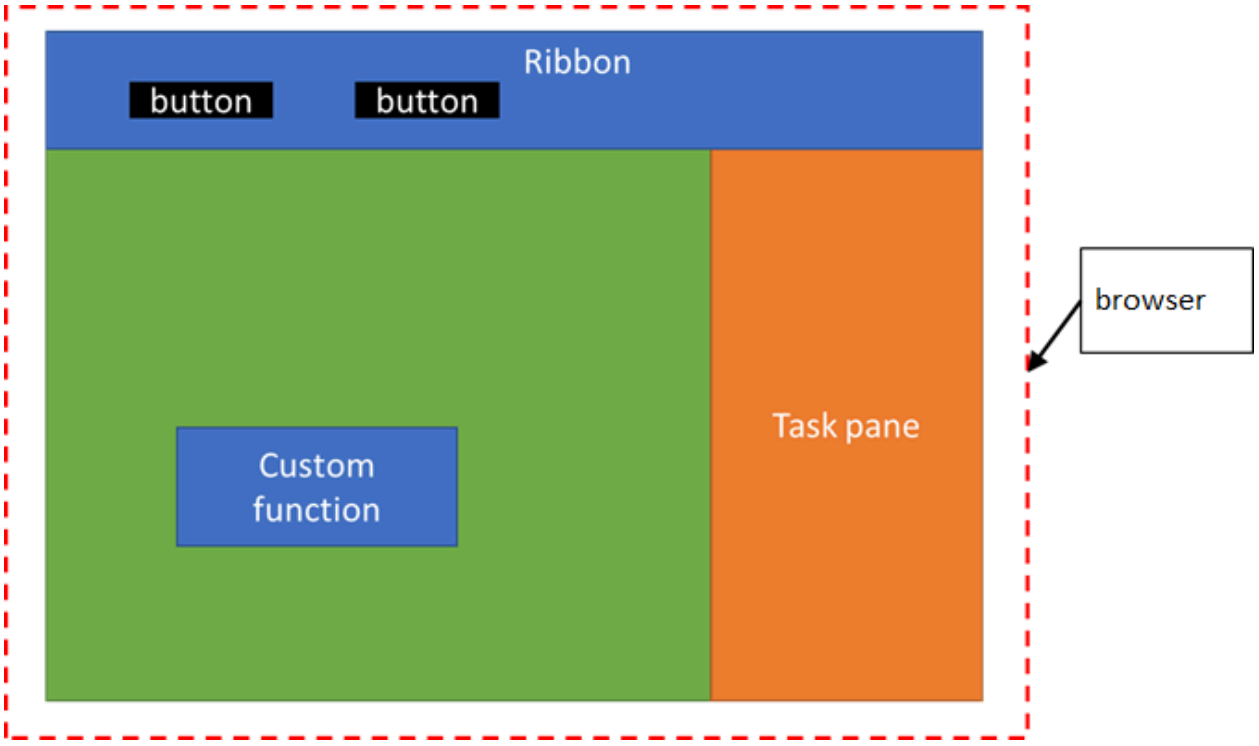
On Windows or on Mac, your add-in will run code for ribbon buttons, custom functions, and the task pane in separate runtime environments. This creates limitations, such as not being able to easily share global data, and not being able to access all CORS functionality from a custom function.

However, you can configure your Office Add-in to share code in the same runtime (also referred to as a shared runtime). This enables better coordination across your add-in and access to the task pane DOM and CORS from all parts of your add-in.

Configuring a shared runtime enables the following scenarios.

- Your Office Add-in can use additional UI features.
  - Change the availability of add-in commands
  - Run code in your Office Add-in when the document opens
  - Show or hide the task pane of your Office Add-in
  - Add custom keyboard shortcuts to your Office Add-ins (supported in Excel and Word add-ins only)
- The following are available for Excel add-ins only.
  - Create custom contextual tabs in Office Add-ins
  - Custom functions will have full CORS support.
  - Custom functions can call Office.js APIs to read spreadsheet document data.

For Office on Windows, the shared runtime uses WebView2 (Microsoft Edge Chromium-based) if the conditions for using it are met as explained in Browsers and webview controls used by Office Add-ins. Otherwise, it uses Trident (Internet Explorer 11). Additionally, any buttons that your add-in displays on the ribbon will run in the same shared runtime. The following image shows how custom functions, the ribbon UI, and the task pane code will all run in the same runtime.



Multiple task panes

Don't design your add-in to use multiple task panes if you are planning to use a shared runtime. A shared runtime only supports the use of one task pane. Note that any task pane without a `<TaskpaneID>` is considered a different task pane.

See also

- Call Excel APIs from a custom function
- Add custom keyboard shortcuts to your Office Add-ins
- Create custom contextual tabs in Office Add-ins
- Change the availability of add-in commands
- Run code in your Office Add-in when the document opens
- Show or hide the task pane of your Office Add-in
- Tutorial: Share data and events between Excel custom functions and the task pane
- Runtimes in Office Add-ins

Activate add-ins with events

08/01/2025

Event-based activation automatically triggers your add-in to complete their tasks without explicitly launching it. This allows the add-in to validate, insert, or refresh critical content without any manual operations. The add-in is opened in the background to avoid disrupting the user. You can also integrate event-based activation with the task pane and function commands.

Overview

While the particular steps to add event-based functionality to your add-in vary by platform and manifest type, the general flow is as follows.

1. Update the manifest with an extension for the event.
2. Connect the event in the manifest with a JavaScript function to handle the event.
3. Have the event handler function perform its actions, then call `event.completed` when it finishes.
4. Call Office.actions.associate to connect the event handler function with the ID specified in the manifest.

Try out event-based activation

Discover how to streamline workflows and improve user experiences with event-based activation. Try out the samples to see the feature in action.

Outlook samples

- Automatically set the subject of a new message or appointment
- Automatically check for an attachment before a message is sent
- Automatically update your signature when switching between mail accounts
- Encrypt attachments, process meeting request attendees, and react to appointment date/time changes using Outlook event-based activation
- Set your signature using Outlook event-based activation
- Identify and tag external recipients using Outlook event-based activation
- Verify the color categories of a message or appointment before it's sent using Smart Alerts
- Verify the sensitivity label of a message

Word samples

- Add headers when a document opens

Supported events

The following tables list events that are currently available and the supported clients for each event. When an event is raised, the handler receives an `event` object which may include details specific to the type of event. The **Description** column includes a link to the related object where applicable.

Excel, PowerPoint, Word events

Expand table

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Supported clients and channels
`OnDocumentOpened`	*Not yet supported*	Occurs when a user opens a document or creates a new document, spreadsheet, or presentation.	• Windows (Build >= 16.0.18324.20032) • Office on the web • Office on Mac will be available later

Outlook events

Support for this feature in Outlook was introduced in requirement set 1.10, with additional events now available in subsequent requirement sets. The following table lists each event's minimum requirement set and the clients and platforms that support it. For more information on Outlook clients and the requirement sets they support, see Requirement sets supported by Exchange servers and Outlook clients.

Expand table

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Minimum requirement set and supported clients
`OnNewMessageCompose`	newMessageComposeCreated	On composing a new message (includes reply, reply all, and forward) but not on editing, for example, a draft.	1.10 • Web browser • Windows (new and classic[1]) • New Mac UI[2] • Android[2] [3] • iOS[2] [3]
`OnNewAppointmentOrganizer`	newAppointmentOrganizerCreated	On creating a new appointment but not on editing an existing one.	1.10 • Web browser • Windows (new and classic[1]) • New Mac UI[2]

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Minimum requirement set and supported clients
`OnMessageAttachmentsChanged`	messageAttachmentsChanged	On adding or removing attachments while composing a message. Event-specific data object: AttachmentsChangedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnAppointmentAttachmentsChanged`	appointmentAttachmentsChanged	On adding or removing attachments while composing an appointment. Event-specific data object: AttachmentsChangedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnMessageRecipientsChanged`	messageRecipientsChanged	On adding or removing recipients while composing a message. Event-specific data object: RecipientsChangedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2] • Android[2] [3] • iOS[2] [3]
`OnAppointmentAttendeesChanged`	appointmentAttendeesChanged	On adding or removing attendees while composing an appointment. Event-specific data object: RecipientsChangedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnAppointmentTimeChanged`	appointmentTimeChanged	On changing date/time while composing an appointment. Event-specific data object:	1.11 • Web browser

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Minimum requirement set and supported clients
		AppointmentTimeChangedEventArgs **Important**: If you drag and drop an appointment to a different date/time slot on the calendar, the `OnAppointmentTimeChanged` event doesn't occur. It only occurs when the date/time is directly changed from an appointment.	• Windows (new and classic[1]) • New Mac UI[2]
`OnAppointmentRecurrenceChanged`	appointmentRecurrenceChanged	On adding, changing, or removing the recurrence details while composing an appointment. If the date/time is changed, the `OnAppointmentTimeChanged` event also occurs. Event-specific data object: RecurrenceChangedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnInfoBarDismissClicked`	infoBarDismissClicked	On dismissing a notification while composing a message or appointment item. Only the add-in that added the notification will be notified. Event-specific data object: InfobarClickedEventArgs	1.11 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnMessageSend`	messageSending	On sending a message item. To learn more, try the Smart Alerts walkthrough.	1.12 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnAppointmentSend`	appointmentSending	On sending an appointment item. To learn more, see Handle OnMessageSend and OnAppointmentSend events in your Outlook add-in with Smart Alerts.	1.12 • Web browser • Windows (new and classic[1])

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Minimum requirement set and supported clients
			• New Mac UI[2]
`OnMessageCompose`	messageComposeOpened	On composing a new message (includes reply, reply all, and forward) or editing a draft.	1.12 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnAppointmentOrganizer`	appointmentOrganizerOpened	On creating a new appointment or editing an existing one.	1.12 • Web browser • Windows (new and classic[1]) • New Mac UI[2]
`OnMessageFromChanged`	messageFromChanged	On changing the mail account in the **From** field of a message being composed. To learn more, see Automatically update your signature when switching between Exchange accounts.	1.13 • Web browser[4] • Windows (new[4] and classic[1]) • New Mac UI[2] • Android[2] [3] • iOS[2] [3]
`OnAppointmentFromChanged`	appointmentFromChanged	On changing the mail account in the organizer field of an appointment being composed. To learn more, see Automatically update your signature when switching between Exchange accounts.	1.13 • New Mac UI[2]
`OnSensitivityLabelChanged`	sensitivityLabelChanged	On changing the sensitivity label while composing a message or appointment. To learn how to manage the sensitivity label of a mail item, see Manage the sensitivity	1.13 • Web browser[4] • Windows (new[4]

Event canonical name and add-in only manifest name	Unified manifest for Microsoft 365 name	Description	Minimum requirement set and supported clients
		label of your message or appointment in compose mode. Event-specific data object: SensitivityLabelChangedEventArgs	and classic[1]) • New Mac UI[2]
`OnMessageReadWithCustomAttachment`	Not available	On opening a message that contains a specific attachment type in read mode.	Preview[5] • Windows (classic[1])
`OnMessageReadWithCustomHeader`	Not available	On opening a message that contains a specific internet header name in read mode.	Preview[5] • Windows (classic[1])

> **Note**
>
> [1] Event-based add-ins in classic Outlook on Windows require a minimum of Windows 10 Version 1903 (Build 18362) or Windows Server 2019 Version 1903 to run.
>
> [2] Add-ins that use the unified manifest for Microsoft 365 aren't directly supported in Outlook on Mac and on mobile devices. To run this type of add-in on Mac and on mobile platforms, the add-in must first be published to AppSource then deployed in the Microsoft 365 Admin Center. For more information, see the "Client and platform support" section of Office Add-ins with the unified app manifest for Microsoft 365.
>
> [3] For more information, see Implement event-based activation in Outlook mobile add-ins.
>
> [4] The `OnMessageFromChanged` and `OnSensitivityLabelChanged` events aren't currently available with the unified manifest for Microsoft 365 in Outlook on the web and the new Outlook on Windows. To handle these events, implement an add-in only manifest instead. For information about the types of manifests, see Office Add-ins manifest.
>
> [5] To preview the `OnMessageReadWithCustomAttachment` and `OnMessageReadWithCustomHeader` events, you must install classic Outlook on Windows Version 2312 (Build 17110.10000) or later. Then, join the Microsoft 365 Insider program and select the **Beta Channel** option to access Office beta builds.

Event-based activation in Outlook on mobile devices

Outlook on mobile supports APIs up to Mailbox requirement set 1.5. However, support is now enabled for additional APIs and features introduced in later requirement sets, such as the `OnNewMessageCompose` event. To learn more, see Implement event-based activation in Outlook mobile add-ins.

Behavior and limitations

As you develop an event-based add-in, be mindful of the following feature behaviors and limitations.

- Event-based add-ins work only when deployed by an administrator. If users install them directly from AppSource or the Office Store, they will not automatically launch (for workarounds to the AppSource limitation, see AppSource listing options for your event-based add-in). Admin deployments are done by uploading the manifest to the Microsoft 365 admin center.
- APIs that interact with the UI or display UI elements are not supported for Word, PowerPoint, and Excel on Windows. This is because the event handler runs in a JavaScript-only runtime. For more information, see Runtimes in Office Add-ins.
- Event-based add-ins require an internet connection to be able to launch when a specific event occurs. Add-in event handlers are expected to be short-running, lightweight, and as noninvasive as possible. After activation, your add-in will time out within approximately 300 seconds, the maximum length of time allowed for running event-based add-ins. To signal that your add-in has completed processing a launch event, your associated event handler must call the event.completed method. (Note that code included after the `event.completed` statement isn't guaranteed to run.) Each time an event that your add-in handles is triggered, the add-in is reactivated and runs the associated event handler, and the timeout window is reset. The add-in ends after it times out, or the user closes the compose window or sends the item.
- The behavior of multiple add-ins that subscribe to the same event isn't deterministic. Outlook launches the add-ins in no particular order. For Excel, PowerPoint, and Word, only one random add-in will be activated. For example, if multiple Word add-ins that handle `OnDocumentOpened`, only one of those handlers will run.
- Currently, only five event-based add-ins can be actively running.
- In all supported Outlook clients, the user must remain on the current mail item where the add-in was activated for it to complete running. Navigating away from the current item (for example, switching to another compose window or tab) terminates the add-in operation. However, an add-in that activates on the `OnMessageSend` event handles item switching differently depending on which Outlook client it's running on. To learn more, see the "User navigates away from current message" section of Handle OnMessageSend and OnAppointmentSend events in your Outlook add-in with Smart Alerts.
- In addition to item switching, an event-based add-in also ceases operation when the user sends the message or appointment they're composing.

Event-based add-in limitations in the new Outlook on Windows

In the new Outlook on Windows, you must keep the main window of the client open for the add-in to process the mail item. If the main window is minimized, the add-in will pause or stop working.

Event-based add-in limitations in Excel, PowerPoint, Word, and classic Outlook on Windows

When developing an event-based add-in to run on a Windows client, be mindful of the following:

- Imports aren't supported in the JavaScript file where you implement the handling for event-based activation.
- Only the JavaScript file referenced in the manifest is supported for event-based activation. You must bundle your event-handling JavaScript code into this single file. The location of the referenced JavaScript file in the manifest varies depending on the type of manifest your add-in uses.

- **Add-in only manifest**: `<Override>` child element of the `<Runtime>` node
- **Unified manifest for Microsoft 365**: `"script"` property of the `"code"` object

  Note that a large JavaScript bundle may cause issues with the performance of your add-in. We recommend preprocessing heavy operations, so that they're not included in your event-handling code.

- When the JavaScript function specified in the manifest to handle an event runs, code in `Office.onReady()` and `Office.initialize` isn't run. We recommend adding any startup logic needed by event handlers, such as checking the user's client version, to the event handlers instead.

Event-based add-in limitations in Excel, PowerPoint, and Word

The following platforms or features are not yet supported.

- Office on Mac
- The unified manifest for Microsoft 365

Unsupported APIs

Some Office.js APIs that change or alter the UI aren't allowed from event-based add-ins. The following are blocked APIs.

Expand table

API	Methods
`Office.devicePermission`	• `requestPermissionsAsync`
`Office.context.auth`*	• `getAccessToken` • `getAccessTokenAsync`
`Office.context.mailbox`	• `displayAppointmentForm` • `displayMessageForm` • `displayNewAppointmentForm` • `displayNewMessageForm`
`Office.context.mailbox.item`	• `close`
`Office.context.ui`	• `displayDialogAsync` • `messageParent`

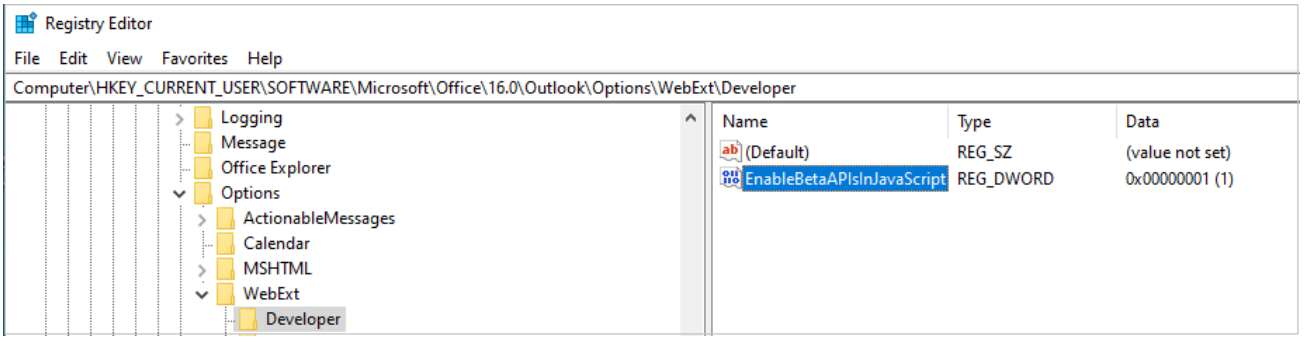
> **Note**
>
> * OfficeRuntime.auth is supported in all versions that support event-based activation and single sign-on (SSO), while Office.auth is only supported in certain Outlook builds. For more information, see Use single sign-on (SSO) or cross-origin resource sharing (CORS) in your event-based or spam-reporting Outlook add-in.

Preview features in event handlers (classic Outlook on Windows)

Classic Outlook on Windows includes a local copy of the production and beta versions of Office.js instead of loading from the content delivery network (CDN). By default, the local production copy of the API is referenced. To reference the local beta copy of the API, you must configure your computer's registry. This will enable you to test preview features in your event handlers in classic Outlook on Windows.

1. In the registry, navigate to `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Outlook\Options\WebExt\Developer`. If the key doesn't exist, create it.

2. Create an entry named `EnableBetaAPIsInJavaScript` and set its value to `1`.



Enable single sign-on (SSO)

To enable SSO in your event-based add-in, you must add its JavaScript file to a well-known URI. For guidance on how to configure this resource, see Use single sign-on (SSO) or cross-origin resource sharing (CORS) in your event-based or spam-reporting Office Add-in.

Request external data

You can request external data by using an API like Fetch or by using XMLHttpRequest (XHR), a standard web API that issues HTTP requests to interact with servers.

> **Note**
>
> If your add-in will operate in a JavaScript-only runtime, use absolute URLs in your Fetch API calls. Relative URLs in Fetch API calls aren't supported in a JavaScript-only runtime.

Be aware that you must use additional security measures when using XMLHttpRequest objects, requiring Same Origin Policy and CORS (Cross-Origin Resource Sharing).

> **Note**
>
> Full CORS support is available in Office on the web, Mac, and Windows (starting in Version 2201, Build 16.0.14813.10000) clients.

To make CORS requests from your event-based add-in, you must add the add-in and its JavaScript file to a well-known URI. For guidance on how to configure this resource, see Use single sign-on (SSO) or cross-origin resource

sharing (CORS) in your event-based or spam-reporting Office Add-in.

Troubleshoot your add-in

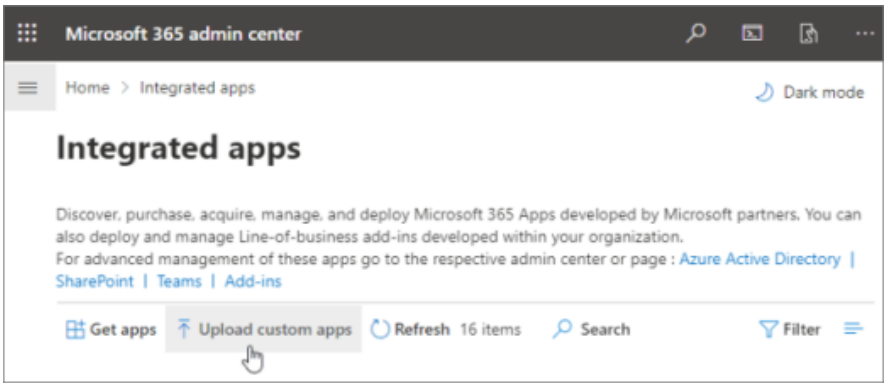
As you develop your event-based add-in, you may need to troubleshoot issues, such as your add-in not loading or the event not occurring. For guidance on how to troubleshoot an event-based add-in, see Troubleshoot event-based and spam-reporting add-ins.

Deploy your add-in

Event-based add-ins are restricted to admin-managed deployments only, even if they're acquired from AppSource. If users acquire the add-in from AppSource or the in-app Office Store, they won't be able to activate the event-based function of the add-in. To learn more about listing your event-based add-in in AppSource, see AppSource listing options for your event-based add-in.

Admin deployments are done by uploading the manifest to the Microsoft 365 admin center. To do so, follow these steps.

1. In the admin portal, expand the **Settings** section in the navigation pane then select **Integrated apps**.
2. On the **Integrated apps** page, choose the **Upload custom apps** action.



> **Important**
>
> Add-ins that use the Smart Alerts feature can only be published to AppSource if the manifest's send mode property is set to the **prompt user** or **soft block** option. If an add-in's send mode property is set to **block**, it can only be deployed by an organization's admin as it will fail AppSource validation.

For more information about how to deploy an add-in, please refer to Deploy and publish Office Add-ins in the Microsoft 365 admin center.

Deploy manifest updates

Because event-based add-ins are deployed by admins, any change you make to the manifest requires admin consent through the Microsoft 365 admin center. Until the admin accepts your changes, users in their organization are blocked from using the add-in. To learn more about the admin consent process, see Admin consent for installing event-based add-ins.

See also

- Troubleshoot event-based and spam-reporting add-ins
- Debug event-based and spam-reporting add-ins
- AppSource listing options for your event-based add-in
- Handle OnMessageSend and OnAppointmentSend events in your Outlook add-in with Smart Alerts

Run code in your Office Add-in when the document opens

Article • 07/11/2023

> **Important**
>
> The shared runtime is only supported in some Office applications. For more information, see **Shared runtime requirement sets**.

You can configure your Office Add-in to load and run code as soon as the document is opened. This is useful if you need to register event handlers, pre-load data for the task pane, synchronize UI, or perform other tasks before the add-in is visible.

> **Note**
>
> The configuration is implemented with a method that your code calls at runtime. This means that the add-in *won't* run the *first time* a user opens the document. The add-in must be opened manually for the first time on any document. After the method runs, either in **Office.initialize**, **Office.onReady**, or because the user takes a code path that runs it; then whenever the document is reopened, the add-in loads immediately and any code in the `Office.initialize` or `Office.onReady` method runs.

> **Note**
>
> This article requires that your Office Add-in is configured to use a **shared runtime**. For more information, see **Configure your Office Add-in to use a shared runtime**.

Configure your add-in to load when the document opens

The following code configures your add-in to load and start running when the document is opened.

JavaScript

```
Office.addin.setStartupBehavior(Office.StartupBehavior.load);
```

> **Note**
>
> The `setStartupBehavior` method is asynchronous.

Place startup code in Office.initialize or Office.onReady

When your add-in is configured to load on document open, it will run immediately. The `Office.initialize` event handler will be called. Place your startup code in the `Office.initialize` or `Office.onReady` event handler.

The following Excel add-in code shows how to register an event handler for change events from the active worksheet. If you configure your add-in to load on document open, this code will register the event handler when the document is opened. You can handle change events before the task pane is opened.

JavaScript

```
// This is called as soon as the document opens.
// Put your startup code here.
Office.initialize = () => {
  // Add the event handler.
  Excel.run(async context => {
    let sheet = context.workbook.worksheets.getActiveWorksheet();
    sheet.onChanged.add(onChange);

    await context.sync();
    console.log("A handler has been registered for the onChanged event.");
  });
};

/**
 * Handle the changed event from the worksheet.
 *
 * @param event The event information from Excel
 */
async function onChange(event) {
    await Excel.run(async (context) => {
        await context.sync();
        console.log("Change type of event: " + event.changeType);
        console.log("Address of event: " + event.address);
        console.log("Source of event: " + event.source);
```

```
  });
}
```

The following PowerPoint add-in code shows how to register an event handler for selection change events from the PowerPoint document. If you configure your add-in to load on document open, this code will register the event handler when the document is opened. You can handle change events before the task pane is opened.

JavaScript

```
// This is called as soon as the document opens.
// Put your startup code here.
Office.onReady(info => {
  if (info.host === Office.HostType.PowerPoint) {

Office.context.document.addHandlerAsync(Office.EventType.DocumentSelectionCh
anged, onChange);
    console.log("A handler has been registered for the onChanged event.");
  }
});

/**
 * Handle the changed event from the PowerPoint document.
 *
 * @param event The event information from PowerPoint
 */
async function onChange(event) {
  console.log("Change type of event: " + event.type);
}
```

Configure your add-in for no load behavior on document open

There may be scenarios when you want to turn off the "run on document open" behavior. The following code configures your add-in not to start when the document is opened. Instead, it will start when the user engages it in some way, such as choosing a ribbon button or opening the task pane. This code has no effect if the method hasn't previously been called on the current document, with `Office.StartupBehavior.load` as the parameter.

> ⓘ Note
>
> If add-in calls the method, with `Office.StartupBehavior.load` as the parameter, in `Office.initialize` or `Office.onReady`, the behavior is turned on again. So, in this

scenario, turning it off only applies to the *next* time the document is opened, not *all* subsequent openings.

JavaScript

```
Office.addin.setStartupBehavior(Office.StartupBehavior.none);
```

Get the current load behavior

There may be scenarios in which your add-in needs to know if it's configured to start automatically the next time the current document is opened. To determine what the current startup behavior is, run the following method, which returns an Office.StartupBehavior value.

JavaScript

```
let behavior = await Office.addin.getStartupBehavior();
```

See also

- Configure your Office Add-in to use a shared runtime
- Share data and events between Excel custom functions and task pane tutorial
- Work with Events using the Excel JavaScript API
- Runtimes in Office Add-ins

Office Add-ins manifest

Article • 02/12/2025

Every Office add-in has a manifest. There are two types of manifests:

- **Add-in only manifest**: This type of manifest can be used for production add-ins in Excel, OneNote, Outlook, PowerPoint, Project, and Word. It can't be used for an app that combines an add-in with some other kind of extension of the Microsoft 365 platform. Its format is XML.
- **Unified manifest for Microsoft 365**: This is an expanded version of the JSON-formatted manifest that has been used for years as the manifest for Teams Apps. Add-ins that use this manifest can be combined with other kinds of extensions of the Microsoft 365 platform in a single app that's installable as a unit. You can use this type of manifest for production Outlook add-ins. It's available for preview in Excel, PowerPoint, and Word add-ins.

> **Note**
>
> Office Add-ins that use the unified manifest for Microsoft 365 are *directly* supported in Office on the web, in new Outlook on Windows, and in Office on Windows connected to a Microsoft 365 subscription, Version 2304 (Build 16320.00000) or later.
>
> When the app package that contains the unified manifest is deployed in AppSource or the Microsoft 365 Admin Center then an add-in only manifest is generated from the unified manifest and stored. This add-in only manifest enables the add-in to be installed on platforms that don't directly support the unified manifest, including Office on Mac, Office on mobile, subscription versions of Office on Windows earlier than 2304 (Build 16320.00000), and perpetual versions of Office on Windows.

The remainder of this article is applicable to both types of manifest.

> **Tip**
>
> - For an overview that is specific to the add-in only manifest, see Office Add-ins with an add-in only manifest.
> - For an overview that's specific to the unified manifest, see Office Add-ins with the unified manifest for Microsoft 365.

- If you have some familiarity with the add-in only manifest, the article Compare the add-in only manifest with the unified manifest for Microsoft 365 explains the unified manifest by comparing it with the add-in only manifest.

The 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.

A manifest file enables an Office Add-in to do the following:

- Describe itself by providing an ID, version, description, display name, and default locale.
- Specify the images used for branding the add-in and iconography used for add-in commands in the Office app ribbon.
- 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.

> **Note**
>
> If you plan to publish your add-in to AppSource and make it available within the Office experience, make sure that you conform to the Commercial marketplace certification policies. 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 1120.3 and the Office Add-in application and availability page).

Hosting requirements

All image URIs, such as those used for add-in commands, must support caching in production. The server hosting the image shouldn't return a `Cache-Control` header specifying `no-cache`, `no-store`, or similar options in the HTTP response. However, when you're developing the add-in and making changes to image files, the caching can prevent you from seeing your changes, so using `Cache-Control` headers is advisable in development.

All URLs to code or content files in the add-in should be **SSL-secured (HTTPS)**. While not strictly required in all add-in scenarios, using an HTTPS endpoint for your add-in is strongly recommended. Add-ins that are not SSL-secured (HTTPS) generate unsecure content warnings and errors during use. If you plan to run your add-in in Office on the web or publish your add-in to AppSource, it must be SSL-secured. If your add-in accesses external data and services, it should be SSL-secured to protect data in transit. Self-signed certificates can be used for development and testing, so long as the certificate is trusted on the local machine.

Best practices for submitting to AppSource

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.

Add-ins submitted to AppSource must also include a support URL in the manifest. For more information, see Validation policies for apps and add-ins submitted to AppSource.

Specify domains you want to open in the add-in window

When running in Office on the web or new Outlook on Windows, your task pane can be navigated to any URL. However, in desktop platforms, 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 manifest file), that URL opens in a new browser window outside the add-in pane of the Office application.

To override this (desktop Office) behavior, specify each domain you want to open in the add-in window in the manifest. If the add-in tries to go to a URL in a domain that is in the list, then it opens in the task pane in both Office on the web and desktop. If it tries to go to a URL that isn't in the list, then, in desktop Office, that URL opens in a new browser window (outside the add-in pane).

> **Note**
>
> There are two exceptions to this behavior.
>
> - It applies only to the root pane of the add-in. If there is an iframe embedded in the add-in page, the iframe can be directed to any URL regardless of whether it is listed in the manifest, even in desktop Office.

- When a dialog is opened with the displayDialogAsync API, the URL that is passed to the method must be in the same domain as the add-in, but the dialog can then be directed to any URL regardless of whether it is listed in the manifest, even in desktop Office.

Specify domains from which Office.js API calls are made

Your add-in can make Office.js API calls from the add-in's domain referenced in the manifest file. If you have other iframes within your add-in that need to access Office.js APIs, add the domain of that source URL to the manifest file. If an iframe with a source not listed in the manifest attempts to make an Office.js API call, then the add-in will receive a permission denied error.

See also

- Specify Office applications and API requirements
- Localization for Office Add-ins

Office Add-ins with the unified app manifest for Microsoft 365

09/02/2025

This article introduces the unified app manifest for Microsoft 365. It assumes that you're familiar with Office Add-ins manifest.

> **Tip**
>
> - For an overview of the add-in only manifest, see **Office Add-ins with the add-in only manifest**.
> - If you're familiar with the add-in only manifest, you might get a grasp on the JSON-formatted unified manifest easier by reading **Compare the add-in only manifest with the unified manifest for Microsoft 365**.

Microsoft is making a number of improvements to the Microsoft 365 developer platform. These improvements provide more consistency in the development, deployment, installation, and administration of all types of extensions of Microsoft 365, including Office Add-ins. These changes are compatible with existing add-ins.

One important improvement is the ability to create a single unit of distribution for all your Microsoft 365 extensions by using the same manifest format and schema.

We've taken an important first step toward these goals by making it possible for you to create Outlook add-ins with a unified manifest for Microsoft 365.

> **Note**
>
> The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

> **Tip**
>
> Ready to get started with the unified manifest? Begin with **Build an Outlook add-in with the unified manifest for Microsoft 365**.

Key properties of the unified manifest

The main reference documentation for the version of the unified app manifest is at Microsoft 365 app manifest schema reference. In this article, we provide a brief description of the meaning of base properties when the Teams App is (or includes) an Office Add-in. This is followed by some basic documentation for the "extensions" property and its descendant properties. There is a full sample manifest for an add-in at Sample unified manifest.

Base properties

Each of the base properties listed in the following table has more extensive documentation at Microsoft 365 app manifest schema. Base properties not included in this table have no meaning for Office Add-ins.

Expand table

JSON property	Purpose
"$schema"	Identifies the manifest schema.
"manifestVersion"	Version of the manifest schema.
`"id"`	GUID of the Teams app/add-in.
"version"	Version of the Teams app/add-in. The format must be `n.n.n` where each `n` can be no more than five digits.
"name"	Public short and long names of the Teams app/add-in. The short name appears at the top of an add-in's task pane.
"description"	Public short and long descriptions of the Teams app/add-in.
"developer"	Information about the developer of the Teams app/add-in.
"localizationInfo"	Configures the default locale and other supported locales.
"validDomains"	See Specify safe domains.
"webApplicationInfo"	Identifies the Teams app/add-in's web app as it is known in Microsoft Entra ID.
"authorization"	Identifies any Microsoft Graph permissions that the add-in needs.

`"extensions"` property

We're working hard to complete reference documentation for the `"extensions"` property and its descendant properties. In the meantime, the following provides some basic documentation. Most, but not all, of the properties have an equivalent element (or attribute) in the add-in only manifest for add-ins. For the most part, the description, and restrictions, that apply to the XML

element or attribute also apply to its JSON property equivalent in the unified manifest. The tables in the '`"extensions"` property' section of Compare the add-in only manifest with the unified manifest for Microsoft 365 can help you determine the XML equivalent of a JSON property.

> **Note**
>
> This table contains only some selected representative descendant properties of `"extensions"`. *It isn't an exhaustive list of all child properties of* `"extensions"`. For the full reference of the unified manifest, see the Microsoft 365 app manifest schema reference.

Expand table

JSON property	Purpose
"requirements.capabilities"	Identifies the requirement sets that the add-in needs to be installable.
"requirements.scopes"	Identifies the Office applications in which the add-in can be installed. For example, `"mail"` means the add-in can be installed in Outlook.
"ribbons"	The ribbons that the add-in customizes.
`"ribbons.contexts"`	Specifies the command surfaces that the add-in customizes. For example, `"mailRead"` or `"mailCompose"`.
`"ribbons.fixedControls"`	Configures and adds the button of an integrated spam-reporting add-in to the Outlook ribbon.
`"ribbons.spamPreProcessingDialog"`	Configures the preprocessing dialog shown after the button of a spam-reporting add-in is selected from the Outlook ribbon.
`"ribbons.tabs"`	Configures custom ribbon tabs.
"alternates"	Specifies backwards compatibility with an equivalent COM add-in, XLL, or both. Also specifies the main icons that are used to represent the add-in on older versions of Office.
"runtimes"	Configures the embedded runtimes that the add-in uses, including various kinds of add-ins that have little or no UI, such as custom function-only add-ins and function commands.
"autoRunEvents"	Configures an event handler for a specified event.
"keyboardShortcuts" (developer preview)	Defines custom keyboard shortcuts or key combinations to run specific actions.

Specify safe domains

There is a `"validDomains"` array in the manifest file that is used to tell Office which domains your add-in should be allowed to navigate to. As noted in Specify domains you want to open in the add-in window, when running in Office on the web, your task pane can be navigated to any URL. However, in desktop platforms, if your add-in tries to go to a URL in a domain other than the domain that hosts the start page, that URL opens in a new browser window outside the add-in pane of the Office application.

To override this behavior in desktop platforms, add each domain you want to open in the add-in window to the list of domains specified in the `"validDomains"` array. If the add-in tries to go to a URL in a domain that is in the list, then it opens in the task pane in both Office on the web and desktop. If it tries to go to a URL that isn't in the list, then in Office on desktop, that URL opens in a new browser window (outside the add-in task pane).

Client and platform support

Add-ins that use the unified manifest can be installed if the Office platform *directly* supports it.

To run an add-in on platforms that don't directly support the unified manifest, you must publish the add-in to AppSource. Then, deploy the add-in in the Microsoft 365 admin center. This way, an add-in only manifest is generated from the unified manifest and stored. The add-in only manifest is then used to install the add-in on platforms that don't directly support the unified manifest.

The following tables lists which Office platforms directly support add-ins that use the unified manifest.

Expand table

Client/platform	Support for add-ins with the unified manifest
Office on the web	Directly supported
Office on Windows (Version 2304 (Build 16320.00000) or later) connected to a Microsoft 365 subscription	Directly supported
new Outlook on Windows	Directly supported
Office on Windows (prior to Version 2304 (Build 16320.00000)) connected to a Microsoft 365 subscription	Not directly supported
Office on Windows (perpetual versions)	Not directly supported

Client/platform	Support for add-ins with the unified manifest
Office on Mac	Not directly supported
Office on mobile	Not directly supported

> **Note**
>
> If you're deploying an add-in in the Microsoft 365 admin center and require it to run on platforms that don't directly support the unified manifest, the add-in must be a published AppSource add-in. Custom add-ins or line-of-business (LOB) add-ins that use the unified manifest can be deployed in the Integrated apps portal of the Microsoft 365 admin center, but they won't be installable on Office versions that don't directly support the unified manifest.

Sample unified manifest

The following is an example of a unified app manifest for an add-in. It doesn't contain every possible manifest property.

JSON

```
{
  "$schema": "https://developer.microsoft.com/json-
schemas/teams/vDevPreview/MicrosoftTeams.schema.json",
  "id": "00000000-0000-0000-0000-000000000000",
  "version": "1.0.0",
  "manifestVersion": "devPreview",
  "name": {
    "short": "Name of your app (<=30 chars)",
    "full": "Full name of app, if longer than 30 characters (<=100 chars)"
  },
  "description": {
    "short": "Short description of your app (<= 80 chars)",
    "full": "Full description of your app (<= 4000 chars)"
  },
  "icons": {
    "outline": "outline.png",
    "color": "color.png"
  },
  "accentColor": "#230201",
  "developer": {
    "name": "Contoso",
    "websiteUrl": "https://www.contoso.com",
    "privacyUrl": "https://www.contoso.com/privacy",
    "termsOfUseUrl": "https://www.contoso.com/servicesagreement"
```

```
  },
  "localizationInfo": {
    "defaultLanguageTag": "en-us",
    "additionalLanguages": [
      {
        "languageTag": "es-es",
        "file": "es-es.json"
      }
    ]
  },
  "webApplicationInfo": {
    "id": "00000000-0000-0000-0000-000000000000",
    "resource": "api://www.contoso.com/prodapp"
  },
  "authorization": {
    "permissions": {
      "resourceSpecific": [
        {
          "name": "Mailbox.ReadWrite.User",
          "type": "Delegated"
        }
      ]
    }
  },
  "extensions": [
    {
      "requirements": {
        "scopes": [ "mail" ],
        "capabilities": [
          {
            "name": "Mailbox", "minVersion": "1.1"
          }
        ]
      },
      "runtimes": [
        {
          "requirements": {
            "capabilities": [
              {
                "name": "MailBox",
                "minVersion": "1.10"
              }
            ]
          },
          "id": "eventsRuntime",
          "type": "general",
          "code": {
            "page": "https://contoso.com/events.html",
            "script": "https://contoso.com/events.js"
          },
          "lifetime": "short",
          "actions": [
            {
              "id": "onMessageSending",
              "type": "executeFunction"
```

```
            },
            {
              "id": "onNewMessageComposeCreated",
              "type": "executeFunction"
            }
          ]
        },
        {
          "requirements": {
            "capabilities": [
              {
                "name": "MailBox", "minVersion": "1.1"
              }
            ]
          },
          "id": "commandsRuntime",
          "type": "general",
          "code": {
            "page": "https://contoso.com/commands.html",
            "script": "https://contoso.com/commands.js"
          },
          "lifetime": "short",
          "actions": [
            {
              "id": "action1",
              "type": "executeFunction"
            },
            {
              "id": "action2",
              "type": "executeFunction"
            },
            {
              "id": "action3",
              "type": "executeFunction"
            }
          ]
        }
      ],
      "ribbons": [
        {
          "contexts": [
            "mailCompose"
          ],
          "tabs": [
            {
              "builtInTabId": "TabDefault",
              "groups": [
                {
                  "id": "dashboard",
                  "label": "Controls",
                  "controls": [
                    {
                      "id": "control1",
                      "type": "button",
                      "label": "Action 1",
```

```
      "icons": [
        {
          "size": 16,
          "url": "test_16.png"
        },
        {
          "size": 32,
          "url": "test_32.png"
        },
        {
          "size": 80,
          "url": "test_80.png"
        }
      ],
      "supertip": {
        "title": "Action 1 Title",
        "description": "Action 1 Description"
      },
      "actionId": "action1"
    },
    {
      "id": "menu1",
      "type": "menu",
      "label": "My Menu",
      "icons": [
        {
          "size": 16,
          "url": "test_16.png"
        },
        {
          "size": 32,
          "url": "test_32.png"
        },
        {
          "size": 80,
          "url": "test_80.png"
        }
      ],
      "supertip": {
        "title": "My Menu",
        "description": "Menu with 2 actions"
      },
      "items": [
        {
          "id": "menuItem1",
          "type": "menuItem",
          "label": "Action 2",
          "supertip": {
            "title": "Action 2 Title",
            "description": "Action 2 Description"
          },
          "actionId": "action2"
        },
        {
          "id": "menuItem2",
```

```
                    "type": "menuItem",
                    "label": "Action 3",
                    "icons": [
                      {
                        "size": 16,
                        "url": "test_16.png"
                      },
                      {
                        "size": 32,
                        "url": "test_32.png"
                      },
                      {
                        "size": 80,
                        "url": "test_80.png"
                      }
                    ],
                    "supertip": {
                      "title": "Action 3 Title",
                      "description": "Action 3 Description"
                    },
                    "actionId": "action3"
                  }
                ]
              }
            ]
          }
        ],
      }
    ]
  },
  {
    "contexts": [ "mailRead" ],
    "tabs": [
      {
        "builtInTabId": "TabDefault",
        "groups": [
          {
            "id": "dashboard",
            "label": "Controls",
            "controls": [
              {
                "id": "control1",
                "type": "button",
                "label": "Action 1",
                "icons": [
                  {
                    "size": 16,
                    "url": "test_16.png"
                  },
                  {
                    "size": 32,
                    "url": "test_32.png"
                  },
                  {
                    "size": 80,
```

```
                "url": "test_80.png"
              }
            ],
            "supertip": {
              "title": "Action 1 Title",
              "description": "Action 1 Description"
            },
            "actionId": "action1"
          }
        ]
      }
    ],
    "customMobileRibbonGroups" [
      {
        "id": "myMobileGroup",
        "label": "Contoso Actions",
        "controls": [
          {
            "id": "msgReadFunctionButton",
            "type": "mobileButton",
            "label": "Action 1",
            "icons": [
              {
                "size": 16,
                "url": "test_16.png"
              },
              {
                "size": 32,
                "url": "test_32.png"
              },
              {
                "size": 80,
                "url": "test_80.png"
              }
            ],
            "supertip": {
              "title": "Action 1 Title",
              "description": "Action 1 Description"
            },
            "actionId": "action1"
          }
        ]
      }
    ]
    "customMobileRibbonGroups": [
      {
        "id": "mobileDashboard",
        "label": "Controls",
        "controls": [
          {
            "id": "control1",
            "type": "mobileButton",
            "label": "Action 1",
            "icons": [
              {
```

```
                    "size": 16,
                    "url": "test_16.png"
                  },
                  {
                    "size": 32,
                    "url": "test_32.png"
                  },
                  {
                    "size": 80,
                    "url": "test_80.png"
                  }
                ],
                "supertip": {
                  "title": "Action 1 Title",
                  "description": "Action 1 Description"
                },
                "actionId": "action1"
              }
            ]
          }
        ]
      }
    ],
    "autoRunEvents": [
      {
        "requirements": {
          "capabilities": [
            {
              "name": "MailBox", "minVersion": "1.10"
            }
          ]
        },
        "events": [
          {
            "type": "newMessageComposeCreated",
            "actionId": "onNewMessageComposeCreated"
          },
          {
            "type": "messageSending",
            "actionId": "onMessageSending",
            "options": {
              "sendMode": "promptUser"
            }
          }
        ]
      }
    ],
    "alternates": [
      {
        "requirements": {
          "scopes": [ "mail" ]
        },
        "prefer": {
```

```
            "comAddin": {
              "progId": "ContosoExtension"
            }
          },
          "hide": {
            "storeOfficeAddin": {
              "officeAddinId": "00000000-0000-0000-0000-000000000000",
              "assetId": "WA000000000"
            }
          },
          "alternateIcons": {
            "icon": {
              "size": 64,
              "url": "https://contoso.com/assets/icon64x64.jpg"
            },
            "highResolutionIcon": {
              "size": 64,
              "url": "https://contoso.com/assets/icon128x128.jpg"
            }
          }
        }
      ]
    }
  ]
}
```

See also

- Create add-in commands with the unified manifest for Microsoft 365
- Microsoft 365 app manifest schema reference

Compare the add-in only manifest with the unified manifest for Microsoft 365

09/02/2025

This article is intended to help readers who are familiar with the add-in only manifest understand the unified manifest by comparing the two. Readers should also see Office Add-ins with the unified manifest for Microsoft 365.

> **Note**
>
> The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

Schemas and general points

There is just one schema for the unified manifest, in contrast to the add-in only manifest which has a total of seven schemas.

Conceptual mapping of the unified and add-in only manifests

This section describes the unified manifest for readers who are familiar with the add-in only manifest. Some points to keep in mind:

- The unified manifest is JSON-formatted.
- JSON doesn't distinguish between attribute and element value like XML does. Typically, the JSON that maps to an XML element makes both the element value and each of the attributes a child property. The following example shows some XML markup and its JSON equivalent.

  XML

  ```xml
  <MyThing color="blue">Some text</MyThing>
  ```

  JSON

  ```
  "myThing" : {
      "color": "blue",
  ```

```
    "text": "Some text"
}
```

- There are many places in the add-in only manifest where an element with a plural name has children with the singular version of the same name. For example, the markup to configure a custom menu includes an `<Items>` element which can have multiple `<Item>` element children. The JSON equivalent of these plural elements is a property with an array as its value. The members of the array are *anonymous* objects, not properties named "item" or "item1", "item2", etc. The following is an example.

  JSON

```
"items": [
    {
        -- markup for a menu item is here --
    },
    {
        -- markup for another menu item is here --
    }
]
```

Top-level structure

The root level of the unified manifest, which roughly corresponds to the `<OfficeApp>` element in the add-in only manifest, is an anonymous object.

The children of `<OfficeApp>` are commonly divided into two notional categories. The `<VersionOverrides>` element is one category. The other consists of all the other children of `<OfficeApp>`, which are collectively referred to as the base manifest. So too, the unified manifest has a similar division. There is a top-level "extensions" property that roughly corresponds in its purposes and child properties to the `<VersionOverrides>` element. The unified manifest also has over 10 other top-level properties that collectively serve the same purposes as the base manifest of the add-in only manifest. These other properties can be thought of collectively as the base manifest of the unified manifest.

Base manifest

The base manifest properties specify characteristics of the add-in that *any* type of extension of Microsoft 365 is expected to have. This includes Teams tabs and message extensions, not just Office add-ins. These characteristics include a public name and a unique ID. The following table shows a mapping of some critical top-level properties in the unified manifest to the XML elements in the current manifest, where the mapping principle is the *purpose* of the markup.

Expand table

JSON property	Purpose	XML elements	Comments
"$schema"	Identifies the manifest schema.	attributes of `<OfficeApp>` and `<VersionOverrides>`	*None*
"id"	GUID of the add-in.	`<Id>`	*None*
"version"	Version of the add-in.	`<Version>`	*None*
"manifestVersion"	Version of the manifest schema.	attributes of `<OfficeApp>`	*None*
"name"	Public name of the add-in.	`<DisplayName>`	*None*
"description"	Public description of the add-in.	`<Description>`	*None*
"accentColor"	*None*	*None*	This property has no equivalent in the add-in only manifest and isn't used in the unified manifest. But it must be present.
"developer"	Identifies the developer of the add-in.	`<ProviderName>`	*None*
"localizationInfo"	Configures the default locale and other supported locales.	`<DefaultLocale>` and `<Override>`	*None*
"webApplicationInfo"	Identifies the add-in's web app as it is known in Microsoft Entra ID.	`<WebApplicationInfo>`	In the add-in only manifest, the `<WebApplicationInfo>` element is inside `<VersionOverrides>`, not the base manifest.
"authorization"	Identifies any Microsoft Graph permissions that the add-in needs.	`<WebApplicationInfo>`	In the add-in only manifest, the `<WebApplicationInfo>` element is inside `<VersionOverrides>`, not the base manifest.

The `<Hosts>`, `<Requirements>`, and `<ExtendedOverrides>` elements are part of the base manifest in the add-in only manifest. But concepts and purposes associated with these elements are

configured inside the `"extensions"` property of the unified manifest.

`"extensions"` property

The `"extensions"` property in the unified manifest primarily represents characteristics of the add-in that wouldn't be relevant to other kinds of Microsoft 365 extensions. For example, the Office applications that the add-in extends (such as, Excel, PowerPoint, Word, and Outlook) are specified inside the `"extensions"` property, as are customizations of the Office application ribbon. The configuration purposes of the `"extensions"` property closely match those of the `<VersionOverrides>` element in the add-in only manifest.

> **Note**
>
> The `<VersionOverrides>` section of the add-in only manifest has a "double jump" system for many string resources. Strings, including URLs, are specified and assigned an ID in the `<Resources>` child of `<VersionOverrides>`. Elements that require a string have a `resid` attribute that matches the ID of a string in the `<Resources>` element. The `"extensions"` property of the unified manifest simplifies things by defining strings directly as property values. There is nothing in the unified manifest that is equivalent to the `<Resources>` element.

The following table shows a mapping of *some* high-level child properties of the `"extensions"` property in the unified manifest to XML elements in the current manifest. Dot notation is used to reference child properties.

> **Note**
>
> This table contains only some selected representative descendant properties of `"extensions"`. *It isn't an exhaustive list of all child properties of `"extensions"`*. For the full reference of the unified manifest, see Microsoft 365 app manifest schema reference.

Expand table

JSON property	Purpose	XML elements	Comments
"requirements.capabilities"	Identifies the requirement sets that the add-in needs to be installable. that the	`<Requirements>` and `<Sets>`	*None*

JSON property	Purpose	XML elements	Comments
	add-in needs to be installable.		
"requirements.scopes"	Identifies the Office applications in which the add-in can be installed.	`<Hosts>`	*None*
"ribbons"	The ribbons that the add-in customizes.	`<Hosts>`, **ExtensionPoints**, and various ***FormFactor** elements	The `"ribbons"` property is an array of anonymous objects that each merge the purposes of the these three elements. See "ribbons" table.
"alternates"	Specifies backwards compatibility with an equivalent COM add-in, XLL, or both.	`<EquivalentAddins>`	See the EquivalentAddins - See also for background information.
"runtimes"	Configures the embedded runtimes that the add-in uses, including various kinds of add-ins that have little or no UI, such as custom function-only add-ins and function commands.	`<Runtimes>`. `<FunctionFile>`, and `<ExtensionPoint>` (of type CustomFunctions)	*None*.
"autoRunEvents"	Configures an event handler for a specified event.	`<ExtensionPoint>` (of type LaunchEvent)	*None*.
"keyboardShortcuts" (developer preview)	Defines custom keyboard shortcuts or key combinations to run specific actions.	`<ExtendedOverrides>`	*None*.

`"ribbons"` table

The following table maps the child properties of the anonymous child objects in the `"ribbons"` array onto XML elements in the current manifest.

Expand table

JSON property	Purpose	XML elements	Comments
`"contexts"`	Specifies the command surfaces that the add-in customizes.	Various ***CommandSurface** elements, such as **PrimaryCommandSurface** and **MessageReadCommandSurface**	*None*.
`"tabs"`	Configures custom ribbon tabs.	`<CustomTab>`	The names and hierarchy of the descendant properties of `"tabs"` closely match the descendants of `<CustomTab>`.
`"fixedControls"`	Configures and adds the button of an integrated spam-reporting add-in to the Outlook ribbon.	`<Control>` child element of `<ReportPhishingCustomization>`	*None*.
`"spamPreProcessingDialog"`	Configures the preprocessing dialog shown after the button of a spam-reporting add-in is selected from the Outlook ribbon.	`<PreProcessingDialog>` child element of `<ReportPhishingCustomization>`	*None*.

For a full sample unified manifest, see Sample unified manifest.

Next steps

- Build your first Outlook add-in

Convert an add-in to use the unified manifest for Microsoft 365

Article • 05/19/2025

To add Teams capabilities or a Copilot extension to an add-in that uses the add-in only manifest, or to just future proof the add-in, you need to convert it to use the unified manifest for Microsoft 365.

> **Note**
>
> The unified manifest for Microsoft 365 can be used in production Outlook add-ins. It's available only as a preview for Excel, PowerPoint, and Word add-ins.

There are three basic tasks to converting an add-in project from the add-in only manifest to the unified manifest.

- Ensure that your manifest is ready to convert.
- Convert the XML-formatted add-in only manifest itself to the JSON format of the unified manifest.
- Package the new manifest and two icon image files (described later) into a zip file for sideloading or deployment. *Depending on how you sideload the converted add-in, this task may be done for you automatically.*

> **Note**
>
> Office Add-ins that use the unified manifest for Microsoft 365 are *directly* supported in Office on the web, in new Outlook on Windows, and in Office on Windows connected to a Microsoft 365 subscription, Version 2304 (Build 16320.00000) or later.
>
> When the app package that contains the unified manifest is deployed in AppSource or the Microsoft 365 Admin Center then an add-in only manifest is generated from the unified manifest and stored. This add-in only manifest enables the add-in to be installed on platforms that don't directly support the unified manifest, including Office on Mac, Office on mobile, subscription versions of Office on Windows earlier than 2304 (Build 16320.00000), and perpetual versions of Office on Windows.

> **Note**
>
> - Add-ins that use the unified manifest can be sideloaded only on Office Version 2304 (Build 16320.20000) or later.

- Projects created in Visual Studio, as distinct from Visual Studio Code, can't be converted at this time.
- If you created the project with Teams Toolkit or Microsoft 365 Agents Toolkit or with the "unified manifest" option in the Office Yeoman Generator, it already uses the unified manifest.

Ensure that your manifest is ready to convert

The following sections describe conditions that must be met before you convert the manifest.

Uninstall the existing version of the add-in

To avoid conflicts with UI control names and other problems, be sure the existing add-in isn't installed on the computer where you do the conversion. If you experience any difficulties uninstalling the add-in, see Remove a ghost add-in.

Ensure that you have two special image files

If your add-in only manifest doesn't already have both **<IconUrl>** and **<HighResolutionIconUrl>** (in that order) elements, then add them just below the **<Description>** element. The values of the **DefaultValue** attribute should be the full URLs of image files. The images must be a specified size as shown in the following table.

Expand table

Office application	<IconUrl>	<HighResolutionIconUrl>
Outlook	64x64 pixels	128x128 pixels
All other Office applications	32x32 pixels	64x64 pixels

The following markup is an example.

XML

```xml
<OfficeApp xmlns="http://schemas.microsoft.com/office/appforoffice/1.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="MailApp">
  <Id>01234567-89ab-cdef-0123-4567-89abcdef0123</Id>
  <Version>1.0</Version>
  <ProviderName>Contoso</ProviderName>
  <DefaultLocale>en-us</DefaultLocale>
  <DisplayName DefaultValue="Great Add-in"/>
```

```
    <Description DefaultValue="A great add-in."/>
    <IconUrl DefaultValue="https://localhost:3000/assets/icon-64.png" />
    <HighResolutionIconUrl DefaultValue="https://localhost:300/assets/icon-128.png"
  />

    <!-- Other markup omitted -->
```

Reduce the number of add-in commands as needed

An add-in that uses the unified manifest may not have more than 20 add-in commands. If the total number of **<Action>** elements in the add-in only manifest is greater than 20, you must redesign the add-in to have no more than 20.

Update the add-in ID, version, domain, and function names in the manifest

1. Change the value of the `<ID>` element to a new random GUID.
2. Update the value of the `<Version>` element and ensure that it conforms to the semver standard (MAJOR.MINOR.PATCH). Each segment can have no more than five digits. For example, change the value `1.0.0.0` to `1.0.1`. The semver standard's prerelease and metadata version string extensions aren't supported.
3. Be sure that the domain segment of the add-in's URLs in the manifest are pointing to `https://localhost:3000`.
4. If your manifest has any **<FunctionName>** elements, make sure their values have fewer than 65 characters.

   > **Important**
   >
   > The value of this element must exactly match the name of an action that's mapped to a function in a JavaScript or TypeScript file with the Office.actions.associate function. If you change it in the manifest, be sure to change it in the `actionId` parameter passed to `associate()` too.

Shorten string values as needed

Review and change, as needed, manifest values in light of the following effects of the conversion.

- The first 30 characters of `<DisplayName>` becomes the value of "name.short" in the unified manifest.
- The first 100 characters of `<DisplayName>` becomes the value of "name.full" in the unified manifest.
- The first 250 characters of `<Description>` becomes the value of "description.short" in the unified manifest.
- The first 4000 characters of `<Description>` becomes the value of "description.full" in the unified manifest.
- The first 32 characters of the `<ProviderName>` becomes the value of "developer.name" in the unified manifest.

Verify that the modified add-in only manifest works

1. Validate the modified add-in only manifest. See Validate an Office Add-in's manifest.
2. Verify that the add-in can be sideloaded and run. See Sideload an Office Add-in for testing.

Resolve any problems before you attempt to convert the project.

Conversion tools and options

There are several ways to carry out the remaining tasks, depending on the IDE and other tools you want to use for your project, and on the tool you used to create the project.

- Convert projects created with the Yeoman generator for Office Add-ins (aka "Yo Office")
- Convert NodeJS and npm projects that weren't created with the Yeoman generator for Office Add-ins (Yo Office)

> **Note**
>
> Conversion of the manifest is one of the effects of importing the add-in project into Agents Toolkit if you do so using the toolkit's importation feature. For details, see Import an add-in project to Agents Toolkit

Convert projects created with the Yeoman generator for Office Add-ins (aka "Yo Office")

If the project was created with the Yeoman generator for Office Add-ins, convert it using the following steps.

1. In the root of the project, open a command prompt or bash shell and run the following command. This converts the manifest and updates the package.json to specify current tooling packages. The new unified manifest is in the root of the project and the old add-in only manifest is in a backup.zip file. For details about this command, see Office-Addin-Project.

   command line

   ```
   npx office-addin-project convert -m <relative-path-to-XML-manifest>
   ```

2. Run `npm install`.

3. To sideload the add-in, see Sideload add-ins created with the Yeoman generator for Office Add-ins (Yo Office).

Convert NodeJS and npm projects that weren't created with the Yeoman generator for Office Add-ins (Yo Office)

If your project wasn't created with Yo Office, use the office-addin-manifest-converter tool.

In the root of the project, open a command prompt or bash shell and run the following command. This command puts the unified manifest in a subfolder with the same name as the filename stem of the original add-in only manifest. For example, if the manifest is named **MyManifest.xml**, the unified manifest is created at **.\MyManifest\MyManifest.json**. For more details about this command, see Office-Addin-Manifest-Converter.

command line

```
npx office-addin-manifest-converter convert <relative-path-to-XML-manifest>
```

Once you have the unified manifest created, there are two ways to create the zip file and sideload it. For more information, see Sideload other NodeJS and npm projects.

> **Note**
>
> If the original add-in only manifest used any **<Override>** elements to localize strings in the manifest, then the conversion process produces JSON string files for each localized language. These files must also be included in the zip file, and they must be at the relative path indicated in the "localizationInfo.additionalLanguages.file" property.

Next steps

Consider whether to maintain both the old and new versions of the add-in. See Manage both a unified manifest and an add-in only manifest version of your Office Add-in.

Office Add-ins with the add-in only manifest

07/11/2025

This article introduces the XML-formatted add-in only manifest for Office Add-ins. It assumes that you're familiar with the Office Add-ins manifest.

> **Tip**
>
> For an overview of the unified manifest for Microsoft 365, see **Office Add-ins with the unified manifest for Microsoft 365**.

Schema versions

Not all Office clients support the latest features, and some Office users will have an older version of Office. Having schema versions lets developers build add-ins that are backwards compatible, using the newest features where they are available but still functioning on older versions.

The <**VersionOverrides**> element in the manifest is an example of this. All elements defined inside <**VersionOverrides**> will override the same element in the other part of the manifest. This means that, whenever possible, Office will use what is in the <**VersionOverrides**> section to set up the add-in. However, if the version of Office doesn't support a certain version of <**VersionOverrides**>, Office will ignore it and depend on the information in the rest of the manifest.

This approach means that developers don't have to create multiple individual manifests, but rather keep everything defined in one file.

The current versions of the schema are:

Expand table

Version	Description
v1.0	Supports version 1.0 of the Office JavaScript API. For example, in Outlook add-ins, this supports the read form.
v1.1	Supports version 1.1 of the Office JavaScript API and <**VersionOverrides**>. For example, in Outlook add-ins, this adds support for the compose form.

Version	Description
<**VersionOverrides**> 1.0	Supports later versions of the Office JavaScript API. This supports add-in commands.
<**VersionOverrides**> 1.1	Supported by Outlook only. This version of <**VersionOverrides**> adds support for newer features, such as pinnable task panes and mobile add-ins.

Even if your add-in manifest uses the <**VersionOverrides**> element, it is still important to include the v1.1 manifest elements to allow your add-in to work with older clients that do not support <**VersionOverrides**>.

> **Note**
>
> Office uses a schema to validate manifests. The schema requires that elements in the manifest appear in a specific order. If you include elements out of the required order, you may get errors when sideloading your add-in. See How to find the proper order of manifest elements elements in the required order.

Required elements

The following table specifies the elements that are required for the three types of Office Add-ins.

> **Note**
>
> There is also a mandatory order in which elements must appear within their parent element. For more information see How to find the proper order of add-in only manifest elements.

Required elements by Office Add-in type

Expand table

Element	Content	Task pane	Mail (Outlook)
OfficeApp	Required	Required	Required
Id	Required	Required	Required
Version	Required	Required	Required

Element	Content	Task pane	Mail (Outlook)
ProviderName	Required	Required	Required
DefaultLocale	Required	Required	Required
DisplayName	Required	Required	Required
Description	Required	Required	Required
SupportUrl**	Required	Required	Required
DefaultSettings (ContentApp) DefaultSettings (TaskPaneApp)	Required	Required	Not available
SourceLocation (ContentApp) SourceLocation (TaskPaneApp) SourceLocation (MailApp)	Required	Required	Required
DesktopSettings	Not available	Not available	Required
Permissions (ContentApp) Permissions (TaskPaneApp) Permissions (MailApp)	Required	Required	Required
Rule (RuleCollection) Rule (MailApp)	Not available	Not available	Required
Requirements (MailApp)*	Not applicable	Not available	Required
Set* Sets (Requirements)* Sets (MailAppRequirements)*	Required	Required	Required
Form* FormSettings*	Not available	Not available	Required
Hosts*	Required	Required	Optional

**Added in the Office Add-in Manifest Schema version 1.1.*

*** SupportUrl is only required for add-ins that are distributed through AppSource.*

Root element

The root element for the Office Add-in manifest is **<OfficeApp>**. This element also declares the default namespace, schema version and the type of add-in. Place all other elements in the manifest within its open and close tags. The following is an example of the root element.

XML

```xml
<OfficeApp
  xmlns="http://schemas.microsoft.com/office/appforoffice/1.1"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xmlns:bt="http://schemas.microsoft.com/office/officeappbasictypes/1.0"

xmlns:mailappor="http://schemas.microsoft.com/office/mailappversionoverrides/1.0"
  xsi:type="MailApp">

  <!-- The rest of the manifest. -->

</OfficeApp>
```

Version

This is the version of the specific add-in. If a developer updates something in the manifest, the version must be incremented as well. This way, when the new manifest is installed, it will overwrite the existing one and the user will get the new functionality. If this add-in was submitted to the store, the new manifest will have to be re-submitted and re-validated. Then, users of this add-in will get the new updated manifest automatically in a few hours, after it is approved.

If the add-in's requested permissions change, users will be prompted to upgrade and re-consent to the add-in. If the admin installed this add-in for the entire organization, the admin will have to re-consent first. Users will be unable to use the add-in until consent is granted.

Hosts

Office add-ins specify the <**Hosts**> element like the following:

XML

```xml
<OfficeApp>
...
  <Hosts>
    <Host Name="Mailbox" />
  </Hosts>
...
</OfficeApp>
```

This is separate from the <**Hosts**> element inside the <**VersionOverrides**> element, which is discussed in Create add-in commands with the add-in only manifest.

Specify safe domains with the AppDomains element

There is an AppDomains element of the add-in only manifest file that is used to tell Office which domains your add-in should be allowed to navigate to. As noted in Specify domains you want to open in the add-in window, when running in Office on the web, your task pane can be navigated to any URL. However, in desktop platforms, 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 element), that URL opens in a new browser window outside the add-in pane of the Office application.

To override this (desktop Office) behavior, add each domain you want to open in the add-in window in the list of domains specified in the <**AppDomains**> element. If the add-in tries to go to a URL in a domain that is in the list, then it opens in the task pane in both Office on the web and desktop. If it tries to go to a URL that isn't in the list, then in desktop Office that URL opens in a new browser window (outside the add-in pane).

The following table describes browser behavior when your add-in attempts to navigate to a URL outside of the add-in's default domain.

Expand table

Office client	Domain defined in AppDomains?	Browser behavior
All clients	Yes	Link opens in add-in task pane.
Office 2016 on Windows (volume-licensed perpetual)	No	Link opens in Internet Explorer 11.
Other clients	No	Link opens in user's default browser.

The following add-in only 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 element within the <**AppDomains**> element list. If the add-in goes to a page in the `www.northwindtraders.com` domain, that page opens in the add-in pane, even in Office desktop.

XML

```
<?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">
  <!--IMPORTANT! Id must be unique for each add-in. If you copy this manifest
ensure that you change this id to your own GUID. -->
  <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"/>
  <SupportUrl DefaultValue="[Insert the URL of a page that provides support
information for the app]" />
  <AppDomains>
    <AppDomain>https://www.northwindtraders.com</AppDomain>
  </AppDomains>
  <DefaultSettings>
    <SourceLocation DefaultValue="https://www.contoso.com/search_app/Default.aspx"
/>
  </DefaultSettings>
  <Permissions>ReadWriteDocument</Permissions>
</OfficeApp>
```

Version overrides in the manifest

The optional VersionOverrides element contains child markup that enables additional add-in features. Some of these are:

- Customizing the Office ribbon and menus.
- Customizing how Office works with the embedded runtimes in which add-ins run.
- Configuring how the add-in interacts with Azure Active Directory and Microsoft Graph for Single Sign-on.

Some descendant elements of `VersionOverrides` have values that override values of the parent `OfficeApp` element. For example, the `Hosts` element in `VersionOverrides` overrides the `Hosts` element in `OfficeApp`.

The `VersionOverrides` element has its own schema, actually four of them, depending on the type of add-in and the features it uses. The schemas are:

- Task pane 1.0
- Content 1.0
- Mail 1.0
- Mail 1.1

When a `VersionOverrides` element is used, then the `OfficeApp` element must have a `xmlns` attribute that identifies the appropriate schema. The possible values of the attribute are the following:

- `http://schemas.microsoft.com/office/taskpaneappversionoverrides`
- `http://schemas.microsoft.com/office/contentappversionoverrides`
- `http://schemas.microsoft.com/office/mailappversionoverrides`

The `VersionOverrides` element itself must also have an `xmlns` attribute specifying the schema. The possible values are the three above and the following:

- `http://schemas.microsoft.com/office/mailappversionoverrides/1.1`

The `VersionOverrides` element also must have an `xsi:type` attribute that specifies the schema version. The possible values are the following:

- `VersionOverridesV1_0`
- `VersionOverridesV1_1`

The following are examples of `VersionOverrides` used, respectively, in a task pane add-in and a mail add-in. Note that when a mail `VersionOverrides` with version 1.1 is used, it must be the last child of a parent `VersionOverrides` of type 1.0. The values of child elements in the inner `VersionOverrides` override the values of the same-named elements in the parent `VersionOverrides` and the grandparent `OfficeApp` element.

XML

```xml
<VersionOverrides
xmlns="http://schemas.microsoft.com/office/taskpaneappversionoverrides"
xsi:type="VersionOverridesV1_0">
    <!-- Child elements are omitted. -->
</VersionOverrides>
```

XML

```xml
<VersionOverrides
xmlns="http://schemas.microsoft.com/office/mailappversionoverrides"
xsi:type="VersionOverridesV1_0">
  <!-- Other child elements are omitted. -->
  <VersionOverrides
xmlns="http://schemas.microsoft.com/office/mailappversionoverrides/1.1"
xsi:type="VersionOverridesV1_1">
    <!-- Child elements are omitted. -->
  </VersionOverrides>
</VersionOverrides>
```

For an example of a manifest that includes a `VersionOverrides` element, see Manifest v1.1 XML file examples and schemas.

Requirements

The <**Requirements**> element specifies the set of APIs available to the add-in. For detailed information about requirement sets, see Office requirement sets availability. For example, in an Outlook add-in, the requirement set must be Mailbox and a value of 1.1 or above.

The <**Requirements**> element can also appear in the <**VersionOverrides**> element, allowing the add-in to specify a different requirement when loaded in clients that support <**VersionOverrides**>.

The following example uses the **DefaultMinVersion** attribute of the <**Sets**> element to require office.js version 1.1 or higher, and the **MinVersion** attribute of the <**Set**> element to require the Mailbox requirement set version 1.1.

XML

```xml
<OfficeApp>
...
  <Requirements>
    <Sets DefaultMinVersion="1.1">
      <Set Name="MailBox" MinVersion="1.1" />
    </Sets>
  </Requirements>
...
</OfficeApp>
```

Localization

Some aspects of the add-in need to be localized for different locales, such as the name, description and the URL that's loaded. These elements can easily be localized by specifying the default value and then locale overrides in the <**Resources**> element within the <**VersionOverrides**> element. The following shows how to override an image, a URL, and a string.

XML

```xml
<Resources>
  <bt:Images>
    <bt:Image id="icon1_16x16"
DefaultValue="https://contoso.com/images/app_icon_small.png" >
      <bt:Override Locale="ar-sa"
Value="https://contoso.com/images/app_icon_small_arsa.png" />
      <!-- Add information for other locales. -->
    </bt:Image>
  </bt:Images>
```

```
  <bt:Urls>
    <bt:Url id="residDesktopFuncUrl"
DefaultValue="https://contoso.com/urls/page_appcmdcode.html" >
      <bt:Override Locale="ar-sa"
Value="https://contoso.com/urls/page_appcmdcode.html?lcid=ar-sa" />
      <!-- Add information for other locales. -->
    </bt:Url>
  </bt:Urls>

  <bt:ShortStrings>
    <bt:String id="residViewTemplates" DefaultValue="Launch My Add-in">
      <bt:Override Locale="ar-sa" Value="<add localized value here>" />
      <!-- Add information for other locales. -->
    </bt:String>
  </bt:ShortStrings>
</Resources>
```

The schema reference contains full information on which elements can be localized.

Manifest v1.1 XML file examples and schemas

The following sections show examples of manifest v1.1 XML files for content, task pane, and mail (Outlook) add-ins.

Task pane

Add-in manifest schemas

XML

```
<?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 copy 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-in's 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-in's
dialog. -->
  <IconUrl DefaultValue="https://contoso.com/assets/icon-32.png" />
  <HighResolutionIconUrl DefaultValue="https://contoso.com/assets/hi-res-
icon.png" />
  <SupportUrl DefaultValue="[Insert the URL of a page that provides support
information for the app]" />
  <!-- Domains that are allowed when navigating. For example, if you use
ShowTaskpane and then have an href link, navigation is only allowed if the
domain is on this list. -->
  <AppDomains>
    <AppDomain>AppDomain1</AppDomain>
    <AppDomain>AppDomain2</AppDomain>
  </AppDomains>
  <!-- End Basic Settings. -->

  <!-- BeginTaskPaneMode integration. 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. DesktopFormFactor is supported. Other form factors
are available depending on the host and feature. -->
        <DesktopFormFactor>
          <!-- This code enables a customizable message to be displayed when
the add-in is loaded successfully upon individual install. -->
          <GetStarted>
            <!-- Title of the Getting Started callout. The resid attribute
points to a ShortString resource. -->
            <Title resid="Contoso.GetStarted.Title"/>
            <!-- Description of the Getting Started callout. resid points to a
LongString resource. -->
```

```
            <Description resid="Contoso.GetStarted.Description"/>
            <!-- Points to a URL resource which details how the add-in should
be used. -->
            <LearnMoreUrl resid="Contoso.GetStarted.LearnMoreUrl"/>
          </GetStarted>
          <!-- Function file is an HTML page that includes, or loads, 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 app 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 ID-->
              <Group id="Contoso.Tab1.Group1">
                <!--Label for your group. resid must point to a ShortString
resource. -->
                <Label resid="Contoso.Tab1.GroupLabel" />
                <Icon>
                  <!-- Each size needs its own icon resource or it will look
distorted when resized. -->
                  <!-- Icons. Required sizes: 16, 32, 80; optional: 20, 24,
40, 48, 64. You should provide as many sizes as possible for a great user
experience. -->
                  <!-- Use PNG icons and remember that all URLs on the
resources section must use HTTPS. -->
                  <bt:Image size="16" resid="Contoso.TaskpaneButton.Icon16" />
                  <bt:Image size="32" resid="Contoso.TaskpaneButton.Icon32" />
                  <bt:Image size="80" resid="Contoso.TaskpaneButton.Icon80" />
                </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.Icon16"
/>
                    <bt:Image size="32" resid="Contoso.FunctionButton.Icon32"
/>
                    <bt:Image size="80" resid="Contoso.FunctionButton.Icon80"
/>
```

```
                  </Icon>
                  <!-- This is what happens when the command is triggered
(e.g., click on the ribbon button). -->
                  <!-- Supported actions are ExecuteFunction 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.Icon16"
/>
                    <bt:Image size="32" resid="Contoso.TaskpaneButton.Icon32"
/>
                    <bt:Image size="80" resid="Contoso.TaskpaneButton.Icon80"
/>
                  </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.Icon16"
/>
                    <bt:Image size="32" resid="Contoso.TaskpaneButton.Icon32"
/>
                    <bt:Image size="80" resid="Contoso.TaskpaneButton.Icon80"
/>
                  </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.Icon16" />
                        <bt:Image size="32"
resid="Contoso.TaskpaneButton.Icon32" />
                        <bt:Image size="80"
resid="Contoso.TaskpaneButton.Icon80" />
                      </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.Icon16" />
                        <bt:Image size="32"
resid="Contoso.TaskpaneButton.Icon32" />
                        <bt:Image size="80"
resid="Contoso.TaskpaneButton.Icon80" />
                      </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. -->
              <Label resid="Contoso.Tab1.TabLabel" />
            </CustomTab>
          </ExtensionPoint>
        </DesktopFormFactor>
      </Host>
    </Hosts>

    <!-- You can use resources across hosts and form factors. -->
    <Resources>
      <bt:Images>
        <bt:Image id="Contoso.TaskpaneButton.Icon16"
DefaultValue="https://myCDN/Images/Button16x16.png" />
```

```
        <bt:Image id="Contoso.TaskpaneButton.Icon32"
DefaultValue="https://myCDN/Images/Button32x32.png" />
        <bt:Image id="Contoso.TaskpaneButton.Icon80"
DefaultValue="https://myCDN/Images/Button80x80.png" />
        <bt:Image id="Contoso.FunctionButton.Icon"
DefaultValue="https://myCDN/Images/ButtonFunction.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>
      <!-- ShortStrings max characters=125. -->
      <bt:ShortStrings>
        <bt:String id="Contoso.FunctionButton.Label" DefaultValue="Execute
Function" />
        <bt:String id="Contoso.TaskpaneButton.Label" DefaultValue="Show Task
Pane" />
        <bt:String id="Contoso.Dropdown.Label" DefaultValue="Dropdown" />
        <bt:String id="Contoso.Item1.Label" DefaultValue="Show Task Pane 1" />
        <bt:String id="Contoso.Item2.Label" DefaultValue="Show Task Pane 2" />
        <bt:String id="Contoso.Tab1.GroupLabel" DefaultValue="Test Group" />
         <bt:String id="Contoso.Tab1.TabLabel" DefaultValue="Test Tab" />
      </bt:ShortStrings>
      <!-- LongStrings max characters=250. -->
      <bt:LongStrings>
        <bt:String id="Contoso.FunctionButton.Tooltip" DefaultValue="Click to
execute function." />
        <bt:String id="Contoso.TaskpaneButton.Tooltip" DefaultValue="Click to
show a task pane." />
        <bt:String id="Contoso.Dropdown.Tooltip" DefaultValue="Click to show
options on this menu." />
        <bt:String id="Contoso.Item1.Tooltip" DefaultValue="Click to show Task
Pane 1." />
        <bt:String id="Contoso.Item2.Tooltip" DefaultValue="Click to show Task
Pane 2." />
      </bt:LongStrings>
    </Resources>
  </VersionOverrides>
  <!-- EndAddinCommandsMode integration. -->
</OfficeApp>
```

Validate an Office Add-in's manifest

For information about validating a manifest against the XML Schema Definition (XSD), see Validate an Office Add-in's manifest.

See also

- How to find the proper order of add-in only manifest elements
- Create add-in commands with the add-in only manifest
- Specify Office applications and API requirements
- Localization for Office Add-ins
- Schema reference for XML Office Add-ins manifests
- Make your Office Add-in compatible with an existing COM or VSTO add-in
- Requesting permissions for API use in add-ins
- Validate an Office Add-in's manifest