Navigation patterns

Article • 05/18/2023

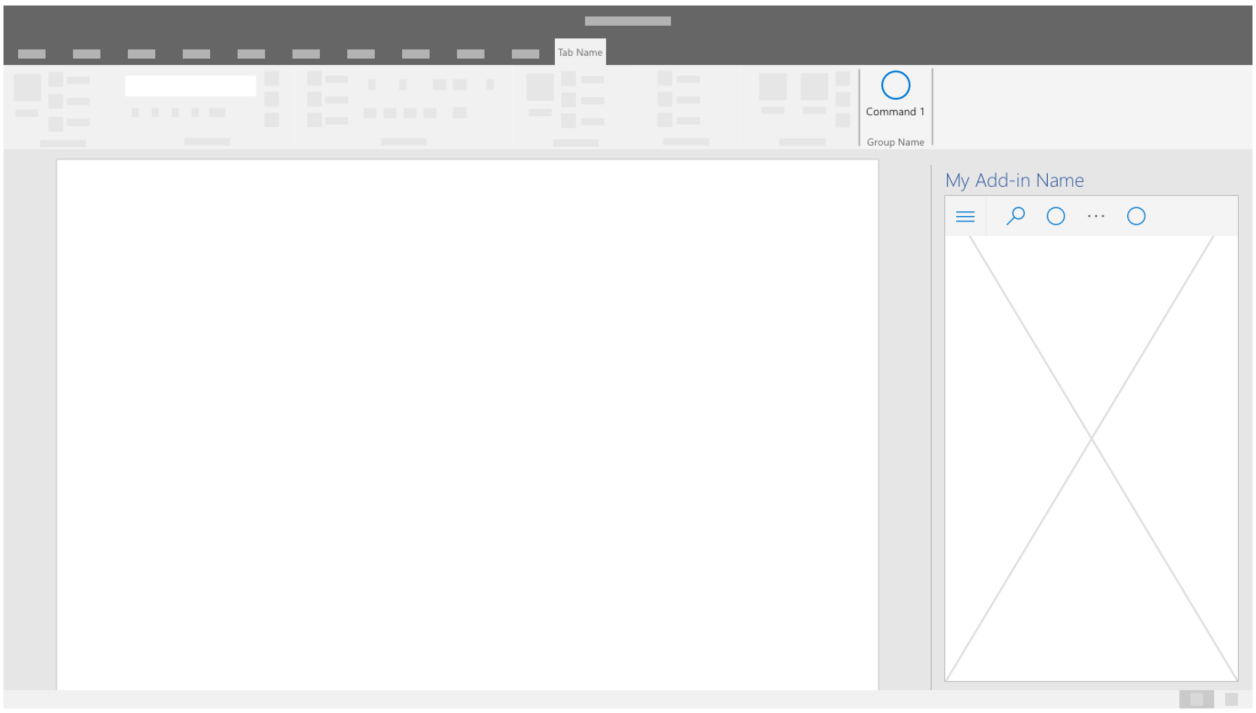
The main features of an add-in are accessed through specific command types and limited screen area. It's important that navigation is intuitive, provides context, and allows the user to move easily throughout the add-in.

Best practices

Do	Don't
Ensure the user has a clearly visible navigation option.	Don't complicate the navigation process by using non-standard UI.
Utilize the following components as applicable to allow users to navigate through your add-in.	Don't make it difficult for the user to understand their current place or context within the add-in

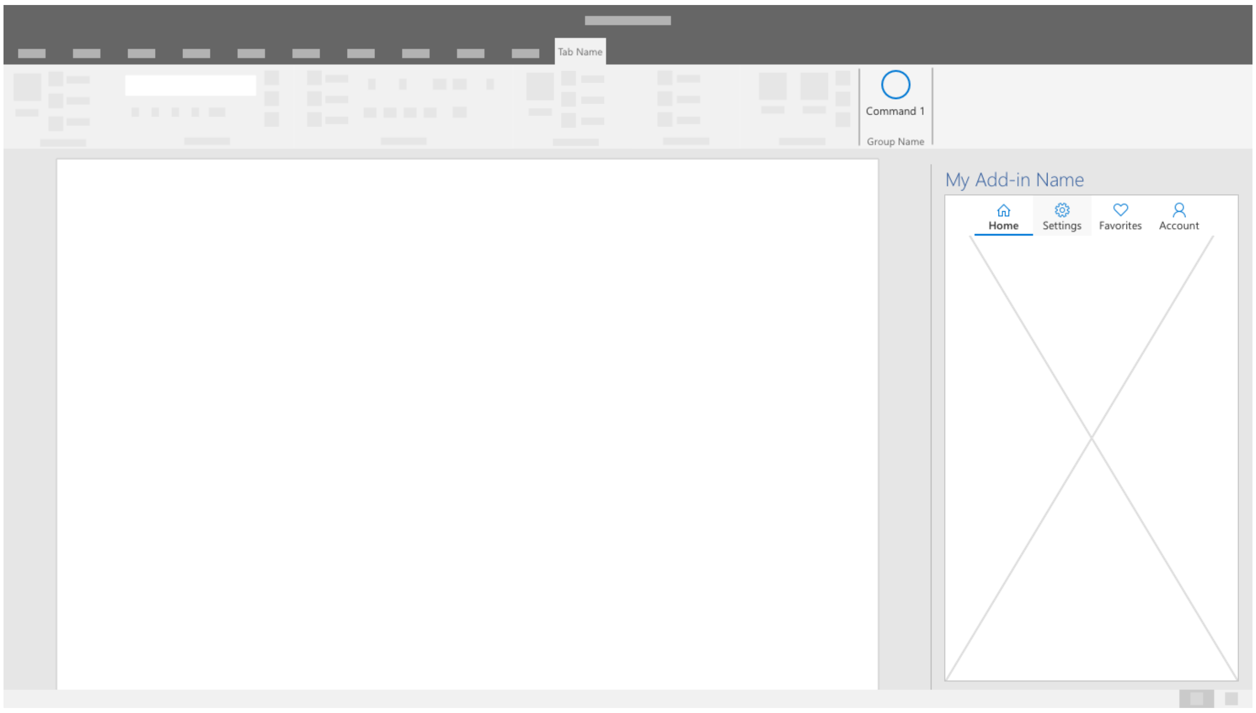
Command Bar

The CommandBar is a surface within the task pane that houses commands that operate on the content of the window, panel, or parent region it resides above. Optional features include a hamburger menu access point, search, and side commands.



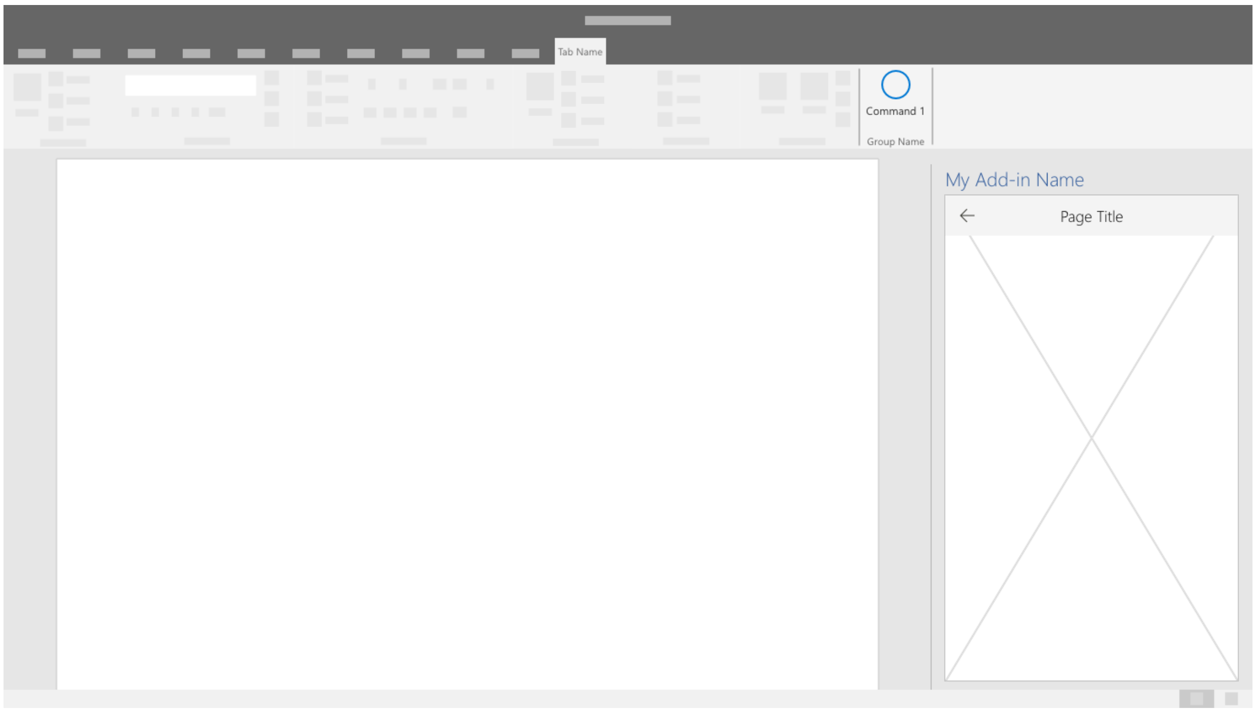
Tab Bar

The tab bar shows navigation using buttons with vertically stacked text and icons. Use the tab bar to provide navigation using tabs with short and descriptive titles.



Back Button

The back button allows users to recover from a drill-down navigational action. This pattern helps ensure users follow an ordered series of steps.

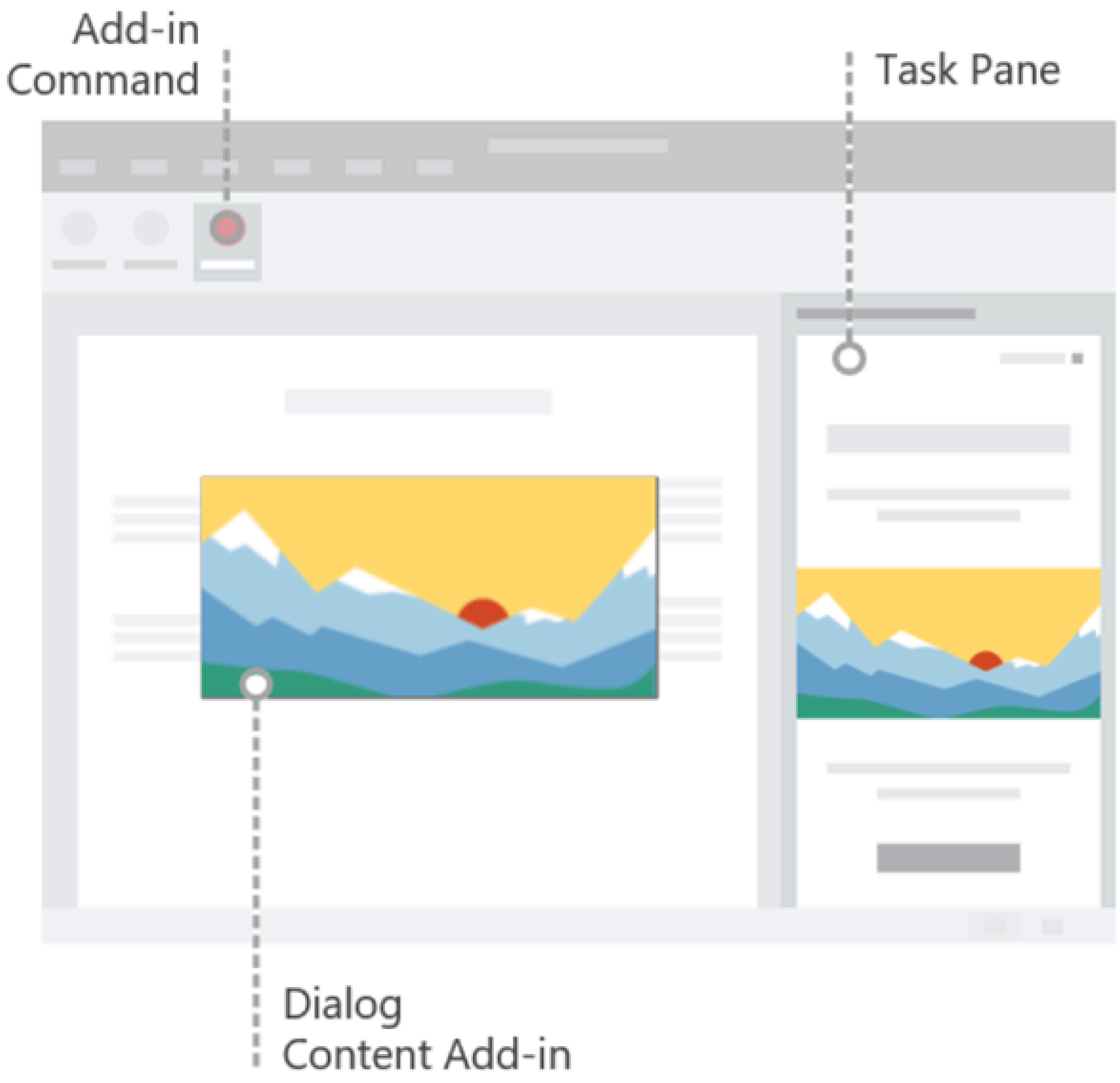


Office UI elements for Office Add-ins

Article • 03/12/2025

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

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



Add-in commands

Use add-in commands to add entry points to your add-in to the Office app ribbon. Commands start actions in your add-in either by running JavaScript code, or by launching an HTML container. You can create two types of add-in commands.

Expand table

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

HTML containers

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

Expand table

HTML container	Description
Task panes	Display custom UI in the right pane of the Office document. Use task panes to allow users to interact with your add-in side-by-side with the Office document.
Content add-ins	Display custom UI embedded within Office documents. Use content add-ins to allow users to interact with your add-in directly within the Office document. For example, you might want to show external content such as videos or data visualizations from other sources.
Dialog boxes	Display custom UI in a dialog box that overlays the Office document. Use a dialog box for interactions that require focus and more real estate, and don't require a side-by-side interaction with the document.

See also

- Add-in commands for Excel, Word, and PowerPoint
- Task panes
- Content add-ins
- Dialog boxes

Add-in commands

08/13/2025

Add-in commands are UI elements that extend the Office UI and start actions in your add-in. You can use add-in commands to add a button on the ribbon or an item to a context menu. When users select an add-in command, they initiate actions such as running JavaScript code, or showing a page of the add-in in a task pane. Add-in commands help users find and use your add-in, which can help increase your add-in's adoption and reuse, and improve customer retention.

> **Note**
>
> - SharePoint catalogs don't support add-in commands. You can deploy add-in commands via the integrated apps portal or AppSource, or use sideloading to deploy your add-in command for testing.
> - Content add-ins don't currently support add-in commands.

Types of add-in commands

There are two types of add-in commands, based on the kind of action that the command triggers.

- **Task pane commands**: The button or menu item opens the add-in's task pane. You add this kind of add-in command with markup in the manifest. The "code behind" the command is provided by Office.

- **Function commands**: The button or menu item runs any arbitrary JavaScript. The code almost always calls APIs in the Office JavaScript Library, but it doesn't have to. This type of add-in typically displays no UI other than the button or menu item itself. Note the following about function commands:

  - The runtime in which the function command runs is a full browser-based runtime. It can render HTML and call out to the Internet to send or get data.

  - The runtime closes when either the function completes or 5 minutes have passed, whichever is earlier.

  - The function that is triggered can call the displayDialogAsync method to show a dialog. This is a good way to display an error, show progress, or prompt the user for input.

> **Note**
>
> Because of the 5-minute timeout, the dialog should be designed so that users complete their interaction and close it within 5 minutes. Your add-in should use a task pane for longer interactions.

- If the add-in is configured to use a shared runtime, the function can also call the showAsTaskpane method.

> **Tip**
>
> Function commands aren't the only way to run arbitrary JavaScript in an add-in. An add-in can also include:
> - Custom handlers for certain events, such as a user opening an new message pane in Outlook.
> - Custom Copilot agents that take actions in response to natural language requests from the add-in's users.

Location of add-in commands

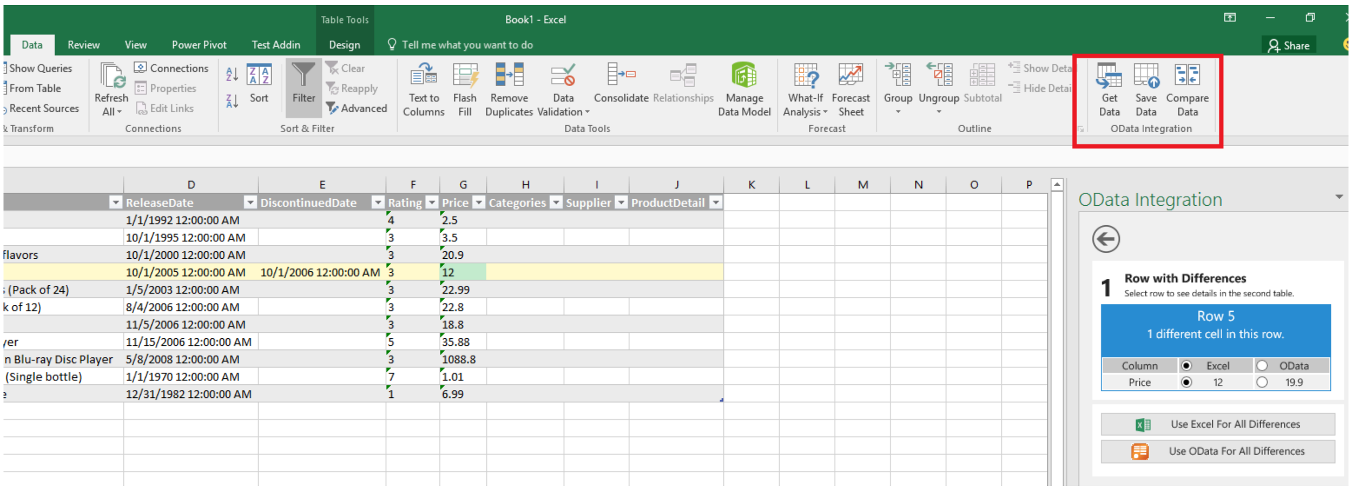
When a user installs an add-in, the add-in's commands are found on the ribbon, in a built-in Office tab or a custom tab that is specified in the manifest. (You can also put add-in commands on a custom contextual tab that your add-in code defines at runtime.) They appear in the UI as a button or an item in a dropdown menu.

As the ribbon or action bar gets more crowded, add-in commands are displayed in the overflow menu. Commands for the same add-in are usually grouped together.

In Office on the web, if you're using the single-line or simplified ribbon layout, the add-in name isn't shown on the ribbon. Only the add-in's command icon is shown.

Excel, PowerPoint, and Word

The following shows an example of add-in commands in a custom group on the **Data** tab of the Excel ribbon.



	D	E	F	G	H	I	J
	ReleaseDate	DiscontinuedDate	Rating	Price	Categories	Supplier	ProductDetail
	1/1/1992 12:00:00 AM		4	2.5			
	10/1/1995 12:00:00 AM		3	3.5			
flavors	10/1/2000 12:00:00 AM		3	20.9			
	10/1/2005 12:00:00 AM	10/1/2006 12:00:00 AM	3	12			
(Pack of 24)	1/5/2003 12:00:00 AM		3	22.99			
k of 12)	8/4/2006 12:00:00 AM		3	22.8			
	11/5/2006 12:00:00 AM		3	18.8			
yer	11/15/2006 12:00:00 AM		5	35.88			
n Blu-ray Disc Player	5/8/2008 12:00:00 AM		3	1088.8			
(Single bottle)	1/1/1970 12:00:00 AM		7	1.01			
e	12/31/1982 12:00:00 AM		1	6.99			

OData Integration

1 **Row with Differences**
Select row to see details in the second table.

Row 5
1 different cell in this row.

Column	Excel	OData
Price	● 12	○ 19.9

Use Excel For All Differences

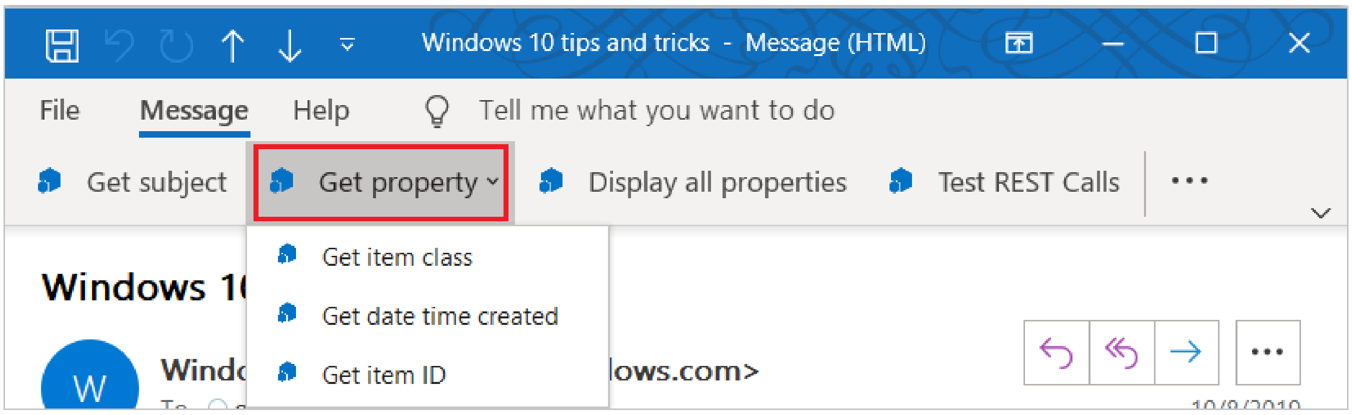
Use OData For All Differences

Outlook

For Outlook, when you want an add-in command on a built-in ribbon tab, rather than creating your own, the command will appear on the default tab based on the platform and current Outlook mode. For guidance, see Use add-ins in Outlook.

Dropdown menu

A dropdown menu add-in command defines a static list of items. The menu can be any mix of items that execute a function or that open a task pane. Submenus aren't supported.



Grouped add-in commands on the ribbon

Multiple add-in commands can be grouped together on the ribbon. A group must contain at least one add-in command in the form of a button or a dropdown menu. In Office on Windows and on Mac, the label and icon of a button or dropdown menu are usually shown for add-in commands in a group. However, the icon size and label visibility may vary due to the following factors that constrain space.

- The number of add-in commands in the group.
- The size of the Office client window.

If the client window is maximized and there are more than three controls in a group, the label of each control is shown, but the size of its icon may vary (some are shown as 16 x 16 pixels while others are shown as 32 x 32 pixels).

When there are two or more add-in commands in a group and space becomes limited, the following adjustments are made to how the add-in commands are displayed. These changes are applied to the groups of add-in commands from right to left across the ribbon in the following sequence.

1. Small icons (16 x 16 pixels) and labels are shown for each add-in command in a group.
2. Only small icons are shown.
3. The group is displayed as a dropdown menu instead of showing individual add-in commands on the ribbon. A scroll slider icon also appears on the ribbon, so that you can scroll through the ribbon.

In Office on the web, the icon size and label visibility of controls in groups don't change as the browser window is resized. The scroll slider icon is simply shown on the ribbon.

Command capabilities

The following command capabilities are currently supported.

Extension points

- Ribbon tabs - Extend built-in tabs or create a new custom core tab. An add-in can have just one custom core tab. (You can also put add-in commands on a custom contextual tab.)

  > **Note**
  >
  > For Outlook, custom tabs are only supported in classic Outlook on Windows. In Outlook on the web, on Mac, and in the new Outlook on Windows, you can put custom groups of controls on one of the built-in ribbon tabs instead.

- Context menus - Extend selected context menus.

Control types

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

Default availability state

You can specify whether the command is available when your add-in launches, and programmatically change the setting.

> **Note**
>
> This feature isn't supported in all Office applications or scenarios. For more information, see Change the availability of add-in commands.

Position on the ribbon

You can specify where a custom tab appears on the Office application's ribbon, such as "just to the right of the Home tab".

> **Note**
>
> This feature isn't supported in all Office applications or scenarios. For more information, see Position a custom tab on the ribbon.

Integration of built-in Office buttons

You can insert the built-in Office ribbon buttons into your custom command groups and your custom ribbon tab.

> **Note**
>
> This feature isn't supported in all Office applications or scenarios. For more information, see Integrate built-in Office buttons into custom tabs.

Contextual tabs

You can specify a custom contextual tab; that is, a tab that is only visible on the ribbon in certain contexts, such as when a chart is selected in Excel.

> **Note**

This feature isn't supported in all Office applications or scenarios. For more information, see Create custom contextual tabs in Office Add-ins.

Supported platforms

Add-in commands are currently supported on the following platforms, except for limitations specified in the subsections of Command capabilities earlier.

- Office on the web
- Office on Windows (Version 1604 (Build 6769.2000) or later, connected to a Microsoft 365 subscription)
- Office on Mac (Version 15.33 (17040900) or later, connected to a Microsoft 365 subscription)
- Perpetual Office 2019 or later on Windows or on Mac

ⓘ **Note**

For information about support in Outlook, see Outlook support notes.

Debug

To debug an add-in command, you must run it in Office on the web. For details, see Debug add-ins in Office on the web.

Best practices

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

- Use commands to represent a specific action with a clear and specific outcome for users. Don't combine multiple actions in a single button.

- Provide granular actions that make common tasks within your add-in more efficient to perform. Minimize the number of steps an action takes to complete.

- For the placement of your commands in the Office app ribbon:
  - Place commands on an existing tab (Insert, Review, and so on) if the functionality provided fits there. For example, if your add-in enables users to insert media, add a group to the Insert tab. Note that not all tabs are available across all Office versions. For more information, see Office Add-ins manifest.

  - Place commands on the Home tab if the functionality doesn't fit on another tab, and you have fewer than six top-level commands. You can also add commands to the Home tab if your add-in needs to work across Office versions (such as Office on the web or desktop) and a tab is not available in all versions (for example, the Design tab doesn't exist in Office on the web).
  - Place commands on a custom tab if you have more than six top-level commands.
  - Name your group to match the name of your add-in. If you have multiple groups, name each group based on the functionality that the commands in that group provide.
  - Don't add unnecessary buttons to increase the real estate of your add-in.
  - Don't position a custom tab to the left of the Home tab, or give it focus by default when the document opens, unless your add-in is the primary way users will interact with the document. Giving excessive prominence to your add-in inconveniences and annoys users and administrators.
  - If your add-in is the primary way users interact with the document and you have a custom ribbon tab, consider integrating into the tab the buttons for the Office functions that users will frequently need.
  - If the functionality provided with a custom tab should only be available in certain contexts, use custom contextual tabs. If you use custom contextual tabs, make sure to implement a fallback experience for when your add-in runs on platforms that don't support custom contextual tabs.

  > **Note**
  >
  > Add-ins that take up too much space might not pass AppSource validation.

- For all icons, follow the icon design guidelines.

- Provide a version of your add-in that works on Office applications or platforms (such as iPad) that don't support commands. A single add-in manifest can be used for these versions.

Next steps

The best way to get started using add-in commands is to take a look at the Office Add-in commands samples on GitHub.

For more information about specifying add-in commands in an add-in only manifest, see Create add-in commands with the add-in only manifest and the VersionOverrides reference content.

For more information about specifying add-in commands in the unified manifest for Microsoft 365, see Create add-in commands with the unified manifest for Microsoft 365.

Create add-in commands with the add-in only manifest

Article • 03/04/2025

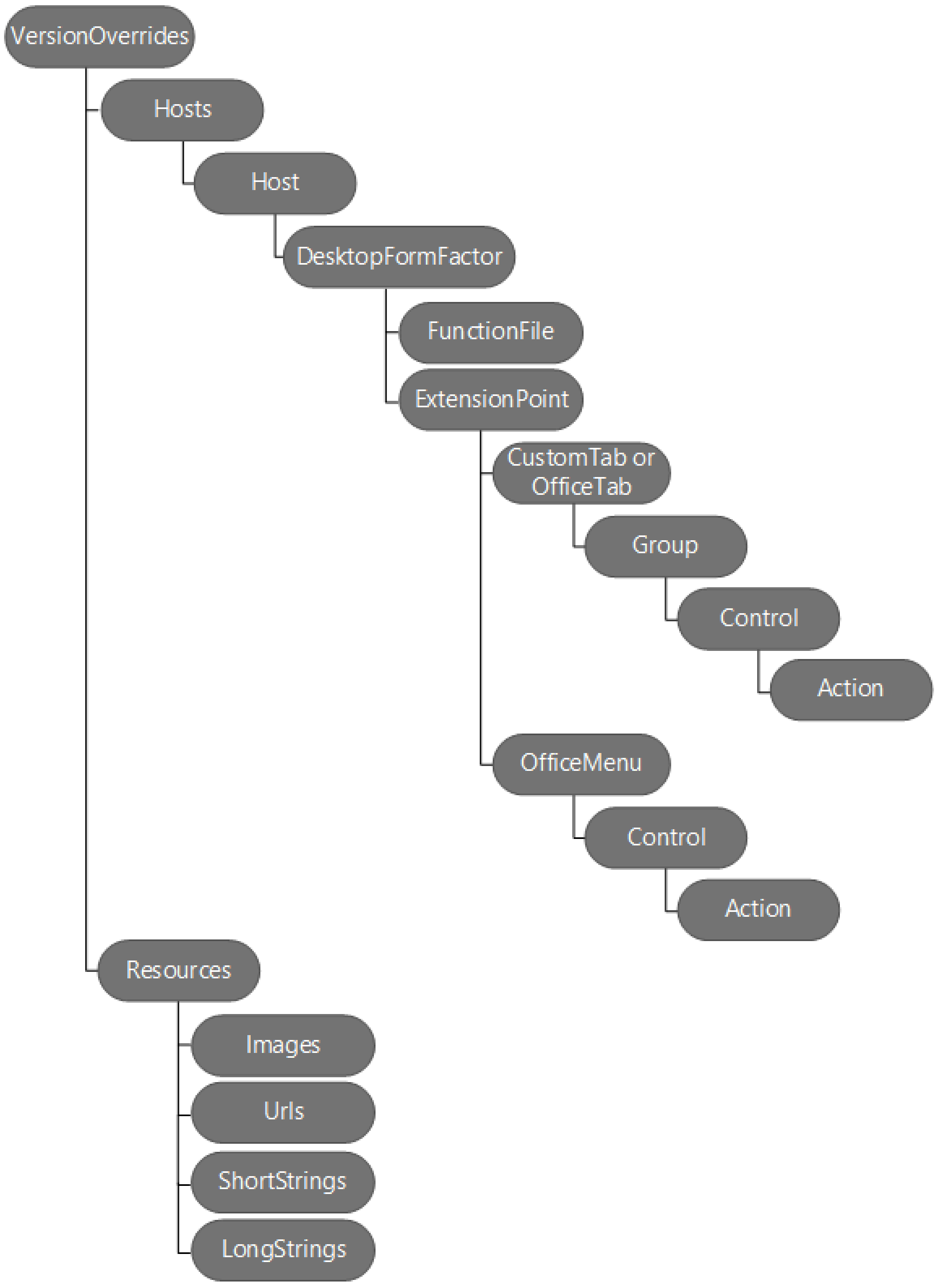
Add-in commands provide an easy way to customize the default Office user interface (UI) with specified UI elements that perform actions. For an introduction to add-in commands, see Add-in commands.

This article describes how to edit your add-in only manifest to define add-in commands and how to create the code for function commands.

Tip

For instructions on how to create add-in commands with the unified manifest for Microsoft 365, see **Create add-in commands with the unified manifest for Microsoft 365**.

The following diagram shows the hierarchy of elements used to define add-in commands. These elements are described in more detail in this article.



Sample commands

All the task pane add-ins created by Yo Office have add-in commands. They contain an add-in command (button) to show the task pane. Generate these projects by following one of the quick starts, such as Build an Excel task pane add-in. Ensure that you have read Add-in commands to understand command capabilities.

Important parts of an add-in command

The following steps explain how to add add-in commands to an existing add-in.

Step 1: Add VersionOverrides element

The **<VersionOverrides>** element is the root element that contains the definition of your add-in command. Details on the valid attributes and implications are found in Version overrides in the manifest.

The following example shows the **<VersionOverrides>** element and its child elements.

XML

```
<OfficeApp>
...
  <VersionOverrides
xmlns="http://schemas.microsoft.com/office/taskpaneappversionoverrides"
xsi:type="VersionOverridesV1_0">
    <Requirements>
      <!-- Add information about requirement sets. -->
    </Requirements>
    <Hosts>
      <Host xsi:type="Workbook">
        <!-- Add information about form factors. -->
      </Host>
    </Hosts>
    <Resources>
      <!-- Add information about resources. -->
    </Resources>
  </VersionOverrides>
...
</OfficeApp>
```

Step 2: Add Hosts, Host, and DesktopFormFactor elements

The **<Hosts>** element contains one or more **<Host>** elements. A **<Host>** element specifies a particular Office application. The **<Host>** element contains child elements that specify the add-in commands to display after your add-in is installed in that Office application. To show the same add-in commands in two or more different Office applications, you must duplicate the child elements in each **<Host>**.

The **<DesktopFormFactor>** element specifies the settings for an add-in that runs in Office on the web, Windows, and Mac.

The following example shows the **<Hosts>**, **<Host>**, and **<DesktopFormFactor>** elements.

```XML
<OfficeApp>
...
  <VersionOverrides
xmlns="http://schemas.microsoft.com/office/taskpaneappversionoverrides"
xsi:type="VersionOverridesV1_0">
  ...
    <Hosts>
      <Host xsi:type="Workbook">
        <DesktopFormFactor>

              <!-- Information about FunctionFile and ExtensionPoint. -->

        </DesktopFormFactor>
      </Host>
    </Hosts>
  ...
  </VersionOverrides>
...
</OfficeApp>
```

Step 3: Add the FunctionFile element

The **<FunctionFile>** element specifies a file that contains JavaScript or TypeScript code to run when an add-in command uses the **ExecuteFunction** action. The **<FunctionFile>** element's **resid** attribute is set to a HTML file that includes all the JavaScript or TypeScript files your add-in commands require. You can't link directly to a JavaScript or TypeScript file. You can only link to an HTML file. The file name is specified as a **<Url>** element in the **<Resources>** element.

> **Note**
>
> The Yo Office projects use webpack to avoid manually adding the JavaScript or TypeScript to the HTML.

The following is an example of the **<FunctionFile>** element.

```XML
<DesktopFormFactor>
    <FunctionFile resid="Commands.Url" />
    <ExtensionPoint xsi:type="PrimaryCommandSurface">
```

```
        <!-- Information about this extension point. -->
      </ExtensionPoint>

      <!-- You can define more than one ExtensionPoint element as needed. -->
</DesktopFormFactor>
```

ⓘ **Important**

Office.js must be initialized before the add-in command logic runs. For more information, see Initialize your Office Add-in.

Outlook notifications

When an add-in needs to provide status updates, such as progress indicators or error messages, it must do so through the notification APIs. The processing for the notifications must also be defined in a separate HTML file that is specified in the `FunctionFile` node of the manifest.

Step 4: Add ExtensionPoint elements

The <ExtensionPoint> element defines where add-in commands should appear in the Office UI.

The following examples show how to use the **<ExtensionPoint>** element with **PrimaryCommandSurface** and **ContextMenu** attribute values, and the child elements that should be used with each.

ⓘ **Important**

For elements that contain an ID attribute, make sure you provide a unique ID. We recommend that you use your company's name along with your ID. For example, use the following format: `<CustomTab id="mycompanyname.mygroupname">`.

XML

```xml
<ExtensionPoint xsi:type="PrimaryCommandSurface">
  <CustomTab id="Contoso Tab">
  <!-- If you want to use a default tab that comes with Office, remove the
above CustomTab element, and then uncomment the following OfficeTab element.
-->
  <!-- <OfficeTab id="TabData"> -->
    <Label resid="residLabel4" />
    <Group id="Group1Id12">
```

```
        <Label resid="residLabel4" />
        <Icon>
          <bt:Image size="16" resid="icon1_32x32" />
          <bt:Image size="32" resid="icon1_32x32" />
          <bt:Image size="80" resid="icon1_32x32" />
        </Icon>
        <Control xsi:type="Button" id="Button1Id1">

          <!-- Information about the control. -->
        </Control>
        <!-- Other controls, as needed. -->
      </Group>
    </CustomTab>
  </ExtensionPoint>
  <ExtensionPoint xsi:type="ContextMenu">
    <OfficeMenu id="ContextMenuCell">
      <Control xsi:type="Menu" id="ContextMenu2">
              <!-- Information about the control. -->
      </Control>
      <!-- Other controls, as needed. -->
    </OfficeMenu>
  </ExtensionPoint>
```

Step 5: Add Control elements

The <Control> element defines the usable surface of command, such as a button or menu, and the action associated with it.

Button controls

A button control performs a single action when the user selects it. It can either run a JavaScript or TypeScript function or show a task pane. The following example shows how to define two buttons. The first button runs a JavaScript function without showing a UI, and the second button shows a task pane. In the **<Control>** element:

- The **type** attribute is required, and must be set to **Button**.
- The **id** attribute of the **<Control>** element is a string with a maximum of 125 characters.
- The **xsi:type** attribute of the child **<Action>** element must be set to **ExecuteFunction** to run a function or **ShowTaskpane** to display a task pane.

XML

```xml
<!-- Define a control that calls a JavaScript function. -->
<Control xsi:type="Button" id="Button1Id1">
  <Label resid="residLabel" />
  <Supertip>
```

```
      <Title resid="residLabel" />
      <Description resid="residToolTip" />
    </Supertip>
    <Icon>
      <bt:Image size="16" resid="icon1_32x32" />
      <bt:Image size="32" resid="icon1_32x32" />
      <bt:Image size="80" resid="icon1_32x32" />
    </Icon>
    <Action xsi:type="ExecuteFunction">
      <FunctionName>highlightSelection</FunctionName>
    </Action>
  </Control>

  <!-- Define a control that shows a task pane. -->
  <Control xsi:type="Button" id="Button2Id1">
    <Label resid="residLabel2" />
    <Supertip>
      <Title resid="residLabel" />
      <Description resid="residToolTip" />
    </Supertip>
    <Icon>
      <bt:Image size="16" resid="icon2_32x32" />
      <bt:Image size="32" resid="icon2_32x32" />
      <bt:Image size="80" resid="icon2_32x32" />
    </Icon>
    <Action xsi:type="ShowTaskpane">
      <SourceLocation resid="residUnitConverterUrl" />
    </Action>
  </Control>
```

Menu controls

A menu control can be used with either **PrimaryCommandSurface** or **ContextMenu**, and defines:

- A root-level menu item.
- A list of submenu items.

When used with **PrimaryCommandSurface**, the root menu item displays as a button on the ribbon. When the button is selected, the submenu displays as a drop-down list. When used with **ContextMenu**, a menu item with a submenu is inserted on the context menu. In both cases, individual submenu items can either run a JavaScript or TypeScript function or show a task pane. Only one level of submenus is supported at this time.

The following example shows how to define a menu item with two submenu items. The first submenu item shows a task pane, and the second submenu item runs a JavaScript function. In the **<Control>** element:

- The **xsi:type** attribute is required, and must be set to **Menu**.

- The **id** attribute is a string with a maximum of 125 characters.

XML

```xml
<Control xsi:type="Menu" id="TestMenu2">
  <Label resid="residLabel3" />
  <Supertip>
    <Title resid="residLabel" />
    <Description resid="residToolTip" />
  </Supertip>
  <Icon>
    <bt:Image size="16" resid="icon1_32x32" />
    <bt:Image size="32" resid="icon1_32x32" />
    <bt:Image size="80" resid="icon1_32x32" />
  </Icon>
  <Items>
    <Item id="showGallery2">
      <Label resid="residLabel3"/>
      <Supertip>
        <Title resid="residLabel" />
        <Description resid="residToolTip" />
      </Supertip>
      <Icon>
        <bt:Image size="16" resid="icon1_32x32" />
        <bt:Image size="32" resid="icon1_32x32" />
        <bt:Image size="80" resid="icon1_32x32" />
      </Icon>
      <Action xsi:type="ShowTaskpane">
        <TaskpaneId>MyTaskPaneID1</TaskpaneId>
        <SourceLocation resid="residUnitConverterUrl" />
      </Action>
    </Item>
    <Item id="showGallery3">
      <Label resid="residLabel5"/>
      <Supertip>
        <Title resid="residLabel" />
        <Description resid="residToolTip" />
      </Supertip>
      <Icon>
        <bt:Image size="16" resid="icon4_32x32" />
        <bt:Image size="32" resid="icon4_32x32" />
        <bt:Image size="80" resid="icon4_32x32" />
      </Icon>
      <Action xsi:type="ExecuteFunction">
        <FunctionName>getButton</FunctionName>
      </Action>
    </Item>
  </Items>
</Control>
```

Sample code for function commands

The following code shows a function that's invoked by a button or menu item control whose **<Action>** element's **xsi:type** is set to **ExecuteFunction**. Note the following about the code.

- The Office.actions.associate call tells Office which function to run when a button or menu item is selected. The value passed to its **actionId** parameter must match the value specified in the **<FunctionName>** element of the manifest. You must have an `Office.actions.associate` call for every function command defined in the manifest.
- The event.completed call signals that you've successfully handled the event. When a function is called multiple times, such as multiple clicks on the same add-in command, all events are automatically queued. The first event runs automatically, while the other events remain on the queue. When your function calls `event.completed`, the next queued call to that function runs. You must implement `event.completed`, otherwise your function won't run.

JavaScript

```javascript
// Initialize the Office Add-in.
Office.onReady(() => {
  // If needed, Office.js is ready to be called.
});

// The command function.
async function highlightSelection(event) {

    // Implement your custom code here. The following code is a simple Excel
example.
    try {
          await Excel.run(async (context) => {
              const range = context.workbook.getSelectedRange();
              range.format.fill.color = "yellow";
              await context.sync();
          });
      } catch (error) {
          // Note: In a production add-in, notify the user through your add-
in's UI.
          console.error(error);
      }

    // Calling event.completed is required. The event.completed call lets
the platform know that processing has completed.
    event.completed();
}

// This maps the function to the action ID specified in the manifest.
Office.actions.associate("highlightSelection", highlightSelection);
```

Step 6: Add the Resources element

The <Resources> element contains resources used by the different child elements of the **<VersionOverrides>** element. Resources include icons, strings, and URLs. An element in the manifest can use a resource by referencing the **id** of the resource. Using the **id** helps organize the manifest, especially when there are different versions of the resource for different locales. An **id** has a maximum of 32 characters.

The following shows an example of how to use the **<Resources>** element. Each resource can have one or more <Override> child elements to define a different resource for a specific locale.

XML

```xml
<Resources>
  <bt:Images>
    <bt:Image id="icon1_16x16"
DefaultValue="https://www.contoso.com/Images/icon_default.png">
      <bt:Override Locale="ja-jp" Value="https://www.contoso.com/Images/ja-
jp16-icon_default.png" />
    </bt:Image>
    <bt:Image id="icon1_32x32"
DefaultValue="https://www.contoso.com/Images/icon_default.png">
      <bt:Override Locale="ja-jp" Value="https://www.contoso.com/Images/ja-
jp32-icon_default.png" />
    </bt:Image>
    <bt:Image id="icon1_80x80"
DefaultValue="https://www.contoso.com/Images/icon_default.png">
      <bt:Override Locale="ja-jp" Value="https://www.contoso.com/Images/ja-
jp80-icon_default.png" />
    </bt:Image>
  </bt:Images>
  <bt:Urls>
    <bt:Url id="residDesktopFuncUrl"
DefaultValue="https://www.contoso.com/Pages/Home.aspx">
      <bt:Override Locale="ja-jp"
Value="https://www.contoso.com/Pages/Home.aspx" />
    </bt:Url>
  </bt:Urls>
  <bt:ShortStrings>
    <bt:String id="residLabel" DefaultValue="GetData">
      <bt:Override Locale="ja-jp" Value="JA-JP-GetData" />
    </bt:String>
  </bt:ShortStrings>
  <bt:LongStrings>
    <bt:String id="residToolTip" DefaultValue="Get data for your document.">
      <bt:Override Locale="ja-jp" Value="JA-JP - Get data for your
document." />
    </bt:String>
  </bt:LongStrings>
</Resources>
```

> **Note**
>
> You must use Secure Sockets Layer (SSL) for all URLs in the **<Image>** and **<Url>** elements.

Outlook support notes

Add-in commands are available in the following Outlook versions.

- Outlook on the web for Microsoft 365 and Outlook.com
- Outlook on the web for Exchange 2016 or later
- new Outlook on Windows
- Outlook 2016 or later on Windows
- Outlook on Mac
- Outlook on Android
- Outlook on iOS

Support for add-in commands in Exchange 2016 requires Cumulative Update 5 .

If your add-in uses an add-in only manifest, then add-in commands are only available for add-ins that don't use ItemHasAttachment, ItemHasKnownEntity, or ItemHasRegularExpressionMatch rules to limit the types of items they activate on. However, contextual add-ins can present different commands depending on whether the currently selected item is a message or appointment, and can choose to appear in read or compose scenarios. Using add-in commands if possible is a best practice.

See also

- Add-in commands
- Sample: Create an Excel add-in with command buttons
- Sample: Create a Word add-in with command buttons
- Sample: Create a PowerPoint add-in with command buttons

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

Article • 05/19/2025

Add-in commands provide an easy way to customize the default Office user interface (UI) with specified UI elements that perform actions. For an introduction to add-in commands, see Add-in commands.

This article describes how to configure the Unified manifest for Microsoft 365 to define add-in commands and how to create the code for function commands.

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

Instructions for creating add-in commands with the add-in only manifest are in Create add-in commands with the add-in only manifest.

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.

Starting point and major steps

Both of the tools that create add-in projects with a unified manifest — the Office Yeoman generator and Microsoft 365 Agents Toolkit — create projects with one or more add-in commands. The only time you won't already have an add-in command is if you are updating an add-in which previously didn't have one.

Two decisions

- Decide which of two types of add-in commands you need: Task pane or function
- Decide which kind of UI element you need: button or menu item. Then carry out the steps in the sections and subsections below that correspond to your decisions.

Add a task pane command

The following subsections explain how to include a task pane command in an add-in.

Configure the runtime for the task pane command

1. Open the unified manifest and find the "extensions.runtimes" array.

2. Ensure that there is a runtime object that has an "actions.type" property with the value `"openPage"`. This type of runtime opens a task pane.

3. Ensure that the "requirements.capabilities" array contains an object that specifies a Requirement Set that supports add-in commands. For Outlook the minimum requirement set for add-in commands is Mailbox 1.3. For other Office host applications, the minimum requirement set for add-in commands is AddinCommands 1.1.

4. Ensure that the `"id"` of the runtime object has a descriptive name such as `"TaskPaneRuntime"`.

5. Ensure that the "code.page" property of the runtime object is set to the URL of the page that should open in the task pane, such as `"https://localhost:3000/taskpane.html"`.

6. Ensure that the "actions.view" of the runtime object has a name that describes the content of the page that you set in the preceding step, such as `"homepage"` or `"dashboard"`.

7. Ensure that the "actions.id" of the runtime object has a descriptive name such as `"ShowTaskPane"` that indicates what happens when the user selects the add-in command button or menu item.

8. Set the other properties and subproperties of the runtime object as shown in the following completed example of a runtime object. The `"type"` and `"lifetime"` properties are required and in Outlook Add-ins. They always have the values shown in this example.

JSON

```json
"runtimes": [
    {
        "requirements": {
            "capabilities": [
                {
                    "name": "Mailbox",
                    "minVersion": "1.3"
                }
            ]
        },
        "id": "TaskPaneRuntime",
        "type": "general",
        "code": {
            "page": "https://localhost:3000/taskpane.html"
        },
        "lifetime": "short",
        "actions": [
            {
                "id": "ShowTaskPane",
                "type": "openPage",
                "view": "homepage"
            }
        ]
    }
]
```

Configure the UI for the task pane command

1. Ensure that the extension object for which you configured a runtime has a "ribbons" array property as a peer to the "runtimes" array. There is typically only one extension object in the "extensions" array.

2. Ensure that the array has an object with array properties named `"contexts"` and `"tabs"`, as shown in the following example.

JSON

```json
"ribbons": [
    {
        "contexts": [
            // child objects omitted
        ],
        "tabs": [
            // child objects omitted
        ]
    }
]
```

3. Ensure that the `"contexts"` array has strings that specify the windows or panes in which the UI for the task pane command should appear. For example, `"mailRead"` means that it will appear in the reading pane or message window when an email message is open, but `"mailCompose"` means it will appear when a new message or a reply is being composed. The following are the allowable values:

   - `"mailRead"`
   - `"mailCompose"`
   - `"meetingDetailsOrganizer"`
   - `"meetingDetailsAttendee"`

   The following is an example.

   JSON

   ```json
   "contexts": [
       "mailRead"
   ],
   ```

4. Ensure that the `"tabs"` array has an object with a `"builtInTabId"` string property that is set to the ID of ribbon tab in which you want your task pane command to appear. Also, ensure that there is a `"groups"` array with at least one object in it. The following is an example.

   JSON

   ```json
   "tabs": [
       {
           "builtInTabID": "TabDefault",
           "groups": [
               {
                   // properties omitted
               }
           ]
       }
   ]
   ```

   Note

   For a list of the possible values of the `"builtInTabID"` property, see Find the IDs of built-in Office ribbon tabs.

5. Ensure that the `"groups"` array has an object to define the custom control group that will hold your add-in command UI controls. The following is an example. Note the following about this JSON:

   - The `"id"` must be unique across all groups in all ribbon objects in the manifest. Maximum length is 64 characters.
   - The `"label"` appears on the group on the ribbon. Although its maximum length is 64 characters, to ensure that the control group fits correctly in the ribbon, we recommend that you limit the `"label"` to 16 characters.
   - One of the `"icons"` appears on the group only if the Office application window, and hence the ribbon, has been sized by the user too small for any of the controls in the group to appear. Office decides when to use one of these icons and which one to use based on the size of the window and the resolution of the device. You cannot control this. You must provide image files for 16, 32, and 80 pixels, while five other sizes are also supported (20, 24, 40, 48, and 64 pixels). You must use Secure Sockets Layer (SSL) for all URLs.

   JSON

   ```json
   "groups": [
       {
           "id": "msgReadGroup",
           "label": "Contoso Add-in",
           "icons": [
               {
                   "size": 16,
                   "url": "https://localhost:3000/assets/icon-16.png"
               },
               {
                   "size": 32,
                   "url": "https://localhost:3000/assets/icon-32.png"
               },
               {
                   "size": 80,
                   "url": "https://localhost:3000/assets/icon-80.png"
               }
           ],
           "controls": [
               {
                   // properties omitted
               }
           ]
       }
   ]
   ```

6. Ensure that there is a control object in the `"controls"` array for each button or custom menu you want. The following is an example. Note the following about this JSON:

- The `"id"`, `"label"`, and `"icons"` properties have the same purpose and the same restrictions as the corresponding properties of a group object, except that they apply to a specific button or menu within the group.
- The `"type"` property is set to `"button"` which means that the control will be a ribbon button. You can also configure a task pane command to be run from a menu item. See Menu and menu items.
- The `"supertip.title"` (maximum length: 64 characters) and `"supertip.description"` (maximum length: 128 characters) appear when the cursor is hovering over the button or menu.
- The `"actionId"` must be an exact match for the "runtimes.actions.id" that you set in Configure the runtime for the task pane command.

JSON

```json
{
    "id": "msgReadOpenPaneButton",
    "type": "button",
    "label": "Show Task Pane",
    "icons": [
        {
            "size": 16,
            "url": "https://localhost:3000/assets/icon-16.png"
        },
        {
            "size": 32,
            "url": "https://localhost:3000/assets/icon-32.png"
        },
        {
            "size": 80,
            "url": "https://localhost:3000/assets/icon-80.png"
        }
    ],
    "supertip": {
        "title": "Show Contoso Task Pane",
        "description": "Opens the Contoso task pane."
    },
    "actionId": "ShowTaskPane"
}
```

You've now completed adding a task pane command to your add-in. Sideload and test it.

Add a function command

The following subsections explain how to include a function command in an add-in.

Create the code for the function command

1. Ensure that your source code includes a JavaScript or Typescript file with the function that you want to run with your function command. The following is an example. Since this article is about creating add-in commands, and not about teaching the Office JavaScript Library, we provide it with minimal comments, but do note the following:

   - For purposes of this article, the file is named **commands.js**.
   - The function will cause a small notification to appear on an open email message with the text "Action performed".
   - Like all code that call APIs in the Office JavaScript Library, it must begin by initializing the library. It does this by calling `Office.onReady`.
   - The last thing the code calls is Office.actions.associate to tell Office which function in the file should be run when the UI for your function command is invoked. The function maps the function name to an action ID that you configure in the manifest in a later step. If you define multiple function commands in the same file, your code must call `associate` for each one.
   - The function must take a parameter of type Office.AddinCommands.Event. The last line of the function must call event.completed.

   JavaScript

   ```javascript
   Office.onReady(function() {
   // Add any initialization code here.
   });

   function setNotification(event) {
   const message = {
       type:
   Office.MailboxEnums.ItemNotificationMessageType.InformationalMessage,
       message: "Performed action.",
       icon: "Icon.80x80",
       persistent: true,
   };

   // Show a notification message.
   Office.context.mailbox.item.notificationMessages.replaceAsync("ActionPerforma
   nceNotification", message);

   // Be sure to indicate when the add-in command function is complete.
   event.completed();
   }

   // Map the function to the action ID in the manifest.
   Office.actions.associate("SetNotification", setNotification);
   ```

2. Ensure that your source code includes an HTML file that is configured to load the function file you created. The following is an example. Note the following about this JSON:

- For purposes of this article, the file is named **commands.html**.
- The `<body>` element is empty because the file has no UI. Its only purpose is to load JavaScript files.
- The Office JavaScript Library and the **commands.js** file that you created in the preceding step is explicitly loaded.

  > ⓘ **Note**
  >
  > It's common in Office Add-in development to use tools like webpack and its plugins to automatically inject `<script>` tags into HTML files at build time. If you use such a tool, you shouldn't include any `<script>` tags in your source file that are going to be inserted by the tool.

HTML

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=Edge" />

        <!-- Office JavaScript Library -->
        <script type="text/javascript"
src="https://appsforoffice.microsoft.com/lib/1/hosted/office.js"></script>
        <!-- Function command file -->
        <script src="commands.js" type="text/javascript"></script>
    </head>
    <body>
    </body>
</html>
```

Configure the runtime for the function command

1. Open the unified manifest and find the `"extensions.runtimes"` array.
2. Ensure that there is a runtime object that has a `"actions.type"` property with the value `"executeFunction"`.
3. Ensure that the `"requirements.capabilities"` array contains objects that specify any Requirement Sets that are needed to support the APIs add-in commands. For Outlook, the minimum requirement set for add-in commands is Mailbox 1.3. But if your function command calls that API that is part of later **Mailbox** requirement set, such as **Mailbox 1.5**,

then you need to specify the later version (e.g., "1.5") as the `"minVersion"` value. For other Office host applications, the minimum requirement set for add-in commands is AddinCommands 1.1.

4. Ensure that the `"id"` of the runtime object has a descriptive name such as "CommandsRuntime".

5. Ensure that the "code.page" property of the runtime object is set to the URL of the UI-less HTML page that loads your function file, such as `"https://localhost:3000/commands.html"`.

6. Ensure that the `"actions.id"` of the runtime object has a descriptive name such as "SetNotification" that indicates what happens when the user selects the add-in command button or menu item.

   Important

   The value of `"actions.id"` must exactly match the first parameter of the call to `Office.actions.associate` in the function file.

7. Set the other properties and subproperties of the runtime object as shown in the following completed example of a runtime object.

   JSON

```json
"runtimes": [
    {
        "id": "CommandsRuntime",
        "type": "general",
        "code": {
            "page": "https://localhost:3000/commands.html"
        },
        "lifetime": "short",
        "actions": [
            {
                "id": "SetNotification",
                "type": "executeFunction",
            }
        ]
    }
]
```

Configure the UI for the function command

1. Ensure that the extension object for which you configured a runtime has a `"ribbons"` array property as a peer to the `"runtimes"` array. There is typically only one extension object in the `"extensions"` array.

2. Ensure that the array has an object with array properties named `"contexts"` and `"tabs"`, as shown in the following example.

   JSON

   ```json
   "ribbons": [
       {
           "contexts": [
               // child objects omitted
           ],
           "tabs": [
               // child objects omitted
           ]
       }
   ]
   ```

3. Ensure that the `"contexts"` array has strings that specify the windows or panes in which the UI for the function command should appear. For example, `"mailRead"` means that it will appear in the reading pane or message window when an email message is open, but `"mailCompose"` means it will appear when a new message or a reply is being composed. The following are the allowable values:

   - `"mailRead"`
   - `"mailCompose"`
   - `"meetingDetailsOrganizer"`
   - `"meetingDetailsAttendee"`

   The following is an example.

   JSON

   ```json
   "contexts": [
       "mailRead"
   ],
   ```

4. Ensure that the `"tabs"` array has an object with a `"builtInTabId"` string property that is set to the ID of ribbon tab in which you want your function command to appear and a `"groups"` array with at least one object in it. The following is an example.

   JSON

```
"tabs": [
    {
        "builtInTabID": "TabDefault",
        "groups": [
            {
                // properties omitted
            }
        ]
    }
]
```

> ⓘ **Note**
>
> For a list of the possible values of the `"builtInTabID"` property, see Find the IDs of built-in Office ribbon tabs.

5. Ensure that the `"groups"` array has an object to define the custom control group that will hold your add-in command UI controls. The following is an example. Note the following about this JSON:

    - The `"id"` must be unique across all groups in all ribbon objects in the manifest. Maximum length is 64 characters.
    - The `"label"` appears on the group on the ribbon. Although its maximum length is 64 characters, to ensure that the control group fits correctly in the ribbon, we recommend that you limit the `"label"` to 16 characters.
    - One of the `"icons"` appears on the group only if the Office application window, and hence the ribbon, has been sized by the user too small for any of the controls in the group to appear. Office decides when to use one of these icons and which one to use based on the size of the window and the resolution of the device. You cannot control this. You must provide image files for 16, 32, and 80 pixels, while five other sizes are also supported (20, 24, 40, 48, and 64 pixels). You must use Secure Sockets Layer (SSL) for all URLs.

JSON

```json
"groups": [
    {
        "id": "msgReadGroup",
        "label": "Contoso Add-in",
        "icons": [
            {
                "size": 16,
                "url": "https://localhost:3000/assets/icon-16.png"
            },
            {
                "size": 32,
                "url": "https://localhost:3000/assets/icon-32.png"
            },
            {
                "size": 80,
                "url": "https://localhost:3000/assets/icon-80.png"
            }
        ],
        "controls": [
            {
                // properties omitted
            }
        ]
    }
]
```

6. Ensure that there is a control object in the `"controls"` array for each button or custom menu you want. The following is an example. Note the following about this JSON:

   - The `"id"`, `"label"`, and `"icons"` properties have the same purpose and the same restrictions as the corresponding properties of a group object, except that they apply to a specific button or menu within the group.
   - The `"type"` property is set to `"button"` which means that the control will be a ribbon button. You can also configure a function command to be run from a menu item. See Menu and menu items.
   - The `"supertip.title"` (maximum length: 64 characters) and `"supertip.description"` (maximum length: 128 characters) appear when the cursor is hovering over the button or menu.
   - The `"actionId"` must be an exact match for the `"runtime.actions.id"` that you set in Configure the runtime for the function command.

JSON

```json
{
    "id": "msgReadSetNotificationButton",
    "type": "button",
    "label": "Set Notification",
    "icons": [
        {
            "size": 16,
            "url": "https://localhost:3000/assets/icon-16.png"
        },
        {
            "size": 32,
            "url": "https://localhost:3000/assets/icon-32.png"
        },
        {
            "size": 80,
            "url": "https://localhost:3000/assets/icon-80.png"
        }
    ],
    "supertip": {
        "title": "Set Notification",
        "description": "Displays a notification message on the current
message."
    },
    "actionId": "SetNotification"
}
```

You've now completed adding a function command to your add-in. Sideload and test it.

Menu and menu items

In addition to custom buttons, you can also add custom drop down menus to the Office ribbon. This section explains how by using an example with two menu items. One invokes a task pane command. The other invokes a function command.

Configure the runtimes and code

Carry out the steps of the following sections:

- Configure the runtime for the task pane command
- Create the code for the function command
- Configure the runtime for the function command

Configure the UI for the menu

1. Ensure that the extension object for which you configured a runtime has a `"ribbons"` array property as a peer to the `"runtimes"` array. There is typically only one extension object in the `"extensions"` array.

2. Ensure that the array has an object with array properties named `"contexts"` and `"tabs"`, as shown in the following example.

JSON

```json
"ribbons": [
    {
        "contexts": [
            // child objects omitted
        ],
        "tabs": [
            // child objects omitted
        ]
    }
]
```

3. Ensure that the `"contexts"` array has strings that specify the windows or panes in which the menu should appear on the ribbon. For example, `"mailRead"` means that it will appear in the reading pane or message window when an email message is open, but `"mailCompose"` means it will appear when a new message or a reply is being composed. The following are the allowable values:

   - `"mailRead"`
   - `"mailCompose"`
   - `"meetingDetailsOrganizer"`
   - `"meetingDetailsAttendee"`

   The following is an example.

   JSON

   ```json
   "contexts": [
       "mailRead"
   ],
   ```

4. Ensure that the `"tabs"` array has an object with a `"builtInTabId"` string property that is set to the ID of ribbon tab in which you want your task pane command to appear and a `"groups"` array with at least one object in it. The following is an example.

   JSON

   ```json
   "tabs": [
       {
           "builtInTabID": "TabDefault",
           "groups": [
               {
                   // properties omitted
               }
           ]
       }
   ]
   ```

   Note

   For a list of the possible values of the `"builtInTabID"` property, see Find the IDs of built-in Office ribbon tabs.

5. Ensure that the `"groups"` array has an object to define the custom control group that will hold your drop down menu control. The following is an example. Note the following about this JSON:

   - The `"id"` must be unique across all groups in all ribbon objects in the manifest. Maximum length is 64 characters.
   - The `"label"` appears on the group on the ribbon. Although its maximum length is 64 characters, to ensure that the control group fits correctly in the ribbon, we recommend that you limit the `"label"` to 16 characters.
   - One of the `"icons"` appears on the group only if the Office application window, and hence the ribbon, has been sized by the user too small for any of the controls in the group to appear. Office decides when to use one of these icons and which one to use based on the size of the window and the resolution of the device. You cannot control this. You must provide image files for 16, 32, and 80 pixels, while five other sizes are also supported (20, 24, 40, 48, and 64 pixels). You must use Secure Sockets Layer (SSL) for all URLs.

   JSON

   ```json
   "groups": [
       {
           "id": "msgReadGroup",
           "label": "Contoso Add-in",
           "icons": [
               {
                   "size": 16,
                   "url": "https://localhost:3000/assets/icon-16.png"
               },
               {
                   "size": 32,
                   "url": "https://localhost:3000/assets/icon-32.png"
               },
               {
                   "size": 80,
                   "url": "https://localhost:3000/assets/icon-80.png"
               }
           ],
           "controls": [
               {
                   // properties omitted
               }
           ]
       }
   ]
   ```

6. Ensure that there is a control object in the `"controls"` array. The following is an example. Note the following about this JSON:

- The `"id"`, `"label"`, and `"icons"` properties have the same purpose and the same restrictions as the corresponding properties of a group object, except that they apply to the drop down menu within the group.
- The `"type"` property is set to `"menu"` which means that the control will be a drop down menu.
- The `"supertip.title"` (maximum length: 64 characters) and `"supertip.description"` (maximum length: 128 characters) appear when the cursor is hovering over the menu.
- The `"items"` property contains the JSON for the two menu options. The values are added in later steps.

JSON

```json
{
    "id": "msgReadMenu",
    "type": "menu",
    "label": "Contoso Menu",
    "icons": [
        {
            "size": 16,
            "url": "https://localhost:3000/assets/icon-16.png"
        },
        {
            "size": 32,
            "url": "https://localhost:3000/assets/icon-32.png"
        },
        {
            "size": 80,
            "url": "https://localhost:3000/assets/icon-80.png"
        }
    ],
    "supertip": {
        "title": "Show Contoso Actions",
        "description": "Opens the Contoso menu."
    },
    "items": [
        {
            "id": "",
            "type": "",
            "label": "",
            "supertip": {},
            "actionId": ""
        },
        {
            "id": "",
            "type": "",
            "label": "",
            "supertip": {},
            "actionId": ""
        }
    ]
}
```

7. The first item shows a task pane. The following is an example. Note the following about this code:

   - The `"id"`, `"label"`, and `"supertip"` properties have the same purpose and the same restrictions as the corresponding properties of the parent menu object, except that they apply to just this menu option.
   - The `"icons"` property is optional for menu items and there isn't one in this example. If you include one, it has the same purposes and restrictions as the `"icons"` property of the parent menu, except that the icon appears on the menu item beside the label.
   - The `"type"` property is set to `"menuItem"`.
   - The `"actionId"` must be an exact match for the `"runtimes.actions.id"` that you set in Configure the runtime for the task pane command.

```JSON
{
    "id": "msgReadOpenPaneMenuItem",
    "type": "menuItem",
    "label": "Show Task Pane",
    "supertip": {
        "title": "Show Contoso Task Pane",
        "description": "Opens the Contoso task pane."
    },
    "actionId": "ShowTaskPane"
},
```

8. The second item runs a function command. The following is an example. Note the following about this code:

   - The `"actionId"` must be an exact match for the `"runtimes.actions.id"` that you set in Configure the runtime for the function command.

```JSON
{
    "id": "msgReadSetNotificationMenuItem",
    "type": "menuItem",
    "label": "Set Notification",
    "supertip": {
        "title": "Set Notification",
        "description": "Displays a notification message on the current
message."
```

```
    },
    "actionId": "SetNotification"
}
```

You've now completed adding a menu to your add-in. Sideload and test it.

See also

- Add-in commands
- Unified manifest for Microsoft 365.

Create custom contextual tabs in Office Add-ins

06/30/2025

A contextual tab is a hidden tab control in the Office ribbon that's displayed in the tab row when a specified event occurs in the Office document. For example, the **Table Design** tab that appears on the Excel ribbon when a table is selected. You include custom contextual tabs in your Office Add-in and specify when they're visible or hidden, by creating event handlers that change the visibility. (However, custom contextual tabs don't respond to focus changes.)

> **Note**
>
> This article assumes that you're familiar with Basic concepts for add-in commands. Please review it if you haven't worked with add-in commands (custom menu items and ribbon buttons) recently.

Prerequisites

Custom contextual tabs are currently only supported on **Excel** and only on the following platforms and builds.

- Excel on the web
- Excel on Windows: Version 2102 (Build 13801.20294) and later.
- Excel on Mac: Version 16.53 (21080600) and later.

Additionally, custom contextual tabs only work on platforms that support the following requirement sets. For more about requirement sets and how to work with them, see Specify Office applications and API requirements.

- RibbonApi 1.2
- SharedRuntime 1.1

> **Tip**
>
> Use the runtime checks in your code to test whether the user's host and platform combination supports these requirement sets as described in Check for API availability at runtime. (The technique of specifying the requirement sets in the manifest, which is also described in that article, doesn't currently work for RibbonApi 1.2.) Alternatively, you can implement an alternate UI experience when custom contextual tabs aren't supported.

Behavior of custom contextual tabs

The user experience for custom contextual tabs follows the pattern of built-in Office contextual tabs. The following are the basic principles for the placement custom contextual tabs.

- When a custom contextual tab is visible, it appears on the right end of the ribbon.
- If one or more built-in contextual tabs and one or more custom contextual tabs from add-ins are visible at the same time, the custom contextual tabs are always to the right of all of the built-in contextual tabs.
- If your add-in has more than one contextual tab and there are contexts in which more than one is visible, they appear in the order in which they are defined in your add-in. (The direction is the same direction as the Office language; that is, is left-to-right in left-to-right languages, but right-to-left in right-to-left languages.) See Define the groups and controls that appear on the tab for details about how you define them.
- If more than one add-in has a contextual tab that's visible in a specific context, then they appear in the order in which the add-ins were launched.
- Custom *contextual* tabs, unlike custom core tabs, aren't added permanently to the Office application's ribbon. They're present only in Office documents on which your add-in is running.

Major steps for including a contextual tab in an add-in

The following are the major steps for including a custom contextual tab in an add-in.

1. Configure the add-in to use a shared runtime.
2. Specify the icons for your contextual tab.
3. Define the groups and controls that appear on the tab.
4. Register the contextual tab with Office.
5. Specify the circumstances when the tab will be visible.

Configure the add-in to use a shared runtime

Adding custom contextual tabs requires your add-in to use the shared runtime. For more information, see Configure an add-in to use a shared runtime.

Specify the icons for your contextual tab

Before you can customize your contextual tab, you must first specify any icons that will appear on it in your add-in's manifest. Each icon must have at least three sizes: 16x16 px, 32x32 px, and 80x80 px. Select the tab for the type of manifest your add-in uses.

Unified manifest for Microsoft 365

In the "extensions.ribbons.tabs.groups.icons" array, specify the icons for the group of contextual tab controls that will be displayed on the host's ribbon. For icons that will be used by the tab's buttons and menus, specify these in the `"icons"` property of the "extensions.ribbons.tabs.groups.controls" object.

Because the contextual tab will only be shown when a certain event occurs, you must also set the "extensions.ribbons.tabs.groups.controls.overriddenByRibbonApi" property to `true`.

The following is an example.

JSON

```
"ribbons": [
    {
        ...
        "tabs": [
            "groups": [
                {
                    "id": "ContosoGroup",
                    ...
                    "icons": [
                        {
                            "size": 16,
                            "url":
"https://cdn.contoso.com/addins/datainsertion/Images/Group16x16.png"
                        },
                        {
                            "size": 32,
                            "url":
"https://cdn.contoso.com/addins/datainsertion/Images/Group32x32.png"
                        },
                        {
                            "size": 80,
                            "url":
"https://cdn.contoso.com/addins/datainsertion/Images/Group80x80.png"
                        }
                    ],
                    "controls": [
                        {
                            "id": "WriteDataButton",
                            ...
                            "icons": [
                                {
                                    "size": 16,
                                    "url":
```

```
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton16x16.png"
                                },
                                {
                                    "size": 32,
                                    "url":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton16x16.png"
                                },
                                {
                                    "size": 80,
                                    "url":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton16x16.png"
                                }
                            ],
                            ...
                            "overriddenByRibbonApi": true
                        },
                        ...
                    ]
                }
            ]
        ]
    }
],
```

ⓘ **Important**

When you move your add-in from development to staging or production, remember to update the URLs in your manifest as needed (such as changing the domain from `localhost` to `contoso.com`).

Define the groups and controls that appear on the tab

Unlike custom core tabs, which are defined in the manifest, custom contextual tabs are defined at runtime with a JSON blob. Your code parses the blob into a JavaScript object, and then passes the object to the Office.ribbon.requestCreateControls method. Custom contextual tabs are only present in documents on which your add-in is currently running. This is different from custom core tabs which are added to the Office application ribbon when the add-in is installed and remain present when another document is opened. Also, the `requestCreateControls` method may be run only once in a session of your add-in. If it's called again, an error is thrown.

We'll construct an example of a contextual tabs JSON blob step-by-step. The full schema for the contextual tab JSON is at dynamic-ribbon.schema.json. If you're working in Visual Studio

Code, you can use this file to get IntelliSense and to validate your JSON. For more information, see Editing JSON with Visual Studio Code - JSON schemas and settings.

1. Begin by creating a JSON string with two array properties named `actions` and `tabs`. The `actions` array is a specification of all the functions that can be executed by controls on the contextual tab. The `tabs` array defines one or more contextual tabs.

   JSON

   ```json
   '{
     "actions": [

     ],
     "tabs": [

     ]
   }'
   ```

2. This simple example of a contextual tab will have only a single button and, thus, only a single action. Add the following as the only member of the `actions` array. About this markup, note:

   - The `id` and `type` properties are mandatory.
   - The value of `type` can be either `"ExecuteFunction"` or `"ShowTaskpane"`.
   - The `functionName` property is only used when the value of `type` is `ExecuteFunction`. It's the name of a function defined in the FunctionFile. For more information about the FunctionFile, see Basic concepts for add-in commands.
   - In a later step, you'll map this action to a button on the contextual tab.

   JSON

   ```json
   {
     "id": "executeWriteData",
     "type": "ExecuteFunction",
     "functionName": "writeData"
   }
   ```

3. Add the following as the only member of the `tabs` array. About this markup, note:

   - The `id` property is required. Use a brief, descriptive ID that is unique among all contextual tabs in your add-in.
   - The `label` property is required. It's a user-friendly string to serve as the label of the contextual tab.

- The `groups` property is required. It defines the groups of controls that will appear on the tab. It must have at least one member.

> ⓘ **Note**
>
> The tab object can also have an optional `visible` property that specifies whether the tab is visible immediately when the add-in starts up. Since contextual tabs are normally hidden until a user event triggers their visibility (such as the user selecting an entity of some type in the document), the `visible` property defaults to `false` when not present. In a later section, we show how to set the property to `true` in response to an event.

JSON

```json
{
  "id": "CtxTab1",
  "label": "Contoso Data",
  "groups": [

  ]
}
```

4. In the simple ongoing example, the contextual tab has only a single group. Add the following as the only member of the `groups` array. About this markup, note:

   - All the properties are required.
   - The `id` property must be unique among all the groups in the manifest. Use a brief, descriptive ID, of up to 125 characters.
   - The `label` is a user-friendly string to serve as the label of the group.
   - The `icon` property's value is an array of objects that specify the icons that the group will have on the ribbon depending on the size of the ribbon and the Office application window.
   - The `controls` property's value is an array of objects that specify the buttons and menus in the group. There must be at least one.

JSON

```json
{
    "id": "CustomGroup111",
    "label": "Insertion",
    "icon": [

    ],
    "controls": [
    ]
}
```

5. Every group must have an icon of at least three sizes: 16x16 px, 32x32 px, and 80x80 px. Optionally, you can also have icons of sizes 20x20 px, 24x24 px, 40x40 px, 48x48 px, and 64x64 px. Office decides which icon to use based on the size of the ribbon and Office application window. Add the following objects to the icon array. (If the window and ribbon sizes are large enough for at least one of the *controls* on the group to appear, then no group icon at all appears. For an example, watch the **Styles** group on the Word ribbon as you shrink and expand the Word window.) About this markup, note:

   - Both the properties are required.
   - The `size` property unit of measure is pixels. Icons are always square, so the number is both the height and the width.
   - The `sourceLocation` property specifies the full URL to the icon. Its value must match the URL specified in the **<Image>** element of the **<Resources>** section of your manifest (see Specify the icons for your contextual tab).

> **Important**
>
> Just as you typically must change the URLs in the add-in's manifest when you move from development to production, you must also change the URLs in your contextual tabs JSON.

JSON

```json
{
    "size": 16,
    "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group16x16.png"
},
{
    "size": 32,
    "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group32x32.png"
},
{
    "size": 80,
    "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group80x80.png"
}
```

6. In our simple ongoing example, the group has only a single button. Add the following object as the only member of the `controls` array. About this markup, note:

- All the properties, except `enabled`, are required.
- `type` specifies the type of control. The values can be `"Button"`, `"Menu"`, or `"MobileButton"`.
- `id` can be up to 125 characters.
- `actionId` must be the ID of an action defined in the `actions` array. (See step 1 of this section.)
- `label` is a user-friendly string to serve as the label of the button.
- `superTip` represents a rich form of tool tip. Both the `title` and `description` properties are required.
- `icon` specifies the icons for the button. The previous remarks about the group icon apply here too.
- `enabled` (optional) specifies whether the button is enabled when the contextual tab appears starts up. The default if not present is `true`.

JSON

```json
{
    "type": "Button",
    "id": "CtxBt112",
    "actionId": "executeWriteData",
    "enabled": false,
    "label": "Write Data",
    "superTip": {
        "title": "Data Insertion",
        "description": "Use this button to insert data into the document."
    },
    "icon": [
        {
            "size": 16,
            "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton16x16.png
"
        },
        {
            "size": 32,
            "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton32x32.png
"
        },
        {
            "size": 80,
            "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton80x80.png
"
        }
    ]
}
```

The following is the complete example of the JSON blob.

JSON

```
`{
  "actions": [
    {
      "id": "executeWriteData",
      "type": "ExecuteFunction",
      "functionName": "writeData"
    }
  ],
  "tabs": [
    {
      "id": "CtxTab1",
      "label": "Contoso Data",
      "groups": [
        {
          "id": "CustomGroup111",
          "label": "Insertion",
          "icon": [
            {
                "size": 16,
                "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group16x16.png"
            },
            {
                "size": 32,
                "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group32x32.png"
            },
            {
                "size": 80,
                "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/Group80x80.png"
            }
          ],
          "controls": [
            {
                "type": "Button",
                "id": "CtxBt112",
                "actionId": "executeWriteData",
                "enabled": false,
                "label": "Write Data",
                "superTip": {
                    "title": "Data Insertion",
                    "description": "Use this button to insert data into the
document."
                },
                "icon": [
                    {
                        "size": 16,
                        "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton16x16.png"
```

```
                        },
                        {
                            "size": 32,
                            "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton32x32.png"
                        },
                        {
                            "size": 80,
                            "sourceLocation":
"https://cdn.contoso.com/addins/datainsertion/Images/WriteDataButton80x80.png"
                        }
                    ]
                }
            ]
        }
    ]
  }
]
}`
```

Register the contextual tab with Office with requestCreateControls

The contextual tab is registered with Office by calling the Office.ribbon.requestCreateControls method. This is typically done in either the function that's assigned to `Office.initialize` or with the `Office.onReady` function. For more about these functions and initializing the add-in, see Initialize your Office Add-in. You can, however, call the method anytime after initialization.

> **Important**
>
> The `requestCreateControls` method may be called only once in a given session of an add-in. An error is thrown if it's called again.

The following is an example. Note that the JSON string must be converted to a JavaScript object with the `JSON.parse` method before it can be passed to a JavaScript function.

JavaScript

```javascript
Office.onReady(async () => {
    const contextualTabJSON = ` ... `; // Assign the JSON string such as the one
at the end of the preceding section.
    const contextualTab = JSON.parse(contextualTabJSON);
    await Office.ribbon.requestCreateControls(contextualTab);
});
```

Specify the contexts when the tab will be visible with requestUpdate

Typically, a custom contextual tab should appear when a user-initiated event changes the add-in context. Consider a scenario in which the tab should be visible when, and only when, a chart (on the default worksheet of an Excel workbook) is activated.

Begin by assigning handlers. This is commonly done in the `Office.onReady` function as in the following example which assigns handlers (created in a later step) to the `onActivated` and `onDeactivated` events of all the charts in the worksheet.

JavaScript

```
Office.onReady(async () => {
    const contextualTabJSON = ` ... `; // Assign the JSON string.
    const contextualTab = JSON.parse(contextualTabJSON);
    await Office.ribbon.requestCreateControls(contextualTab);

    await Excel.run(context => {
        const charts = context.workbook.worksheets
            .getActiveWorksheet()
            .charts;
        charts.onActivated.add(showDataTab);
        charts.onDeactivated.add(hideDataTab);
        return context.sync();
    });
});
```

Next, define the handlers. The following is a simple example of a `showDataTab`, but see Handling the HostRestartNeeded error later in this article for a more robust version of the function. About this code, note:

- Office controls when it updates the state of the ribbon. The Office.ribbon.requestUpdate method queues a request to update. The method will resolve the `Promise` object as soon as it has queued the request, not when the ribbon actually updates.
- The parameter for the `requestUpdate` method is a RibbonUpdaterData object that (1) specifies the tab by its ID *exactly as specified in the JSON* and (2) specifies visibility of the tab.
- If you have more than one custom contextual tab that should be visible in the same context, you simply add additional tab objects to the `tabs` array.

JavaScript

```
async function showDataTab() {
    await Office.ribbon.requestUpdate({
        tabs: [
            {
                id: "CtxTab1",
                visible: true
            }
        ]});
}
```

The handler to hide the tab is nearly identical, except that it sets the `visible` property back to `false`.

The Office JavaScript library also provides several interfaces (types) to make it easier to construct the `RibbonUpdateData` object. The following is the `showDataTab` function in TypeScript and it makes use of these types.

TypeScript

```typescript
const showDataTab = async () => {
    const myContextualTab: Office.Tab = {id: "CtxTab1", visible: true};
    const ribbonUpdater: Office.RibbonUpdaterData = { tabs: [ myContextualTab ]};
    await Office.ribbon.requestUpdate(ribbonUpdater);
}
```

Toggle tab visibility and the enabled status of a button at the same time

The `requestUpdate` method is also used to toggle the enabled or disabled status of a custom button on either a custom contextual tab or a custom core tab. For details about this, see Change the availability of add-in commands. There may be scenarios in which you want to change both the visibility of a tab and the enabled status of a button at the same time. You do this with a single call of `requestUpdate`. The following is an example in which a button on a core tab is enabled at the same time as a contextual tab is made visible.

JavaScript

```javascript
function myContextChanges() {
    Office.ribbon.requestUpdate({
        tabs: [
            {
                id: "CtxTab1",
                visible: true
            },
            {
                id: "OfficeAppTab1",
                groups: [
                    {
                        id: "CustomGroup111",
                        controls: [
                            {
                                id: "MyButton",
                                enabled: true
                            }
                        ]
                    }
                ]
            ]}
        ]
    });
}
```

In the following example, the button that's enabled is on the very same contextual tab that is being made visible.

JavaScript

```javascript
function myContextChanges() {
    Office.ribbon.requestUpdate({
        tabs: [
            {
                id: "CtxTab1",
                visible: true,
                groups: [
                    {
                        id: "CustomGroup111",
                        controls: [
                            {
                                id: "MyButton",
                                enabled: true
                            }
                        ]
                    }
                ]
            }
        ]
    });
}
```

Open a task pane from contextual tabs

To open your task pane from a button on a custom contextual tab, create an action in the JSON with a `type` of `ShowTaskpane`. Then define a button with the `actionId` property set to the `id` of the action. This opens the default task pane specified in your manifest.

JSON

```json
`{
  "actions": [
    {
      "id": "openChartsTaskpane",
      "type": "ShowTaskpane",
      "title": "Work with Charts",
      "supportPinning": false
    }
  ],
  "tabs": [
    {
      // some tab properties omitted
      "groups": [
        {
          // some group properties omitted
          "controls": [
            {
                "type": "Button",
                "id": "CtxBt112",
                "actionId": "openChartsTaskpane",
                "enabled": false,
                "label": "Open Charts Taskpane",
                // some control properties omitted
            }
          ]
        }
      ]
    }
  ]
}`
```

To open any task pane that's not the default task pane, specify a `sourceLocation` property in the definition of the action. In the following example, a second task pane is opened from a different button.

> ⓘ **Important**
>
> - When a `sourceLocation` is specified for the action, then the task pane does *not* use the shared runtime. It runs in a new separate runtime.
> - No more than one task pane can use the shared runtime, so no more than one action of type `ShowTaskpane` can omit the `sourceLocation` property.

JSON

```json
`{
  "actions": [
    {
      "id": "openChartsTaskpane",
      "type": "ShowTaskpane",
      "title": "Work with Charts",
      "supportPinning": false
    },
    {
      "id": "openTablesTaskpane",
      "type": "ShowTaskpane",
      "title": "Work with Tables",
      "supportPinning": false
      "sourceLocation": "https://MyDomain.com/myPage.html"
    }
  ],
  "tabs": [
    {
      // some tab properties omitted
      "groups": [
        {
          // some group properties omitted
          "controls": [
            {
                "type": "Button",
                "id": "CtxBt112",
                "actionId": "openChartsTaskpane",
                "enabled": false,
                "label": "Open Charts Taskpane",
                // some control properties omitted
            },
            {
                "type": "Button",
                "id": "CtxBt113",
                "actionId": "openTablesTaskpane",
                "enabled": false,
                "label": "Open Tables Taskpane",
                // some control properties omitted
            }
          ]
        }
      ]
    }
  ]
}`
```

Localize the JSON text

The JSON blob that's passed to `requestCreateControls` isn't localized the same way that the manifest markup for custom core tabs is localized (which is described at Control localization from the manifest). Instead, the localization must occur at runtime using distinct JSON blobs for each locale. We suggest that you use a `switch` statement that tests the Office.context.displayLanguage property. The following is an example.

JavaScript

```
function GetContextualTabsJsonSupportedLocale () {
    const displayLanguage = Office.context.displayLanguage;

        switch (displayLanguage) {
            case 'en-US':
                return `{
                    "actions": [
                        // actions omitted
                     ],
                    "tabs": [
                        {
                          "id": "CtxTab1",
                          "label": "Contoso Data",
                          "groups": [
                              // groups omitted
                          ]
                        }
                    ]
                }`;

            case 'fr-FR':
                return `{
                    "actions": [
                        // actions omitted
                    ],
                    "tabs": [
                        {
                          "id": "CtxTab1",
                          "label": "Contoso Données",
                          "groups": [
                              // groups omitted
                          ]
                       }
                    ]
               }`;

            // Other cases omitted
       }
}
```

Then your code calls the function to get the localized blob that's passed to `requestCreateControls`, as in the following example.

JavaScript

```
const contextualTabJSON = GetContextualTabsJsonSupportedLocale();
```

Best practices for custom contextual tabs

Implement an alternate UI experience when custom contextual tabs aren't supported

Some combinations of platform, Office application, and Office build don't support `requestCreateControls`. Your add-in should be designed to provide an alternate experience to users who are running the add-in on one of those combinations. The following sections describe two ways of providing a fallback experience.

Use noncontextual tabs or controls

The add-in's manifest provides a way to create a fallback experience in an add-in that implements custom contextual tabs when the add-in is running on an application or platform that doesn't support custom contextual tabs. The strategy is to define a custom core tab (that is, *noncontextual* custom tab) in the manifest that duplicates the ribbon customizations of the custom contextual tabs in your add-in. Then you use special manifest markup to enable the custom core tab to be visible all the time on platform and version combinations that don't support custom contextual tabs. The process depends on which type of manifest your 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.

Begin by defining a custom core tab (that is, *noncontextual* custom tab) in the manifest that duplicates the ribbon customizations of the custom contextual tabs in your add-in. Then, mark any control groups, or individual controls, or menu items that shouldn't be visible on platforms that support contextual tabs. You mark a group, control, or menu item object by adding an `"overriddenByRibbonApi"` property to it and setting its value to `true`. The effect of doing so is the following:

- If the add-in runs on an application and platform that support custom contextual tabs, then the marked custom groups, controls, and menu items won't appear on the ribbon. Instead, the custom contextual tab will be created when the add-in calls the `requestCreateControls` method.
- If the add-in runs on an application or platform that *doesn't* support `requestCreateControls`, then the groups, controls, and menu items do appear on the custom core tab.

The following is an example. Note that "Contoso.MyButton1" will appear on the custom core tab only when custom contextual tabs aren't supported. However, the parent group (with "ContosoButton2") and the custom core tab will appear regardless of whether custom contextual tabs are supported.

JSON

```
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        "id": "MyTab",
                        "groups": [
                            {
                                ...
                                "controls": [
                                    {
                                        "id": "Contoso.MyButton1",
                                        ...
                                        "overriddenByRibbonApi": true
                                    },
                                    {
                                        "id": "Contoso.MyButton2",
                                        ...
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
    }
]
```

The following is another example. Note that "MyControlGroup" will appear on the custom core tab only when custom contextual tabs aren't supported. However, the parent custom core tab (with unmarked groups) will appear regardless of whether custom contextual tabs are supported.

JSON

```
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        "id": "MyTab",
                        "groups": [
                            {
                                "id": "MyControlGroup",
                                "overriddenByRibbonApi": true
                                ...
                                "controls": [
                                    {
                                        "id": "Contoso.MyButton1",
                                        ...
                                    }
                                ]
                            },
                            ... other groups configured here
                        ]
                    }
                ]
            }
        ]
    }
]
```

When a parent menu control is marked with `"overriddenByRibbonApi": true`, then it isn't visible, and all of its child markup is ignored when custom contextual tabs aren't supported. So, it doesn't matter if any of those child menu items have the `"overriddenByRibbonApi"` property or what its value is. The implication of this is that if a menu item must be visible in all contexts, then not only should it not be marked with `"overriddenByRibbonApi": true`, but *its ancestor menu control must also not be marked this way*. A similar point applies to ribbon controls. If a control must be visible in all contexts, then not only should it not be marked with `"overriddenByRibbonApi": true`, but its parent group must also not be marked this way.

ⓘ Important

Don't mark *all* of the child items of a menu with `"overriddenByRibbonApi": true`. This is pointless if the parent element is marked with `"overriddenByRibbonApi": true` for reasons given in the preceding paragraph. Moreover, if you leave out the `"overriddenByRibbonApi"` property on the parent menu control (or set it to `false`), then the parent will appear regardless of whether custom contextual tabs are supported, but it will be empty when they are supported. So, if all the child elements shouldn't appear when custom contextual tabs are supported, mark the *parent* menu control with `"overriddenByRibbonApi": true`.

A parallel point applies to groups and controls, don't mark all of the controls in a group with `"overriddenByRibbonApi": true`. This is pointless if the parent group is marked with `"overriddenByRibbonApi": true`. Moreover, if you leave out the `"overriddenByRibbonApi"` property on the parent group (or set it to `false`), then the group will appear regardless of whether custom contextual tabs are supported, but it will have no controls in it when they are supported. So, if all the controls shouldn't appear when custom contextual tabs are supported, mark the parent group with `"overriddenByRibbonApi": true`.

Use APIs that show or hide a task pane in specified contexts

As an alternative to using the manifest, your add-in can define a task pane with UI controls that duplicate the functionality of the controls on a custom contextual tab. Then use the Office.addin.showAsTaskpane and Office.addin.hide methods to show the task pane when the contextual tab would have been shown if it was supported. For details on how to use these methods, see Show or hide the task pane of your Office Add-in.

Handle the HostRestartNeeded error

In some scenarios, Office is unable to update the ribbon and will return an error. For example, if the add-in is upgraded and the upgraded add-in has a different set of custom add-in commands, then the Office application must be closed and reopened. Until it is, the `requestUpdate` method will return the error `HostRestartNeeded`. Your code should handle this error. The following is an example of how. In this case, the `reportError` method displays the error to the user.

JavaScript

```javascript
function showDataTab() {
    try {
        Office.ribbon.requestUpdate({
            tabs: [
                {
                    id: "CtxTab1",
                    visible: true
                }
            ]});
    }
    catch(error) {
        if (error.code == "HostRestartNeeded"){
            reportError("Contoso Awesome Add-in has been upgraded. Please save
your work, then close and reopen the Office application.");
        }
    }
}
```

Resources

- Code sample: Create custom contextual tabs on the ribbon
- Community demo of contextual tabs sample
  https://www.youtube-nocookie.com/embed/9tLfm4boQIo

Change the availability of add-in commands

Article • 03/12/2025

When some functionality in your add-in should only be available in certain contexts, you can programmatically configure your custom add-in commands to only be available in these contexts. For example, a function that changes the header of a table should only be available when the cursor is in a table.

> **Note**
>
> - This article assumes that you're familiar with the basic concepts for add-in commands. Please review it if you haven't worked with add-in commands (custom menu items and ribbon buttons) recently.

Supported capabilities

You can programmatically change the availability of an add-in command for the following capabilities.

- Ribbon buttons, menus, and tabs.
- Context menu items.

Office application and requirement set support

The following table outlines the Office applications that support configuring the availability of add-in commands. It also lists the requirement sets needed to use the API.

Expand table

Add-in command capability	Requirement set	Supported Office applications
Ribbon buttons, menus, and tabs	RibbonApi 1.1	• Excel • PowerPoint • Word

Add-in command capability	Requirement set	Supported Office applications
Context menu items	ContextMenuApi 1.1	• Excel • PowerPoint • Word

Tip

To learn how to test for platform support with requirement sets, see Office versions and requirement sets.

Configure a shared runtime

To change the availability of a ribbon or context menu control or item, the manifest of your add-in must first be configured to use a shared runtime. For guidance on how to set up a shared runtime, see Configure your Office Add-in to use a shared runtime.

Programmatically change the availability of an add-in command

Ribbon

Deactivate ribbon controls at launch

Note

Only the controls on the ribbon can be deactivated when the Office application starts. You can't deactivate custom controls added to a context menu at launch.

By default, a custom button or menu item on the ribbon is available for use when the Office application launches. To deactivate it when Office starts, you must specify this in the manifest. The process depends on which type of manifest your add-in uses.

- Unified manifest for Microsoft 365
- Add-in only manifest

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.

Just add an "enabled" property with the value `false` to the control or menu item object. The following shows the basic structure.

JSON

```json
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        "id": "MyTab",
                        "groups": [
                            {
                                ...
                                "controls": [
                                    {
                                        "id": "Contoso.MyButton1",
                                        ...
                                        "enabled": false
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
    }
]
```

Add-in only manifest

Just add an Enabled element immediately *below* (not inside) the Action element of the control item. Then, set its value to `false`.

The following shows the basic structure of a manifest that configures the **<Enabled>** element.

XML

```xml
<OfficeApp ...>
  ...
  <VersionOverrides ...>
    ...
    <Hosts>
      <Host ...>
        ...
        <DesktopFormFactor>
          <ExtensionPoint ...>
            <CustomTab ...>
              ...
              <Group ...>
                ...
                <Control ... id="Contoso.MyButton3">
                  ...
                  <Action ...>
                  <Enabled>false</Enabled>
...
</OfficeApp>
```

Change the availability of a ribbon control

To update the availability of a button or menu item on the ribbon, perform the following steps.

1. Create a RibbonUpdaterData object that specifies the following:
   - The IDs of the command, including its parent group and tab. The IDs must match those declared in the manifest.
   - The availability status of the command.
2. Pass the **RibbonUpdaterData** object to the Office.ribbon.requestUpdate() method.

The following is a simple example. Note that "MyButton", "OfficeAddinTab1", and "CustomGroup111" are copied from the manifest.

JavaScript

```javascript
function enableButton() {
    const ribbonUpdaterData = {
        tabs: [
            {
                id: "OfficeAppTab1",
                groups: [
                    {
                      id: "CustomGroup111",
                      controls: [
                        {
                            id: "MyButton",
                            enabled: true
                        }
                      ]
                    }
                ]
            }
        ]
    };

    Office.ribbon.requestUpdate(ribbonUpdaterData);
}
```

There are several interfaces (types) to make it easier to construct the **RibbonUpdateData** object.

- Office.Control
- Office.Group
- Office.Tab

The following is the equivalent example in TypeScript and it makes use of these types.

TypeScript

```typescript
const enableButton = async () => {
    const button: Control = { id: "MyButton", enabled: true };
    const parentGroup: Group = { id: "CustomGroup111", controls:
[button] };
    const parentTab: Tab = { id: "OfficeAddinTab1", groups:
[parentGroup] };
    const ribbonUpdater: RibbonUpdaterData = { tabs: [parentTab] };
    Office.ribbon.requestUpdate(ribbonUpdater);
}
```

Tip

You can `await` the call of **requestUpdate()** if the parent function is asynchronous, but note that the Office application controls when it updates the state of the ribbon. The **requestUpdate()** method queues a request to update. The method will resolve the promise object as soon as it has queued the request, not when the ribbon actually updates.

Toggle tab visibility and the enabled status of a button at the same time

The **requestUpdate** method is also used to toggle the visibility of a custom contextual tab. For details about this and example code, see Create custom contextual tabs in Office Add-ins.

Change the state in response to an event

A common scenario in which the state of a ribbon or context menu control should change is when a user-initiated event changes the add-in context. Consider a scenario in which a button should be available when, and only when, a chart is activated. Although the following example uses ribbon controls, a similar implementation can be applied to custom items on a context menu.

1. First, set the <**Enabled**> element for the button in the manifest to `false`. For guidance on how to configure this, see Deactivate ribbon controls at launch.

2. Then, assign handlers. This is commonly done in the **Office.onReady** function as in the following example. In the example, handlers (created in a later step) are assigned to the **onActivated** and **onDeactivated** events of all the charts in an Excel worksheet.

   JavaScript

   ```javascript
   Office.onReady(async () => {
       await Excel.run((context) => {
           const charts = context.workbook.worksheets
               .getActiveWorksheet()
               .charts;
           charts.onActivated.add(enableChartFormat);
           charts.onDeactivated.add(disableChartFormat);
           return context.sync();
       });
   });
   ```

3. Define the `enableChartFormat` handler. The following is a simple example. For a more robust way of changing a control's status, see Best practice: Test for control status errors.

JavaScript

```javascript
function enableChartFormat() {
    const button =
        {
            id: "ChartFormatButton",
            enabled: true
        };
    const parentGroup =
        {
            id: "MyGroup",
            controls: [button]
        };
    const parentTab =
        {
            id: "CustomChartTab",
            groups: [parentGroup]
        };
    const ribbonUpdater = { tabs: [parentTab] };
    Office.ribbon.requestUpdate(ribbonUpdater);
}
```

4. Define the `disableChartFormat` handler. It's identical to the `enableChartFormat` handler, except that the **enabled** property of the button object is set to `false`.

Best practice: Test for control status errors

In some circumstances, the ribbon or context menu doesn't repaint after `requestUpdate` is called, so the control's clickable status doesn't change. For this reason it's a best practice for the add-in to keep track of the status of its controls. The add-in should conform to the following rules.

- Whenever `requestUpdate` is called, the code should record the intended state of the custom buttons and menu items.
- When a custom control is selected, the first code in the handler should check to see if the button should have been available. If it shouldn't have been available, the code should report or log an error and try again to set the buttons to the intended state.

The following example shows a function that deactivates a button on the ribbon and records the button's status. In this example, `chartFormatButtonEnabled` is a global boolean variable that's initialized to the same value as the Enabled element for the button in the add-in's manifest. Although the example uses a ribbon button, a similar implementation can be applied to custom items on a context menu.

JavaScript

```javascript
function disableChartFormat() {
    const button =
    {
        id: "ChartFormatButton",
        enabled: false
    };
    const parentGroup =
    {
        id: "MyGroup",
        controls: [button]
    };
    const parentTab =
    {
        id: "CustomChartTab",
        groups: [parentGroup]
    };
    const ribbonUpdater = { tabs: [parentTab] };
    Office.ribbon.requestUpdate(ribbonUpdater);

    chartFormatButtonEnabled = false;
}
```

The following example shows how the button's handler tests for an incorrect state of the button. Note that `reportError` is a function that shows or logs an error.

JavaScript

```javascript
function chartFormatButtonHandler() {
    if (chartFormatButtonEnabled) {

        // Do work here.

    } else {
        // Report the error and try to make the button unavailable again.
        reportError("That action is not possible at this time.");
        disableChartFormat();
    }
}
```

Error handling

In some scenarios, Office is unable to update the ribbon or context menu and will return an error. For example, if the add-in is upgraded and the upgraded add-in has a different set of custom add-in commands, then the Office application must be closed and reopened. Until it is, the `requestUpdate` method will return the error `HostRestartNeeded`. The following is an example of how to handle this error. In this case, the `reportError` method displays the error to the user. Although the example uses a ribbon button, a similar implementation can be applied to custom items on a context menu.

```JavaScript
function disableChartFormat() {
    try {
        const button =
        {
            id: "ChartFormatButton",
            enabled: false
        };
        const parentGroup =
        {
            id: "MyGroup",
            controls: [button]
        };
        const parentTab =
        {
            id: "CustomChartTab",
            groups: [parentGroup]
        };
        const ribbonUpdater = { tabs: [parentTab] };
        Office.ribbon.requestUpdate(ribbonUpdater);

        chartFormatButtonEnabled = false;
    }
    catch(error) {
        if (error.code == "HostRestartNeeded"){
            reportError("Contoso Awesome Add-in has been upgraded. Please
save your work, close the Office application, and restart it.");
        }
    }
}
```

See also

- Add-in commands
- Create add-in commands with the add-in only manifest
- Create custom contextual tabs in Office Add-ins

Integrate built-in Office buttons into custom control groups and tabs

06/10/2025

You can insert built-in Office buttons into your custom control groups on the Office ribbon by using markup in the add-in's manifest. (You can't insert your custom add-in commands into a built-in Office group.) You can also insert entire built-in Office control groups into your custom ribbon tabs.

ⓘ Note

This article assumes that you're familiar with the article Basic concepts for add-in commands. Please review it if you haven't done so recently.

ⓘ Important

The add-in feature described in this article is only available in **PowerPoint** on the web, on Windows, and on Mac.

Open the tab for the type of manifest your add-in uses for the details of the manifest markup.

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.

Insert a built-in control group into a custom tab

To insert a built-in Office control group into a custom tab, add a group object with a "builtInGroupId" property *instead of an "id" property* to the "groups" array of your custom tab object. Set to the ID of the built-in group. See Find the IDs of controls and control groups. *The built-in group object should have no other properties.*

The following example adds the Office Paragraph control group to a custom tab.

JSON

```
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        "id": "MyTab",
                        ...
                        "groups": [
                            ... // Optionally, other groups in the tab.
                            {
                                "builtInGroupId": "Paragraph"
                            },
                            ... // Optionally, other groups in the tab.
                        ]
                    }
                ]
            }
        ]
    }
]
```

Insert a built-in control into a custom group

To insert a built-in Office control into a custom group, add a control object with a `"builtInControlId"` property *instead of an* `"id"` *property* to the "controls" array of your custom group object. Set to the ID of the built-in control. See Find the IDs of controls and control groups. *The built-in control object should have no other properties.*

The following example adds the Office Superscript control to a custom group.

JSON

```
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        ...
                        "groups": [
                                    {
                                        "id": "MyGroup",
                                        ...
                                        "controls": [
                                            ... // Optionally, other controls in the
group.
                                            {
                                                "builtInControlId": "Superscript"
                                            },
                                            ... // Optionally, other controls in the
group.
                                        ]
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
```

Note

Users can customize the ribbon in the Office application. Any user customizations will override your manifest settings. For example, a user can remove a button from any group and remove any group from a tab.

Find the IDs of controls and control groups

The IDs for supported controls and control groups are in files in the repo Office Control IDs. Follow the instructions in the ReadMe file of that repo.

Behavior on unsupported platforms

If your add-in is installed on a platform that doesn't support requirement set AddinCommands 1.3, then the markup described in this article is ignored and the built-in Office controls/groups won't appear in your custom groups/tabs. To prevent your add-in from being installed on platforms that don't support the markup, you must specify **AddinCommands 1.3** in the manifest as a requirement for installation. For instructions, see Specify which Office versions and platforms can host your add-in. Alternatively, design your add-in to have an experience when **AddinCommands 1.3** isn't supported, as described in Design for alternate experiences. For example, if your add-in contains instructions that assume the built-in buttons are in your custom groups, you could design a version that assumes that the built-in buttons are only in their usual places.

Position a custom tab on the ribbon

Article • 02/12/2025

You can specify where you want your add-in's custom tab to appear on the Office application's ribbon by using markup in the add-in's manifest.

ⓘ **Note**

This article assumes that you're familiar with the article Basic concepts for add-in commands. Please review it if you haven't done so recently.

ⓘ **Important**

- The add-in feature and markup described in this article is *only available in PowerPoint on the web*.
- The markup described in this article only works on platforms that support requirement set **AddinCommands 1.3**. See Behavior on unsupported platforms below.

By default, a custom tab is added to the end of the ribbon. However, you can specify where you want a custom tab to appear by identifying which built-in Office tab you want it to be next to and specifying whether it should be on the left or right side of the built-in tab. Open the tab for the type of manifest your add-in uses for the details of the manifest markup.

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 position your custom tab, include a "position" property in the "extensions.ribbons.tabs" object. Set the "position.builtInTabId" property to the ID of the built-in tab that you want your custom tab to be next to. (See Find the IDs of built-in Office ribbon tabs.) Set the "position.align" property to either "before" (left) or "after" (right).

In the following example, the custom tab is configured to appear *just after* the **Review** tab.

JSON

```
"extensions": [
    ...
    {
        ...
        "ribbons": [
            ...
            {
                ...
                "tabs": [
                    {
                        "id": "MyTab",
                        ...
                        "position": {
                            "builtInTabId": "TabReview",
                            "align": "after"
                        }
                    }
                ]
            }
        ]
    }
]
```

How user actions can affect custom tab positioning

- If the user installs more than one add-in whose custom tab is configured for the same place, say after the **Review** tab, then the tab for the most recently installed add-in will be located in that place. The tabs of the previously installed add-ins will be moved over one place. For example, the user installs add-ins A, B, and C in that order and all are configured to insert a tab after the **Review** tab, then the tabs will appear in this order: **Review**, **AddinCTab**, **AddinBTab**, **AddinATab**.
- Users can customize the ribbon in the Office application. For example, a user can move or hide your add-in's tab. You cannot prevent this or detect that it has happened.
- If a user moves one of the built-in tabs, then Office interprets the positioning markup in the manifest in terms of *the default location of the built-in tab*. For example, if the user moves the **Review** tab to the right end of the ribbon, Office will interpret the markup in the previous example as meaning "put the custom tab just to the right of *where the **Review** tab would be by default*."

Specify which tab has focus when the document opens

Office always gives default focus to the tab that's immediately to the right of the **File** tab. By default this is the **Home** tab. If you configure your custom tab to be before the **Home** tab, then your custom tab will have focus when the document opens.

> **Important**
>
> Giving excessive prominence to your add-in inconveniences and annoys users and administrators. Don't position a custom tab before the **Home** tab unless your add-in is the primary way users will interact with the document.

Behavior on unsupported platforms

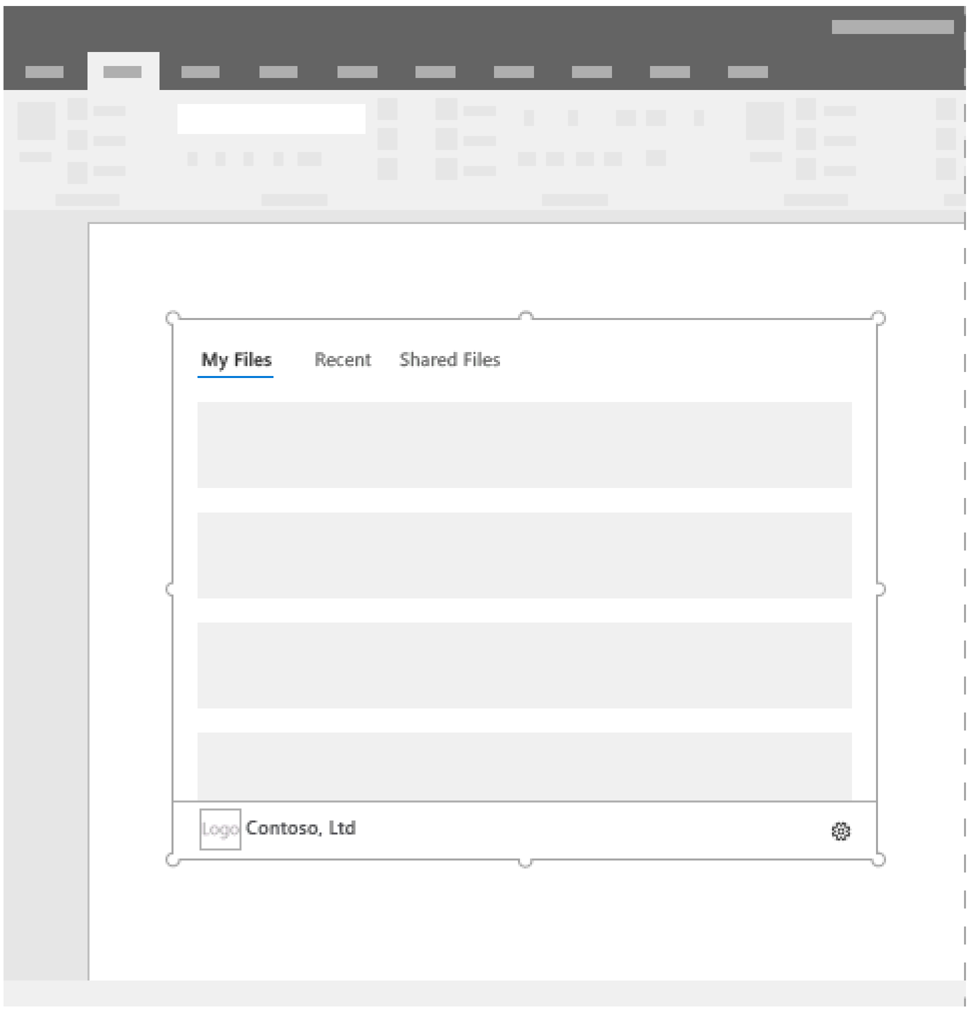
If your add-in is installed on a platform that doesn't support requirement set AddinCommands 1.3, then the markup described in this article is ignored and your custom tab will appear as the rightmost tab on the ribbon. To prevent your add-in from being installed on platforms that don't support the markup, you must specify **AddinCommands 1.3** in the manifest as a requirement for installation. For instructions, see Specify which Office versions and platforms can host your add-in. Alternatively, design your add-in to have an alternate experience when **AddinCommands 1.3** isn't supported, as described in Design for alternate experiences. For example, if your add-in contains instructions that assume the custom tab is where you want it, you could have an alternate version that assumes the tab is the rightmost.

Content Office Add-ins

Article • 02/12/2025

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

Figure 1. Typical layout for content add-ins



Best practices

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

Variants

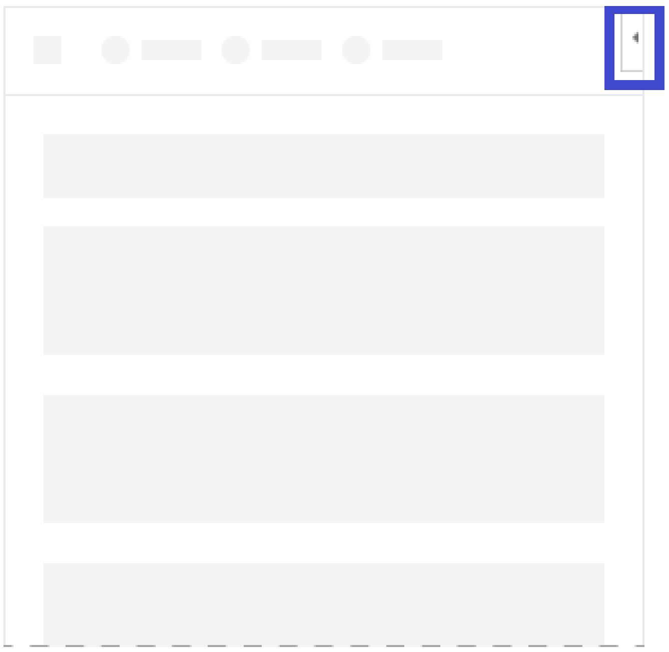
Content add-in sizes for Excel and PowerPoint in Office desktop and in a web browser are user specified.

Personality menu

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

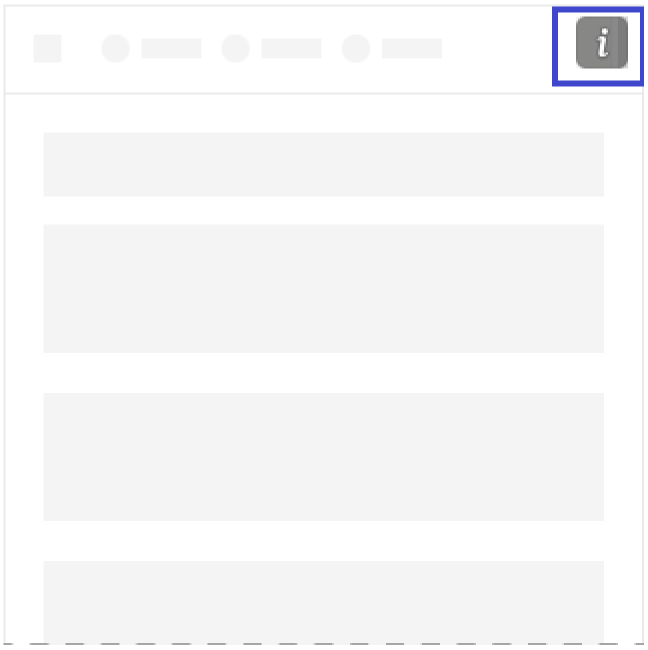
For Windows, the personality menu measures 12x32 pixels, as shown.

Figure 2. Personality menu on Windows



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

Figure 3. Personality menu on Mac



Implementation

There are minor differences in the manifests between content add-ins and add-ins that use task panes. Open the tab for the type of manifest you're using.

Unified manifest for Microsoft 365

ⓘ **Note**

The unified manifest is available in Excel, PowerPoint, and Word as a developer preview. For Outlook, it's generally available and can be used in production add-ins.

Configure the manifest with the following steps.

1. Add a "contentRuntimes" child array to the extension object in the "extensions" array.
2. Remove the "runtimes" property if it is present. The "runtimes" array is for task pane or mail add-ins. These cannot be combined with a content add-in.
3. Add an anonymous content runtime object in the "contentRuntimes" array.
4. Set the "id" property of the object to a descriptive name.
5. Set the "code.page" object to the full URL of the custom content that you want to embed in the document.
6. Optionally, set the "requestedWidth" and "requestedHeight" properties to a size between 32 and 1000 pixels. If these properties aren't used, the Office application determines the size.

7. Optionally, set the "disableSnapshot" property to `true` to prevent Office from saving a snapshot of the content component with the document.

The following is an example of a "contentRuntimes" property.

JSON

```json
"contentRuntimes": [
    {
        "id": "ContentRuntime",
        "code": {
            "page": "https://localhost:3000/content.html"
        },
        "requestedWidth": 100,
        "requestedHeight": 100,
        "disableSnapshot": true,
    }
]
```

For a sample that implements a content add-in, see Excel Content Add-in Humongous Insurance on GitHub.

To create your own content add-in, see the Excel content add-in quick start and PowerPoint content add-in quick start.

Support considerations

- Check to see if your Office Add-in will work on a specific Office application or platform.
- Some content add-ins may require the user to "trust" the add-in to read and write to Excel or PowerPoint. You can declare what level of permissions you want your user to have in the add-in's manifest.

See also

- Office client application and platform availability for Office Add-ins
- Fabric Core in Office Add-ins
- UX design patterns for Office Add-ins
- Requesting permissions for API use in add-ins