Best practices and rules for the Office Dialog API

06/17/2025

This article provides rules, limitations, and best practices for the Office Dialog API, including best practices for designing the UI of a dialog and using the API within a single-page application (SPA).

> **Note**
>
> To familiarize yourself with the basics of using the Office Dialog API, see Use the Office Dialog API in your Office Add-ins.
>
> See also Handling errors and events with the Office dialog box.

Rules and limitations

- A dialog box can only navigate to HTTPS URLs, not HTTP.
- The URL passed to the displayDialogAsync method must be in the exact same domain as the add-in itself. It can't be a subdomain. However, the page that is passed to it can redirect to a page in another domain.
- A host page can have only one dialog box open at a time. The host page could be either a task pane or the function file of a function command. Multiple dialogs can be open at the same time from custom ribbon buttons or menu items.
- Only two Office APIs can be called in the dialog box,
  - Office.context.ui.messageParent
  - `Office.context.requirements.isSetSupported` (For more information, see Specify Office applications and API requirements.)
- The messageParent function should usually be called from a page in the exact same domain as the add-in itself, but this isn't mandatory. For more information, see Cross-domain messaging to the host runtime.
- When a dialog box opens, it's centered on the screen on top of the Office application.
- A dialog box can be moved and resized by the user.
- A dialog box appears in the order in which it was created.

> **Tip**
>
> In Office on the web and new Outlook on Windows, if the domain of your dialog is different from that of your add-in and it enforces the Cross-Origin-Opener-Policy: same-origin response header, your add-in will be blocked from accessing messages from the dialog and your users will be shown error 12006. To prevent this, you must set the header to `Cross-Origin-Opener-Policy: unsafe-none` or configure your add-in and dialog to be in the same domain.

- In Outlook on the web and new Outlook on Windows, don't set the window.name property when configuring a dialog in your add-in. The `window.name` property is used by these Outlook clients to maintain functionality across page redirects.

Best practices

Avoid overusing dialog boxes

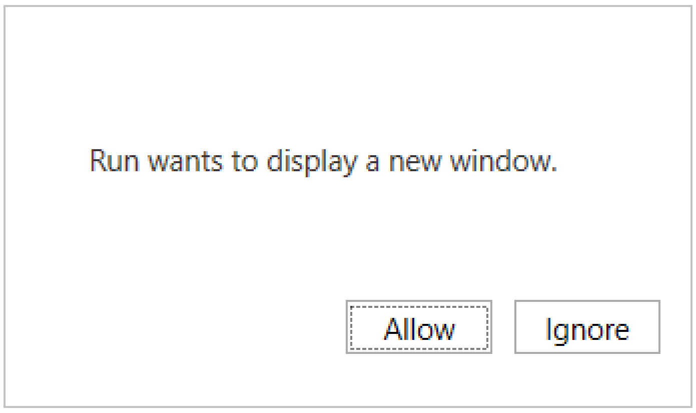
Because overlapping UI elements are discouraged, avoid opening a dialog box from a task pane unless your scenario requires it. When you consider how to use the surface area of a task pane, note that task panes can be tabbed. For an example of a tabbed task pane, see the Excel Add-in JavaScript SalesTracker sample.

Design a dialog box UI

For best practices in dialog box design, see Dialog boxes in Office Add-ins.

Handle pop-up blockers with Office on the web

Attempting to display a dialog box while using Office on the web may cause the browser's pop-up blocker to block the dialog box. To prevent this, Office on the web prompts the user to **Allow** or **Ignore** opening the dialog.


Run wants to display a new window.

If the user chooses **Allow**, the Office dialog box opens. If the user chooses **Ignore**, the prompt closes and the Office dialog box does not open. Instead, the `displayDialogAsync` method returns error 12009. Your code should catch this error and either provide an alternate experience that doesn't require a dialog, or display a message to the user advising that the add-in requires them to allow the dialog. (For more about 12009, see Errors from displayDialogAsync.)

If, for any reason, you want to turn off this feature, then your code must opt out. It makes this request with the DialogOptions object that is passed to the `displayDialogAsync` method. Specifically, the object should include `promptBeforeOpen: false`. When this option is set to false, Office on the web won't prompt the user to allow the add-in to open a dialog, and the Office dialog won't open.

Request access to device capabilities in Office on the web and new Outlook on Windows

If your add-in requires access to a user's device capabilities, a dialog to request for permissions is available through the device permission API. Device capabilities include a user's camera, geolocation, and microphone. This applies to the following Office applications.

- Office on the web (Excel, Outlook, PowerPoint, and Word) running in Chromium-based browsers, such as Microsoft Edge or Google Chrome
- new Outlook on Windows

When your add-in calls Office.context.devicePermission.requestPermissions or Office.context.devicePermission.requestPermissionsAsync, a dialog is shown with the requested device capabilities and the options to **Allow**, **Allow once**, or **Deny** access. To learn more, see View, manage, and install add-ins for Excel, PowerPoint, and Word.

> **Note**

- Add-ins that run in Office desktop clients or in browsers not based on Chromium automatically show a dialog requesting for a user's permission. The developer doesn't need to implement the device permission API on these platforms.
- Add-ins that run in Safari are blocked from accessing a user's device capabilities. The device permission API isn't supported in Safari.
- Access to a user's geolocation is only supported in **Outlook on the web** and new **Outlook on Windows**.

Don't use the _host_info value

Office automatically adds a query parameter called `_host_info` to the URL that is passed to `displayDialogAsync`. It is appended after your custom query parameters, if any. It isn't appended to any subsequent URLs that the dialog box navigates to. Microsoft may change the content of this value, or remove it entirely, so your code shouldn't read it. The same value is added to the dialog box's session storage (that is, the Window.sessionStorage property). Again, *your code should neither read nor write to this value*.

Open another dialog immediately after closing one

You can't have more than one dialog open from a given host page, so your code should call Dialog.close on an open dialog before it calls `displayDialogAsync` to open another dialog. The `close` method is asynchronous. For this reason, if you call `displayDialogAsync` immediately after a call of `close`, the first dialog may not have completely closed when Office attempts to open the second. If that happens, Office will return a 12007 error: "The operation failed because this add-in already has an active dialog."

The `close` method doesn't accept a callback parameter, and it doesn't return a Promise object so it cannot be awaited with either the `await` keyword or with a `then` method. For this reason, we suggest the following technique when you need to open a new dialog immediately after closing a dialog: encapsulate the code to open the new dialog in a function and design the function to recursively call itself if the call of `displayDialogAsync` returns `12007`. The following is an example.

JavaScript

```javascript
function openFirstDialog() {
  Office.context.ui.displayDialogAsync(
    "https://MyDomain/firstDialog.html",
    { width: 50, height: 50 },
    (result) => {
      if (result.status === Office.AsyncResultStatus.Succeeded) {
        const dialog = result.value;
        dialog.close();
        openSecondDialog();
      }
      else {
         // Handle errors.
      }
    }
  );
}

function openSecondDialog() {
  Office.context.ui.displayDialogAsync(
    "https://MyDomain/secondDialog.html",
    { width: 50, height: 50 },
    (result) => {
      if (result.status === Office.AsyncResultStatus.Failed) {
        if (result.error.code === 12007) {
          openSecondDialog(); // Recursive call.
        }
        else {
         // Handle other errors.
        }
      }
    }
  );
}
```

Alternatively, you could force the code to pause before it tries to open the second dialog by using the setTimeout method. The following is an example.

JavaScript

```
function openFirstDialog() {
    Office.context.ui.displayDialogAsync(
      "https://MyDomain/firstDialog.html",
      { width: 50, height: 50 },
      (result) => {
        if (result.status === Office.AsyncResultStatus.Succeeded) {
          const dialog = result.value;
          dialog.close();
          setTimeout(() => {
            Office.context.ui.displayDialogAsync(
              "https://MyDomain/secondDialog.html",
              { width: 50, height: 50 },
              (result) => {
                // Callback body.
              }
            );
          }, 1000);
        }
        else {
           // Handle errors.
```

```
            }
        }
    );
}
```

Best practices for using the Office Dialog API in an SPA

If your add-in uses client-side routing, as single-page applications (SPAs) typically do, you have the option to pass the URL of a route to the displayDialogAsync method instead of the URL of a separate HTML page. *We recommend against doing so for the reasons given below.*

> **Note**
>
> This article isn't relevant to *server-side* routing, such as in an Express-based web application.

Problems with SPAs and the Office Dialog API

The Office dialog box is in a new window with its own instance of the JavaScript engine, and hence it's own complete execution context. If you pass a route, your base page and all its initialization and bootstrapping code run again in this new context, and any variables are set to their initial values in the dialog box. So this technique downloads and launches a second instance of your application in the box window, which partially defeats the purpose of an SPA. In addition, code that changes variables in the dialog box window doesn't change the task pane version of the same variables. Similarly, the dialog box window has its own session storage (the Window.sessionStorage property), which isn't accessible from code in the task pane. The dialog box and the host page on which `displayDialogAsync` was called look like two different clients to your server. (For a reminder of what a host page is, see Open a dialog box from a host page.)

So, if you passed a route to the `displayDialogAsync` method, you wouldn't really have an SPA; you'd have *two instances of the same SPA*. Moreover, much of the code in the task pane instance would never be used in that instance and much of the code in the dialog box instance would never be used in that instance. It would be like having two SPAs in the same bundle.

Microsoft recommendations

Instead of passing a client-side route to the `displayDialogAsync` method, we recommend that you do one of the following:

- If the code that you want to run in the dialog box is sufficiently complex, create two different SPAs explicitly; that is, have two SPAs in different folders of the same domain. One SPA runs in the dialog box and the other in the dialog box's host page where `displayDialogAsync` was called.
- In most scenarios, only simple logic is needed in the dialog box. In such cases, your project will be greatly simplified by hosting a single HTML page, with embedded or referenced JavaScript, in the domain of your SPA. Pass the URL of the page to the `displayDialogAsync` method. While this means that you are deviating from the literal idea of a single-page app; you don't really have a single instance of an SPA when you are using the Office Dialog API.

See also

- Use the Office Dialog API in Office Add-ins
- Handle errors and events in the Office dialog box

Alternative ways of passing messages to a dialog box from its host page

Article • 12/01/2023

The recommended way to pass data and messages from a parent page to a child dialog is with the `messageChild` method as described in Use the Office dialog API in your Office Add-ins. If your add-in is running on a platform or host that doesn't support the DialogApi 1.2 requirement set, there are two other ways that you can pass information to the dialog.

- Store the information somewhere accessible to both the host window and dialog. The two windows don't share a common session storage (the Window.sessionStorage property), but *if they have the same domain* (including port number, if any), they share a common local storage.

  > **Note**
  >
  > Changes to browser security will affect your strategy for token handling.
  >
  > - If your add-in runs in **Office on the web** in the Microsoft Edge Legacy (non-Chromium) or Safari browser, the dialog and task pane don't share the same **local storage**, so it can't be used to communicate between them.
  > - Starting in Version 115 of Chromium-based browsers, such as Chrome and Edge, **storage partitioning** is being tested to prevent specific side-channel cross-site tracking (see also **Microsoft Edge browser policies**). This means that data stored by storage APIs, such as local storage, are only available to contexts with the same origin and the same top-level site. To work around this, in your browser, go to **chrome://flags** or **edge://flags**, then set the **Experimental third-party storage partitioning (#third-party-storage-partitioning)** flag to **Disabled**. Where possible, we recommend to pass data between the dialog and task pane using the **messageParent** and **messageChild** methods as described in **Use the Office dialog API in your Office Add-ins**.

- Add query parameters to the URL that is passed to `displayDialogAsync`.

Use local storage

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

JavaScript

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

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

JavaScript

```javascript
const clientID = localStorage.getItem("clientID");
// You can also use property syntax:
// const clientID = localStorage.clientID;
```

Use query parameters

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

JavaScript

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

For a sample that uses this technique, see Insert Excel charts using Microsoft Graph in a PowerPoint add-in.

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

Important

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

Handle errors and events in the Office dialog box

Article • 03/12/2025

This article describes how to trap and handle errors when opening the dialog box and errors that happen inside the dialog box.

> **Note**
>
> This article presupposes that you're familiar with the basics of using the Office dialog API as described in Use the Office dialog API in your Office Add-ins.
>
> See also Best practices and rules for the Office dialog API.

Your code should handle two categories of events.

- Errors returned by the call of `displayDialogAsync` because the dialog box can't be created.
- Errors, and other events, in the dialog box.

Errors from displayDialogAsync

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

Expand table

Code number	Meaning
12004	The domain of the URL passed to `displayDialogAsync` isn't trusted. The domain must be the same domain as the host page (including protocol and port number). In Outlook on the web and the new Outlook on Windows, this error occurs when an add-in is hosted on a localhost server and its manifest doesn't specify an AppDomain element for localhost.
12005	The URL passed to `displayDialogAsync` uses the HTTP protocol. HTTPS is required. (In some versions of Office, the error message text returned with 12005 is the same one returned for 12004.)

Code number	Meaning
12007	A dialog box is already opened from this host window. A host window, such as a task pane, can only have one dialog box open at a time.
12009	The user chose to ignore the dialog box. This error can occur in Office on the web, where users may choose not to allow an add-in to present a dialog box. For more information, see Handling pop-up blockers with Office on the web.
12011	The add-in is running in Office on the web and the user's browser configuration is blocking popups. This most commonly happens when the browser is Edge Legacy and the domain of the add-in is in different security zone from the domain that the dialog is trying to open. Another scenario which triggers this error is that the browser is Safari and it's configured to block all popups. Consider responding to this error with a prompt to the user to change their browser configuration or use a different browser.

When `displayDialogAsync` is called, it passes an AsyncResult object to its callback function. When the call is successful, the dialog box is opened, and the `value` property of the `AsyncResult` object is a Dialog object. For an example of this, see Send information from the dialog box to the host page. When the call to `displayDialogAsync` fails, the dialog box isn't created, the `status` property of the `AsyncResult` object is set to `Office.AsyncResultStatus.Failed`, and the `error` property of the object is populated. You should always provide a callback that tests the `status` and responds when it's an error. For an example that reports the error message regardless of its code number, see the following code. (The `showNotification` function, not defined in this article, either displays or logs the error. For an example of how you can implement this function within your add-in, see Office Add-in Dialog API Example.)

JavaScript

```javascript
let dialog;
Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html',
function (asyncResult) {
    if (asyncResult.status === Office.AsyncResultStatus.Failed) {
        showNotification(asyncResult.error.code = ": " +
asyncResult.error.message);
    } else {
        dialog = asyncResult.value;
        dialog.addEventHandler(Office.EventType.DialogMessageReceived,
processMessage);
    }
});
```

Errors and events in the dialog box

Three errors and events in the dialog box will raise a `DialogEventReceived` event in the host page. For a reminder of what a host page is, see Open a dialog box from a host page.

Expand table

Code number	Meaning
12002	One of the following: • No page exists at the URL that was passed to `displayDialogAsync`. • The page that was passed to `displayDialogAsync` loaded, but the dialog box was then redirected to a page that it can't find or load, or it has been directed to a URL with invalid syntax.
12003	The dialog box was directed to a URL with the HTTP protocol. HTTPS is required.
12006	One of the following: • The dialog box was closed, usually because the user chose the **Close** button **X**. • The dialog returned a Cross-Origin-Opener-Policy: same-origin response header. To prevent this, you must set the header to `Cross-Origin-Opener-Policy: unsafe-none` or configure your add-in and dialog to be in the same domain as the host page.

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

JavaScript

```javascript
let dialog;
Office.context.ui.displayDialogAsync('https://myDomain/myDialog.html',
    function (result) {
        dialog = result.value;
        dialog.addEventHandler(Office.EventType.DialogEventReceived,
processDialogEvent);
    }
);
```

For an example of a handler for the `DialogEventReceived` event that creates custom error messages for each error code, see the following example.

JavaScript

```javascript
function processDialogEvent(arg) {
    switch (arg.error) {
        case 12002:
            showNotification("The dialog box has been directed to a page
that it can't find or load, or the URL syntax is invalid.");
            break;
        case 12003:
            showNotification("The dialog box has been directed to a URL with
the HTTP protocol. HTTPS is required.");            break;
        case 12006:
            showNotification("Dialog closed.");
            break;
        default:
            showNotification("Unknown error in dialog box.");
            break;
    }
}
```

See also

For a sample add-in that handles errors in this way, see Office Add-in Dialog API Example.

Use the Office dialog box to show a video

Article • 04/04/2023

This article explains how to play a video in an Office Add-in dialog box.

> **Note**
>
> This article presumes you're familiar with the basics of using the Office dialog box as described in **Use the Office dialog API in your Office Add-ins**.

To play a video in a dialog box with the Office dialog API, follow these steps.

1. Create a page containing an iframe and no other content. The page must be in the same domain as the host page. For a reminder of what a host page is, see Open a dialog box from a host page. In the `src` attribute of the iframe, point to the URL of an online video. The protocol of the video's URL must be HTTPS. In this article, we'll call this page "video.dialogbox.html". The following is an example of the markup.

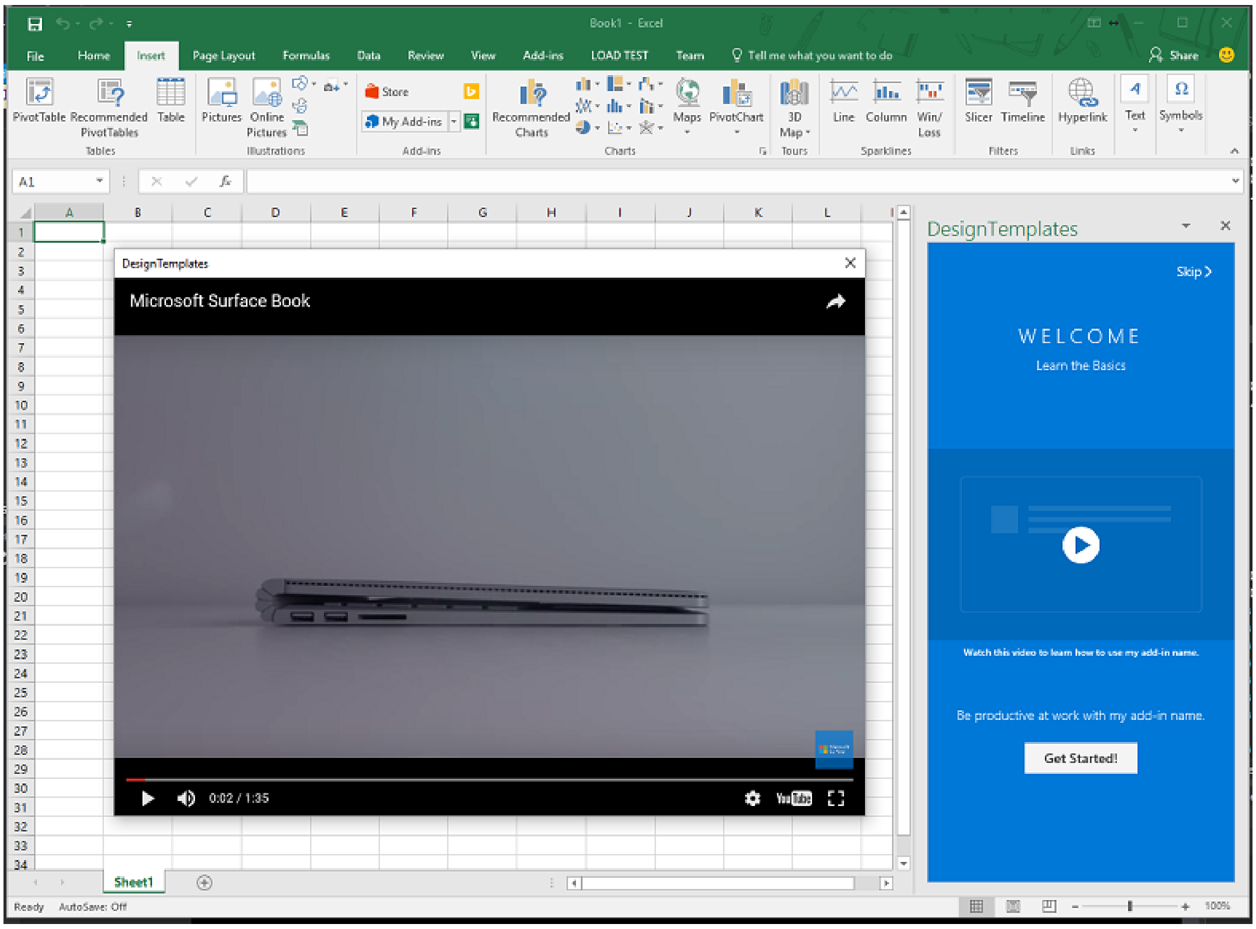
   HTML

   ```html
   <iframe class="ms-firstrun-video__player"  width="640" height="360"
       src="https://www.youtube.com/embed/XVfOe5mFbAE?rel=0&autoplay=1"
       frameborder="0" allowfullscreen>
   </iframe>
   ```

2. Use a call of `displayDialogAsync` in the host page to open video.dialogbox.html.

3. If your add-in needs to know when the user closes the dialog box, register a handler for the `DialogEventReceived` event and handle the 12006 event. For details, see Errors and events in the Office dialog box.

For a sample of a video playing in a dialog box, see the video placemat design pattern.



Watch this video to learn how to use my add-in name.

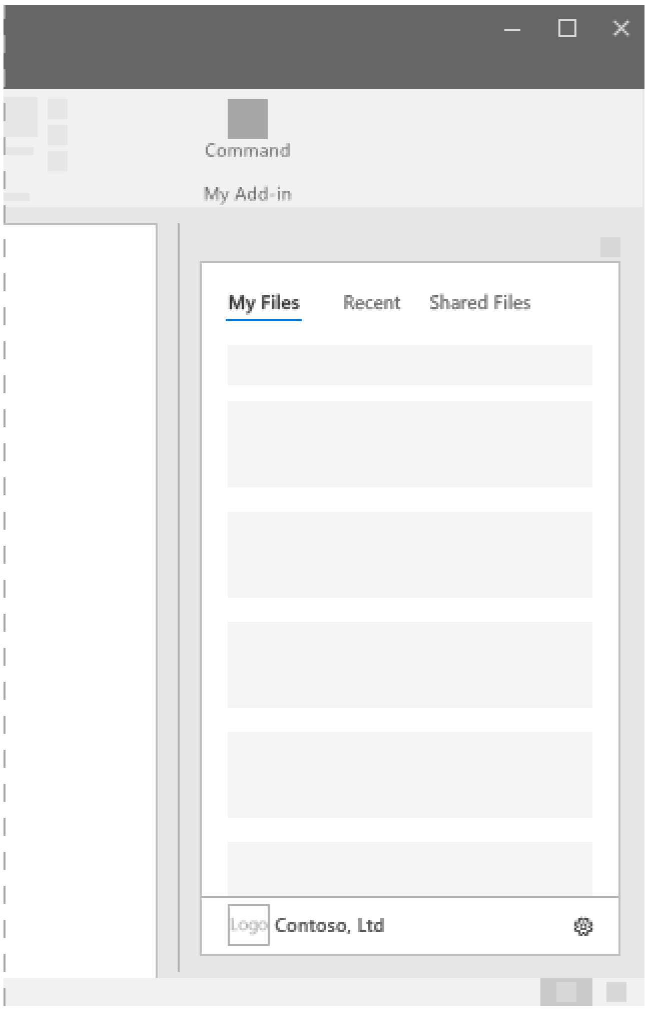
Be productive at work with my add-in name.

Task panes in Office Add-ins

Article • 08/18/2023

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

Figure 1. Typical task pane layout



Best practices

Expand table

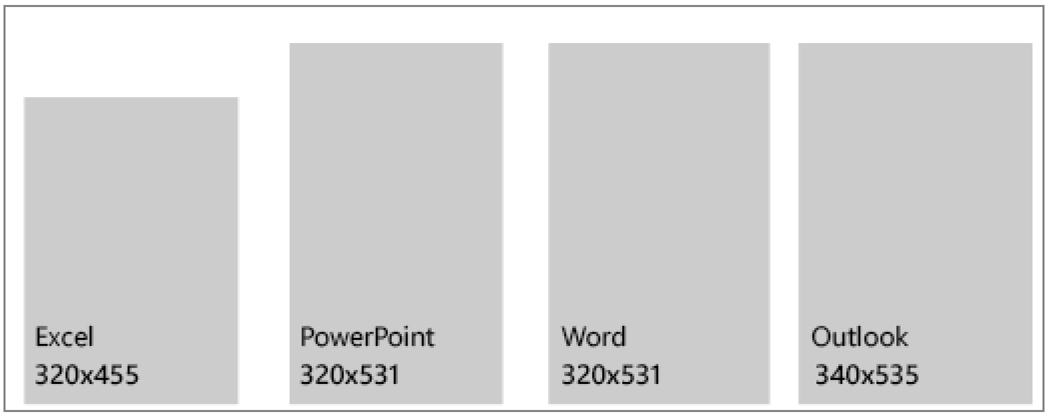
Do	Don't
Include the name of your add-in in the title.	Don't append your company name to the title.

Do	**Don't**
Use short descriptive names in the title.	Don't append strings such as "add-in," "for Word," or "for Office" to the title of your add-in.
Include some navigational or commanding element such as the CommandBar or Pivot at the top of your add-in.	*None*
Include a branding element such as the BrandBar at the bottom of your add-in unless your add-in is to be used within Outlook.	*None*

Variants

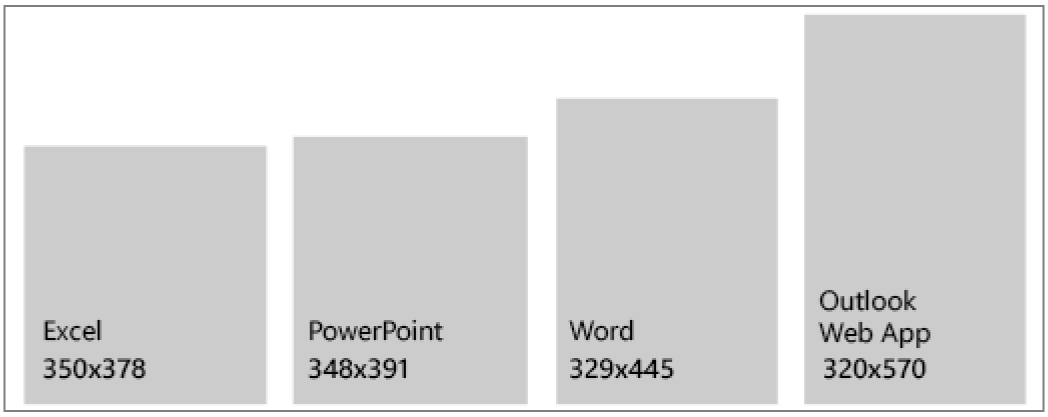
The following images show the various task pane sizes with the Office app ribbon at a 1366x768 resolution. For Excel, additional vertical space is required to accommodate the formula bar.

Figure 2. Office 2016 desktop task pane sizes



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

Figure 3. Office task pane sizes



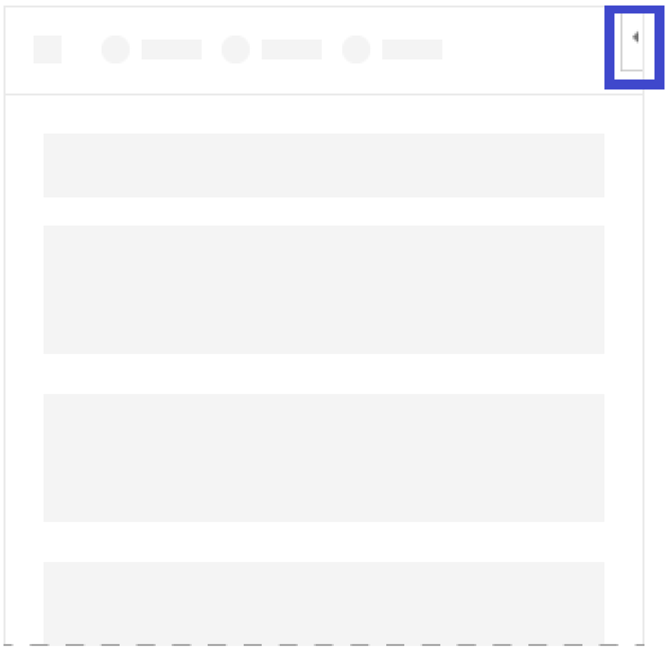
- Excel - 350x378 pixels
- PowerPoint - 348x391 pixels
- Word - 329x445 pixels
- Outlook (on the web) - 320x570 pixels

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. (The personality menu isn't supported in Outlook.)

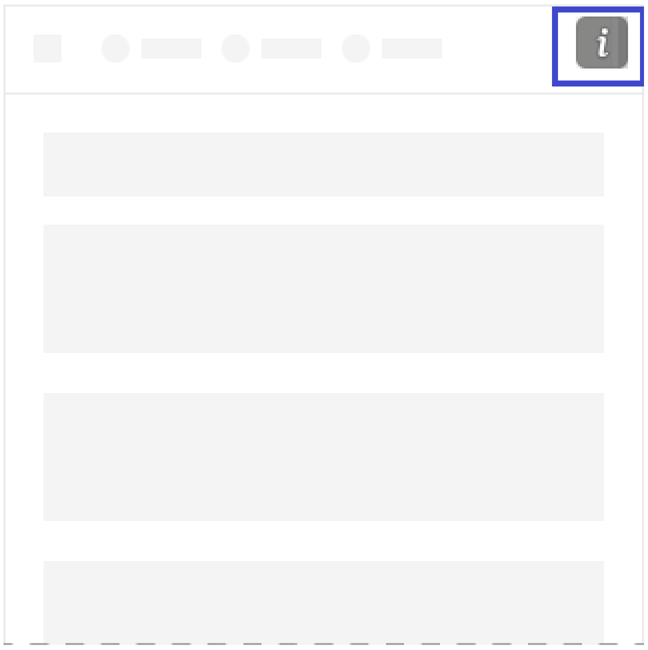
For Windows, the personality menu measures 12x32 pixels, as shown.

Figure 4. Personality menu on Windows



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

Figure 5. Personality menu on Mac



Implementation

For a sample that implements a task pane, see Excel Add-in JS WoodGrove Expense Trends on GitHub.

See also

- Fabric Core in Office Add-ins
- UX design patterns for Office Add-ins

Show or hide the task pane of your Office Add-in

Article • 02/12/2025

Important

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

You can show the task pane of your Office Add-in by calling the `Office.addin.showAsTaskpane()` method.

JavaScript

```javascript
function onCurrentQuarter() {
    Office.addin.showAsTaskpane()
    .then(function() {
        // Code that enables task pane UI elements for
        // working with the current quarter.
    });
}
```

The previous code assumes a scenario where there is an Excel worksheet named **CurrentQuarterSales**. The add-in will make the task pane visible whenever this worksheet is activated. The method `onCurrentQuarter` is a handler for the Office.Worksheet.onActivated event which has been registered for the worksheet.

You can also hide the task pane by calling the `Office.addin.hide()` method.

JavaScript

```javascript
function onCurrentQuarterDeactivated() {
    Office.addin.hide();
}
```

The previous code is a handler that is registered for the Office.Worksheet.onDeactivated event.

Additional details on showing the task pane

When you call `Office.addin.showAsTaskpane()`, Office will display in a task pane the file that you specified in the manifest. The configuration depends on what type of manifest you're using.

- **Unified manifest for Microsoft 365**: The URL of the file is assigned as the value of a "runtimes.code.page" property of the runtime object which has an action object of type "openPage".

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

- **Add-in only manifest**: The URL of the file is assigned as the resource ID (`resid`) value of the task pane. This `resid` value can be assigned or changed by opening your manifest file and locating **<SourceLocation>** inside the `<Action xsi:type="ShowTaskpane">` element.

(See Configure your Office Add-in to use a shared runtime for additional details.)

Since `Office.addin.showAsTaskpane()` is an asynchronous method, your code will continue running until the method is complete. Wait for this completion with either the `await` keyword or a `then()` method, depending on which JavaScript syntax you are using.

Configure your add-in to use the shared runtime

To use the `showAsTaskpane()` and `hide()` methods, your add-in must use the shared runtime. For more information, see Configure your Office Add-in to use a shared runtime.

Preservation of state and event listeners

The `hide()` and `showAsTaskpane()` methods only change the *visibility* of the task pane. They do not unload or reload it (or reinitialize its state).

Consider the following scenario: A task pane is designed with tabs. The **Home** tab is open when the add-in is first launched. Suppose a user opens the **Settings** tab and, later, code in the task pane calls `hide()` in response to some event. Still later code calls `showAsTaskpane()` in response to another event. The task pane will reappear, and the **Settings** tab is still selected.

My Add-in Name
Home
Settings
Favorites
Account

In addition, any event listeners that are registered in the task pane continue to run even when the task pane is hidden.

Consider the following scenario: The task pane has a registered handler for the Excel `Worksheet.onActivated` and `Worksheet.onDeactivated` events for a sheet named **Sheet1**. The activated handler causes a green dot to appear in the task pane. The deactivated handler turns the dot red (which is its default state). Suppose then that code calls `hide()` when **Sheet1** is not activated and the dot is red. While the task pane is hidden, **Sheet1** is activated. Later code calls `showAsTaskpane()` in response to some event. When the task pane opens, the dot is green because the event listeners and handlers ran even though the task pane was hidden.

Handle the visibility changed event

When your code changes the visibility of the task pane with `showAsTaskpane()` or `hide()`, Office triggers the `VisibilityModeChanged` event. It can be useful to handle this event. For example, suppose the task pane displays a list of all the sheets in a workbook. If a new worksheet is added while the task pane is hidden, making the task pane visible would not, in itself, add the new worksheet name to the list. But your code can respond to the `VisibilityModeChanged` event to reload the Worksheet.name property of all the worksheets in the Workbook.worksheets collection as shown in the example code below.

To register a handler for the event, you do not use an "add handler" method as you would in most Office JavaScript contexts. Instead, there is a special function to which you pass your handler: Office.addin.onVisibilityModeChanged. The following is an example. Note that the `args.visibilityMode` property is type VisibilityMode.

JavaScript

```
Office.addin.onVisibilityModeChanged(function(args) {
    if (args.visibilityMode == "Taskpane") {
        // Code that runs whenever the task pane is made visible.
        // For example, an Excel.run() that loads the names of
        // all worksheets and passes them to the task pane UI.
    }
});
```

The function returns another function that *deregisters* the handler. Here is a simple, but not robust, example.

JavaScript

```
const removeVisibilityModeHandler =
    Office.addin.onVisibilityModeChanged(function(args) {
        if (args.visibilityMode == "Taskpane") {
            // Code that runs whenever the task pane is made visible.
        }
    });


// In some later code path, deregister with:
removeVisibilityModeHandler();
```

The `onVisibilityModeChanged` method is asynchronous and returns a promise, which means that your code needs to await the fulfillment of the promise before it can call the **deregister** handler.

JavaScript

```
// await the promise from onVisibilityModeChanged and assign
// the returned deregister handler to removeVisibilityModeHandler.
const removeVisibilityModeHandler =
    await Office.addin.onVisibilityModeChanged(function(args) {
        if (args.visibilityMode == "Taskpane") {
            // Code that runs whenever the task pane is made visible.
        }
    });
```

The deregister function is also asynchronous and returns a promise. So, if you have code that should not run until after the deregistration is complete, then you should await the promise returned by the deregister function.

JavaScript

```javascript
// await the promise from the deregister handler before continuing
await removeVisibilityModeHandler();
// subsequent code here
```

See also

- Configure your Office Add-in to use a shared runtime
- Run code in your Office Add-in when the document opens

Add custom keyboard shortcuts to your Office Add-ins

Article • 03/12/2025

Keyboard shortcuts, also known as key combinations, make it possible for your add-in's users to work more efficiently. Keyboard shortcuts also improve the add-in's accessibility for users with disabilities by providing an alternative to the mouse.

There are three steps to add keyboard shortcuts to an add-in.

1. Configure the add-in's manifest to use a shared runtime.
2. Define custom keyboard shortcuts and the actions they'll run.
3. Map custom actions to their functions using the Office.actions.associate API.

Prerequisites

Keyboard shortcuts are currently only supported in the following platforms and build of **Excel** and **Word**.

- Office on the web
- Office on Windows
  - **Excel**: Version 2102 (Build 13801.20632) and later
  - **Word**: Version 2408 (Build 17928.20114) and later
- Office on Mac
  - **Excel**: Version 16.55 (21111400) and later
  - **Word**: Version 16.88 (24081116) and later

Additionally, keyboard shortcuts only work on platforms that support the following requirement sets. For information about requirement sets and how to work with them, see Specify Office applications and API requirements.

- SharedRuntime 1.1
- KeyboardShortcuts 1.1 (required if the add-in provides its users with the option to customize keyboard shortcuts)

Tip

To start with a working version of an add-in with keyboard shortcuts already configured, clone and run the Use keyboard shortcuts for Office Add-in actions sample. When you're ready to add keyboard shortcuts to your own add-in, continue with this article.

Define custom keyboard shortcuts

The process to define custom keyboard shortcuts for your add-in varies depending on the type of manifest your add-in uses. Select the tab for the type of manifest you're using.

Tip

To learn more about manifests for Office Add-ins, see Office Add-ins manifest.

Unified app manifest for Microsoft 365

Note

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

If your add-in uses the unified app manifest for Microsoft 365, custom keyboard shortcuts and their actions are defined in the manifest.

1. In your add-in project, open the **manifest.json** file.
2. Add the following object to the "extensions.runtimes" array. Note the following about this markup.
   - The "actions" objects specify the functions your add-in can run. In the following example, an add-in will be able to show and hide a task pane. You'll create these functions in a later section. Currently, custom keyboard shortcuts can only run actions that are of type "executeFunction".
   - While the "actions.displayName" property is optional, it's required if a custom keyboard shortcut will be created for the action. This property is used to describe the action of a keyboard shortcut. The description you provide appears in the dialog that's shown to a user when there's a shortcut conflict between multiple add-ins or with Microsoft 365. Office

appends the name of the add-in in parentheses at the end of the description. For more information on how conflicts with keyboard shortcuts are handled, see Avoid key combinations in use by other add-ins.

JSON

```json
"runtimes": [
    {
        "id": "TaskPaneRuntime",
        "type": "general",
        "code": {
            "page": "https://localhost:3000/taskpane.html"
        },
        "lifetime": "long",
        "actions": [
            {
                "id": "ShowTaskpane",
                "type": "executeFunction",
                "displayName": "Show task pane (Contoso Add-in)"
            },
            {
                "id": "HideTaskpane",
                "type": "executeFunction",
                "displayName": "Hide task pane (Contoso Add-in)"
            }
        ],
    }
]
```

3. Add the following to the "extensions" array. Note the following about the markup.

    - The SharedRuntime 1.1 requirement set is specified in the "requirements.capabilities" object to support custom keyboard shortcuts.
    - Each "shortcuts" object represents a single action that's invoked by a keyboard shortcut. It specifies the supported key combinations for various platforms, such as Office on the web, on Windows, and on Mac. For guidance on how to create custom key combinations, see Guidelines for custom key combinations.
    - A default key combination must be specified. It's used on all supported platforms if there isn't a specific combination configured for a particular platform.
    - The value of the "actionId" property must match the value specified in the "id" property of the applicable "extensions.runtimes.actions" object.

JSON

```json
"keyboardShortcuts": [
    {
        "requirements": {
            "capabilities": [
                {
                    "name": "SharedRuntime",
                    "minVersion": "1.1"
                }
            ]
        },
        "shortcuts": [
            {
                "key": {
                    "default": "Ctrl+Alt+Up",
                    "mac": "Command+Shift+Up",
                    "web": "Ctrl+Alt+1",
                    "windows": "Ctrl+Alt+Up"
                },
                "actionId": "ShowTaskpane"
            },
            {
                "key": {
                    "default": "Ctrl+Alt+Down",
                    "mac": "Command+Shift+Down",
                    "web": "Ctrl+Alt+2",
                    "windows": "Ctrl+Alt+Up"
                },
                "actionId": "HideTaskpane"
            }
        ]
    }
]
```

Map custom actions to their functions

1. In your project, open the JavaScript file loaded by your HTML page in the **<FunctionFile>** element.

2. In the JavaScript file, use the Office.actions.associate API to map each action you specified in an earlier step to a JavaScript function. Add the following JavaScript to the file. Note the following about the code.

    - The first parameter is the name of an action that you mapped to a keyboard shortcut. The location of the name of the action depends on the type of manifest your add-in uses.
        - **Unified app manifest for Microsoft 365**: The value of the "extensions.keyboardShortcuts.shortcuts.actionId" property in the

  manifest.json file.
  - **Add-in only manifest**: The value of the "actions.id" property in the shortcuts JSON file.
- The second parameter is the function that runs when a user presses the key combination that's mapped to an action.

JavaScript

```javascript
Office.actions.associate("ShowTaskpane", () => {
    return Office.addin.showAsTaskpane()
        .then(() => {
            return;
        })
        .catch((error) => {
            return error.code;
        });
});
```

JavaScript

```javascript
Office.actions.associate("HideTaskpane", () => {
    return Office.addin.hide()
        .then(() => {
            return;
        })
        .catch((error) => {
            return error.code;
        });
});
```

Guidelines for custom key combinations

Use the following guidelines to create custom key combinations for your add-ins.

- A keyboard shortcut must include at least one modifier key (`Alt`/`Option`, `Ctrl`/`Cmd`, `Shift`) and only one other key. These keys must be joined with a `+` character.
- The `Cmd` modifier key is supported on the macOS platform.
- On macOS, the `Alt` key is mapped to the `Option` key. On Windows, the `Cmd` key is mapped to the `Ctrl` key.
- The `Shift` key can't be used as the only modifier key. It must be combined with either `Alt`/`Option` or `Ctrl`/`Cmd`.
- Key combinations can include characters "A-Z", "a-z", "0-9", and the punctuation marks "-", "_", and "+". By convention, lowercase letters aren't used in keyboard shortcuts.

- When two characters are linked to the same physical key on a standard keyboard, then they're synonyms in a custom keyboard shortcut. For example, `Alt` + `a` and `Alt` + `A` are the same shortcut, as well as `Ctrl` + `-` and `Ctrl` + `_` ("-" and "_" are linked to the same physical key).

> **Note**
>
> Custom keyboard shortcuts must be pressed simultaneously. KeyTips, also known as sequential key shortcuts (for example, `Alt` + `H`, `H`), aren't supported in Office Add-ins.

Browser shortcuts that can't be overridden

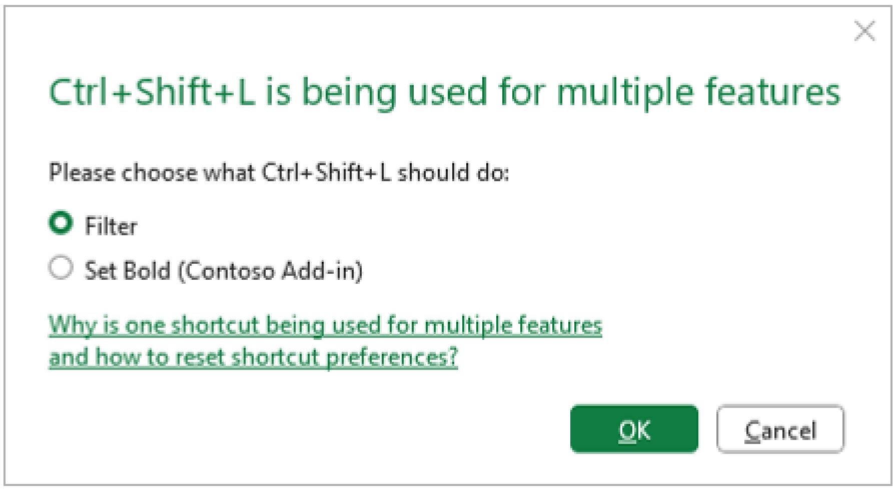
When using custom keyboard shortcuts on the web, some keyboard shortcuts that are used by the browser can't be overridden by add-ins. The following list is a work in progress. If you discover other combinations that can't be overridden, please let us know by using the feedback tool at the bottom of this page.

- `Ctrl` + `N`
- `Ctrl` + `Shift` + `N`
- `Ctrl` + `T`
- `Ctrl` + `Shift` + `T`
- `Ctrl` + `W`
- `Ctrl` + `PgUp` / `PgDn`

Avoid key combinations in use by other add-ins

There are many keyboard shortcuts that are already in use by Microsoft 365. Avoid registering keyboard shortcuts for your add-in that are already in use. However, there may be some instances where it's necessary to override existing keyboard shortcuts or handle conflicts between multiple add-ins that have registered the same keyboard shortcut.

In the case of a conflict, the user will see a dialog box the first time they attempt to use a conflicting keyboard shortcut. Note that the source of the text for the add-in option that's displayed in this dialog varies depending on the type of manifest your add-in uses. - **Unified app manifest for Microsoft 365**: The value of the "extensions.runtimes.actions.displayName" property in the **manifest.json** file. - **Add-in only manifest**: The value of the "actions.name" property in the shortcuts JSON file.



Ctrl+Shift+L is being used for multiple features

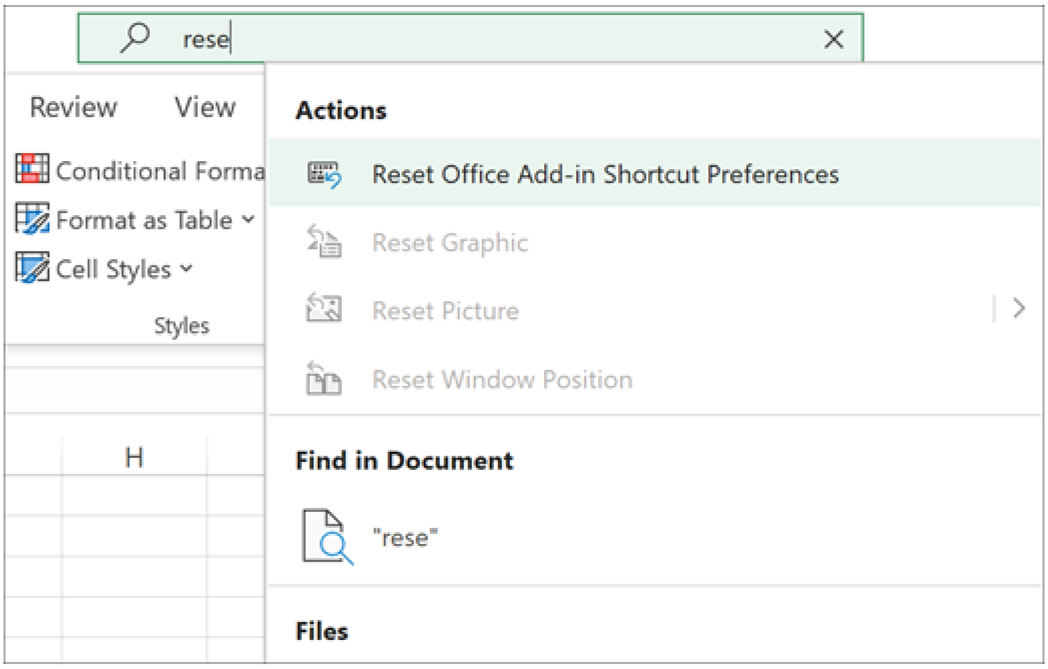
Please choose what Ctrl+Shift+L should do:

[✓ Filter]

Set Bold (Contoso Add-in)

Why is one shortcut being used for multiple features and how to reset shortcut preferences?

The user can select which action the keyboard shortcut will take. After making the selection, the preference is saved for future uses of the same shortcut. The shortcut preferences are saved per user, per platform. If the user wishes to change their preferences, they can invoke the **Reset Office Add-ins shortcut preferences** command from the **Tell me** search box. Invoking the command clears all of the user's add-in shortcut preferences and the user will again be prompted with the conflict dialog box the next time they attempt to use a conflicting shortcut.



For the best user experience, we recommend that you minimize keyboard shortcut conflicts with these good practices.

- Use only keyboard shortcuts with the following pattern: `Ctrl` + `Shift` + `Alt` +*x*, where *x* is some other key.
- Avoid using established keyboard shortcuts in Excel and Word. For a list, see the following:
  - Keyboard shortcuts in Excel
  - Keyboard shortcuts in Word

- When the keyboard focus is inside the add-in UI, `Ctrl` + `Space` and `Ctrl` + `Shift` + `F10` won't work as these are essential accessibility shortcuts.
- On a Windows or Mac computer, if the **Reset Office Add-ins shortcut preferences** command isn't available on the search menu, the user can manually add the command to the ribbon by customizing the ribbon through the context menu.

Localize the description of a keyboard shortcut

You may need to localize your custom keyboard shortcuts in the following scenarios.

- Your add-in supports multiple locales.
- Your add-in supports different alphabets, writing systems, or keyboard layouts.

Guidance on how to localize your keyboard shortcuts varies depending on the type of manifest your add-in uses.

Unified app manifest for Microsoft 365

To learn how to localize your custom keyboard shortcuts with the unified app manifest for Microsoft 365, see Localize strings in your app manifest.

Turn on shortcut customization for specific users

Note

The APIs described in this section require the KeyboardShortcuts 1.1 requirement set.

Users of your add-in can reassign the actions of the add-in to alternate keyboard combinations.

Use the Office.actions.replaceShortcuts method to assign a user's custom keyboard combinations to your add-ins actions. The method takes a parameter of type `{[actionId:string]: string|null}`, where the `actionId`s are a subset of the action IDs that must be defined in the add-in's extended manifest JSON. The values are the user's preferred key combinations. The value can also be `null`, which will remove any customization for that `actionId` and revert to the specified default keyboard combination.

If the user is logged into Microsoft 365, the custom combinations are saved in the user's roaming settings per platform. Customizing shortcuts aren't currently supported for anonymous users.

```JavaScript
const userCustomShortcuts = {
    ShowTaskpane: "Ctrl+Shift+1",
    HideTaskpane: "Ctrl+Shift+2"
};

Office.actions.replaceShortcuts(userCustomShortcuts)
    .then(() => {
        console.log("Successfully registered shortcut.");
    })
    .catch((error) => {
        if (error.code == "InvalidOperation") {
            console.log("ActionId doesn't exist or shortcut combination is
invalid.");
        }
    });
```

To find out what shortcuts are already in use for the user, call the Office.actions.getShortcuts method. This method returns an object of type `[actionId:string]:string|null}`, where the values represent the current keyboard combination the user must use to invoke the specified action. The values can come from three different sources.

- If there was a conflict with the shortcut and the user has chosen to use a different action (either native or another add-in) for that keyboard combination, the value returned will be `null` since the shortcut has been overridden and there's no keyboard combination the user can currently use to invoke that add-in action.
- If the shortcut has been customized using the Office.actions.replaceShortcuts method, the value returned will be the customized keyboard combination.
- If the shortcut hasn't been overridden or customized, the value returned varies depending on the type of manifest the add-in uses.
    - **Unified app manifest for Microsoft 365**: The shortcut specified in the **manifest.json** file of the add-in.
    - **Add-in only manifest**: The shortcut specified in the shortcuts JSON file of the add-in.

The following is an example.

```JavaScript
```

```
Office.actions.getShortcuts()
    .then((userShortcuts) => {
       for (const action in userShortcuts) {
           let shortcut = userShortcuts[action];
           console.log(action + ": " + shortcut);
       }
    });
```

As described in Avoid key combinations in use by other add-ins, it's a good practice to avoid conflicts in shortcuts. To discover if one or more key combinations are already in use, pass them as an array of strings to the Office.actions.areShortcutsInUse method. The method returns a report containing key combinations that are already in use in the form of an array of objects of type `{shortcut: string, inUse: boolean}`. The `shortcut` property is a key combination, such as "Ctrl+Shift+1". If the combination is already registered to another action, the `inUse` property is set to `true`. For example, `[{shortcut: "Ctrl+Shift+1", inUse: true}, {shortcut: "Ctrl+Shift+2", inUse: false}]`. The following code snippet is an example.

JavaScript

```javascript
const shortcuts = ["Ctrl+Shift+1", "Ctrl+Shift+2"];
Office.actions.areShortcutsInUse(shortcuts)
    .then((inUseArray) => {
        const availableShortcuts = inUseArray.filter((shortcut) => {
            return !shortcut.inUse;
        });
        console.log(availableShortcuts);
        const usedShortcuts = inUseArray.filter((shortcut) => {
            return shortcut.inUse;
        });
        console.log(usedShortcuts);
    });
```

Implement custom keyboard shortcuts across supported Microsoft 365 apps

You can implement a custom keyboard shortcut to be used across supported Microsoft 365 apps, such as Excel and Word. If the implementation to perform the same task is different on each app, you must use the `Office.actions.associate` method to call a different callback function for each app. The following code is an example.

JavaScript

```
const host = Office.context.host;
if (host === Office.HostType.Excel) {
    Office.actions.associate("ChangeFormat", changeFormatExcel);
} else if (host === Office.HostType.Word) {
    Office.actions.associate("ChangeFormat", changeFormatWord);
}
...
```

See also

- Office Add-in sample: Use keyboard shortcuts for Office Add-in actions
- Shared runtime requirement sets
- Keyboard shortcuts requirement sets