Add-in only manifest reference for Office Add-ins

Article • 08/05/2024

This section contains information about every element used by an Office Add-in's add-in only manifest. To learn more about how the manifest describes your add-in to an Office application, see Office Add-ins with an add-in only manifest.

Elements

Expand table

Name	Category
Action	VersionOverrides
AllFormFactors	VersionOverrides
AllowSnapshot	General
AlternateId	General
AppDomain	General
AppDomains	General
CitationText	General
Control	VersionOverrides
Control (Button)	VersionOverrides
Control (Menu)	VersionOverrides
Control (MobileButton)	VersionOverrides
CustomTab	VersionOverrides
DefaultLocale	General
DefaultSettings	General
Description	General
DesktopFormFactor	VersionOverrides
DesktopSettings	General

Name	Category
Dictionary	General
DictionaryHomePage	General
DisableEntityHighlighting	General
DisplayName	General
Enabled	VersionOverrides
EquivalentAddin	General
EquivalentAddins	General
Event	VersionOverrides
ExtendedPermission	VersionOverrides
ExtendedPermissions	VersionOverrides
ExtensionPoint	VersionOverrides
ExtendedOverrides	General
FileName	General
Form	General
FormSettings	General
FunctionFile	VersionOverrides
GetStarted	VersionOverrides
Group	VersionOverrides
HighResolutionIconUrl	General
Host	General
Hosts	General
Icon	VersionOverrides
IconUrl	General
Id	General
Image	VersionOverrides
Images	VersionOverrides

Name	Category
Item	VersionOverrides
Items	VersionOverrides
LaunchEvent	VersionOverrides
LaunchEvents	VersionOverrides
LongStrings	VersionOverrides
Metadata	General
Method	General
Methods	General
MobileFormFactor	VersionOverrides
Namespace	General
OfficeApp	General
OfficeMenu	VersionOverrides
OfficeTab	VersionOverrides
OverriddenByRibbonApi	VersionOverrides
Override	General
Page	VersionOverrides
Permissions	General
PhoneSettings	General
ProgId	General
ProviderName	General
QueryUri	General
RequestedHeight	General
RequestedWidth	General
Requirements	General
Resources	VersionOverrides
Rule	General

Name	Category
Runtime	VersionOverrides
Runtimes	VersionOverrides
Scopes	VersionOverrides
Script	VersionOverrides
Set	General
Sets	General
ShortStrings	VersionOverrides
SourceLocation	General
SourceLocation (custom functions)	VersionOverrides
String	VersionOverrides
Supertip	VersionOverrides
SupportsSharedFolders	VersionOverrides
SupportUrl	General
TabletSettings	General
TargetDialect	General
TargetDialects	General
Tokens	General
Token	General
Type	General
Url	VersionOverrides
Urls	VersionOverrides
Version	General
VersionOverrides	General
VersionOverrides 1.0 Content	VersionOverrides
VersionOverrides 1.0 Mail	VersionOverrides
VersionOverrides 1.1 Mail	VersionOverrides

Name	Category
VersionOverrides 1.0 TaskPane	VersionOverrides
WebApplicationInfo	VersionOverrides

How to find the proper order of manifest elements

06/13/2025

The XML elements in the manifest of an Office Add-in must be under the proper parent element *and* in a specific order, relative to each other, under the parent.

The required ordering is specified in the XSD files in the Schemas folder. The XSD files are categorized into subfolders for taskpane, content, and mail add-ins.

For example, in the `<OfficeApp>` element, the `<Id>`, `<Version>`, `<ProviderName>` must appear in that order. If an `<AlternateId>` element is added, it must be between the `<Id>` and `<Version>` element. Your manifest will not be valid and your add-in will not load, if any element is in the wrong order.

> **Note**
>
> The validator within office-addin-manifest uses the same error message when an element is out-of-order as it does when an element is under the wrong parent. The error says the child element is not a valid child of the parent element. If you get such an error but the reference documentation for the child element indicates that it *is* valid for the parent, then the problem is likely that the child has been placed in the wrong order.

The following sections show the manifest elements in the order in which they must appear. There are differences depending on whether the `type` attribute of the `<OfficeApp>` element is `TaskPaneApp`, `ContentApp`, or `MailApp`. To keep these sections from becoming too unwieldy, the highly complex `<VersionOverrides>` element is broken out into separate sections.

> **Note**
>
> Not all of the elements shown are mandatory. If the `minOccurs` value for a element is **0** in the schema, the element is optional.

Basic task pane add-in element ordering

XML

```xml
<OfficeApp xsi:type="TaskPaneApp">
    <Id>
```

```
    <AlternateID>
    <Version>
    <ProviderName>
    <DefaultLocale>
    <DisplayName>
        <Override>
    <Description>
        <Override>
    <IconUrl>
        <Override>
    <HighResolutionIconUrl>
        <Override>
    <SupportUrl>
    <AppDomains>
        <AppDomain>
    <Hosts>
        <Host>
    <Requirements>
        <Sets>
            <Set>
        <Methods>
            <Method>
    <DefaultSettings>
        <SourceLocation>
            <Override>
    <Permissions>
    <Dictionary>
        <TargetDialects>
        <QueryUri>
        <CitationText>
        <DictionaryName>
        <DictionaryHomePage>
    <VersionOverrides>*
    <ExtendedOverrides>
```

*See Task pane add-in element ordering within VersionOverrides for the ordering of children elements of VersionOverrides.

Basic mail add-in element ordering

XML

```
<OfficeApp xsi:type="MailApp">
    <Id>
    <AlternateId>
    <Version>
    <ProviderName>
    <DefaultLocale>
    <DisplayName>
        <Override>
    <Description>
```

```
        <Override>
    <IconUrl>
        <Override>
    <HighResolutionIconUrl>
        <Override>
    <SupportUrl>
    <AppDomains>
        <AppDomain>
    <Hosts>
        <Host>
    <Requirements>
        <Sets>
            <Set>
    <FormSettings>
        <Form>
        <DesktopSettings>
            <SourceLocation>
            <RequestedHeight>
        <TabletSettings>
            <SourceLocation>
            <RequestedHeight>
        <PhoneSettings>
            <SourceLocation>
    <Permissions>
    <Rule>
    <DisableEntityHighlighting>
    <VersionOverrides>*
```

*See Mail add-in element ordering within VersionOverrides Ver. 1.0 and Mail add-in element ordering within VersionOverrides Ver. 1.1 for the ordering of children elements of VersionOverrides.

Basic content add-in element ordering

XML

```
<OfficeApp xsi:type="ContentApp">
    <Id>
    <AlternateId>
    <Version>
    <ProviderName>
    <DefaultLocale>
    <DisplayName>
        <Override>
    <Description>
        <Override>
    <IconUrl >
        <Override>
    <HighResolutionIconUrl>
        <Override>
    <SupportUrl>
```

```
    <AppDomains>
        <AppDomain>
    <Hosts>
        <Host>
    <Requirements>
    <Sets>
        <Set>
    <Methods>
        <Method>
    <DefaultSettings>
        <SourceLocation>
            <Override>
        <RequestedWidth>
        <RequestedHeight>
    <Permissions>
    <AllowSnapshot>
    <VersionOverrides>*
```

*See Content add-in element ordering within VersionOverrides for the ordering of children elements of VersionOverrides.

Task pane add-in element ordering within VersionOverrides

XML

```
<VersionOverrides>
    <Description>
    <Requirements>
        <Sets>
            <Set>
    <Hosts>
        <Host>
            <Runtimes>
                <Runtime>
            <AllFormFactors>
                <ExtensionPoint>
                    <Script>
                        <SourceLocation>
                    <Page>
                        <SourceLocation>
                    <Metadata>
                        <SourceLocation>
                    <Namespace>
            <DesktopFormFactor>
                <GetStarted>
                    <Title>
                    <Description>
                    <LearnMoreUrl>
                <FunctionFile>
```

```
                <ExtensionPoint>
                    <OfficeTab>
                        <Group>
                            <Label>
                            <Icon>
                                <Image>
                            <Control>
                            <Label>
                            <Supertip>
                                <Title>
                                <Description>
                            <Icon>
                                <Image>
                            <Action>
                                <TaskpaneId>
                                <SourceLocation>
                                <Title>
                                <FunctionName>
                            <Enabled>
                            <Items>
                                <Item>
                                <Label>
                                <Supertip>
                                    <Title>
                                    <Description>
                                <Action>
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
                    <CustomTab>
                        <Group> (can be below <OfficeGroup>)
                            <OverriddenByRibbonApi>
                            <Label>
                            <Icon>
                                <Image>
                            <Control>
                                <OverriddenByRibbonApi>
                                <Label>
                                <Supertip>
                                    <Title>
                                    <Description>
                                <Icon>
                                    <Image>
                                <Action>
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
                                <Enabled>
                                <Items>
                                    <Item>
                                        <OverriddenByRibbonApi>
                                        <Label>
                                        <Supertip>
```

```
                                    <Title>
                                    <Description>
                                <Action>
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
                    <OfficeGroup> (can be above <Group>)
                    <Label>
                    <InsertAfter> (or <InsertBefore>)
                <OfficeMenu>
                    <Control>
                        <Label>
                        <Supertip>
                            <Title>
                            <Description>
                        <Icon>
                            <Image>
                        <Action>
                            <TaskpaneId>
                            <SourceLocation>
                            <Title>
                            <FunctionName>
                        <Enabled>
                        <Items>
                            <Item>
                                <Label>
                                <Supertip>
                                    <Title>
                                    <Description>
                                <Action>
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
    <Resources>
        <Images>
            <Image>
                <Override>
        <Urls>
            <Url>
                <Override>
        <ShortStrings>
            <String>
                <Override>
        <LongStrings>
            <String>
                <Override>
    <WebApplicationInfo>
        <Id>
        <Resource>
        <Scopes>
            <Scope>
    <EquivalentAddins>
        <EquivalentAddin>
```

```
                <ProgId>
                <DisplayName>
                <FileName>
                <Type>
            <Effect>
```

Mail add-in element ordering within VersionOverrides Ver. 1.0

XML

```
<VersionOverrides>
    <Description>
    <Requirements>
        <Sets>
            <Set>
    <Hosts>
        <Host>
            <DesktopFormFactor>
                <ExtensionPoint>
                    <OfficeTab>
                        <Group>
                            <Label>
                            <Control>
                                <Label>
                                <Supertip>
                                    <Title>
                                    <Description>
                                <Icon>
                                    <Image>
                                <Action>
                                    <SourceLocation>
                                    <FunctionName>
                    <CustomTab>
                        <Group>
                            <Label>
                            <Icon>
                                <Image>
                            <Control>
                                <Label>
                                <Supertip>
                                    <Title>
                                    <Description>
                                <Icon>
                                    <Image>
                                <Action>
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
                                <Items>
```

```
                                    <Item>
                                        <Label>
                                        <Supertip>
                                            <Title>
                                            <Description>
                                        <Action>
                                            <TaskpaneId>
                                            <SourceLocation>
                                            <Title>
                                            <FunctionName>
                        <Label>
                    <OfficeMenu>
                        <Control>
                            <Label>
                            <Supertip>
                                <Title>
                                <Description>
                            <Icon>
                                <Image>
                            <Action>
                                <TaskpaneId>
                                <SourceLocation>
                                <Title>
                                <FunctionName>
                            <Items>
                                <Item>
                                    <Label>
                                    <Supertip>
                                        <Title>
                                        <Description>
                                    <Action>
                                        <TaskpaneId>
                                        <SourceLocation>
                                        <Title>
                                        <FunctionName>
    <Resources>
        <Images>
            <Image>
                <Override>
        <Urls>
            <Url>
                <Override>
        <ShortStrings>
            <String>
                <Override>
        <LongStrings>
            <String>
                <Override>
    <VersionOverrides>*
```

* A VersionOverrides with `type` value `VersionOverridesV1_1`, instead of `VersionOverridesV1_0`, can be nested at the end of the outer VersionOverrides. See Mail add-in element ordering within VersionOverrides Ver. 1.1 for the ordering of elements in `VersionOverridesV1_1`.

Mail add-in element ordering within VersionOverrides Ver. 1.1

XML

```
<VersionOverrides>
    <Description>
    <Requirements>
    <Sets>
        <Set>
    <Hosts>
    <Host>
        <DesktopFormFactor>
            <ExtensionPoint>
                <OfficeTab>
                    <Group>
                        <Label>
                        <Tooltip>
                        <Control>
                            <Label>
                            <Supertip>
                                <Title>
                                <Description>
                            <Icon>
                                <Image>
                            <Action>
                                <SourceLocation>
                                <FunctionName>
                <CustomTab>
                    <Group>
                        <Label>
                        <Icon>
                            <Image>
                        <Control>
                            <Label>
                            <Supertip>
                                <Title>
                                <Description>
                            <Icon>
                                <Image>
                            <Action>
                                <TaskpaneId>
                                <SourceLocation>
                                <Title>
                                <FunctionName>
                            <Items>
                                <Item>
                                    <Label>
                                    <Supertip>
                                        <Title>
                                        <Description>
                                    <Action>
```

```
                                    <TaskpaneId>
                                    <SourceLocation>
                                    <Title>
                                    <FunctionName>
                <Label>
            <OfficeMenu>
                <Control>
                    <Label>
                    <Supertip>
                        <Title>
                        <Description>
                    <Icon>
                        <Image>
                    <Action>
                        <TaskpaneId>
                        <SourceLocation>
                        <Title>
                        <FunctionName>
                    <Items>
                        <Item>
                            <Label>
                            <Supertip>
                                <Title>
                                <Description>
                            <Action>
                                <TaskpaneId>
                                <SourceLocation>
                                <Title>
                                <FunctionName>
            <SourceLocation>
            <Label>
            <CommandSurface>
        <MobileFormFactor>
            <ExtensionPoint>
                <Group>
                    <Label>
                    <Control>
                        <Label>
                        <Icon>
                            <Image>
                        <Action>
                            <SourceLocation>
                            <FunctionName>
                <Control>
                    <Label>
                    <Icon>
                        <Image>
                    <Action>
                        <SourceLocation>
                        <FunctionName>
    <Resources>
        <Images>
            <Image>
                <Override>
        <Urls>
```

```
            <Url>
                <Override>
        <ShortStrings>
            <String>
                <Override>
        <LongStrings>
            <String>
                <Override>
    <WebApplicationInfo>
        <Id>
        <Resource>
        <Scopes>
            <Scope>
    <EquivalentAddins>
        <EquivalentAddin>
            <ProgId>
            <DisplayName>
            <FileName>
            <Type>
    <ConnectedServiceControls>
        <ConnectedServiceControlsScopes>
            <Scope>
    <ExtendedPermissions>
        <ExtendedPermission>
```

Content add-in element ordering within VersionOverrides

XML

```
<VersionOverrides>
    <WebApplicationInfo>
        <Id>
        <Resource>
        <Scopes>
            <Scope>
```

See also

- Reference for Office Add-ins manifests (v1.1)
- Official schema definitions

Referencing the Office JavaScript API library

Article • 01/16/2025

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

HTML

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

This will download and cache the Office JavaScript API files the first time your add-in loads to make sure that it is using the most up-to-date implementation of Office.js and its associated files for the specified version.

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

Office.js-specific web API behavior

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

HTML

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

```
</script>

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

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

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

API versioning and backward compatibility

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

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

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

Enabling IntelliSense for a TypeScript project

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

command line

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

Preview APIs

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

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

CDN references for other Microsoft 365 environments

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

See also

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

Privacy and security for Office Add-ins

07/24/2025

Process security

Office Add-ins are secured by an add-in runtime environment, a multiple-tier permissions model, and performance governors. This framework protects the user's experience in the following ways.

- Access to the Office client application's UI frame is managed.
- Only indirect access to the Office client application's UI thread is allowed.
- Modal interactions aren't allowed - for example, calls to JavaScript `alert`, `confirm`, and `prompt` methods aren't allowed because they're modal.

Further, the runtime framework provides the following benefits to ensure that an Office Add-in can't damage the user's environment.

- Isolates the process the add-in runs in.
- Doesn't require .dll or .exe replacement or ActiveX components.
- Makes add-ins easy to install and uninstall.

Also, the use of memory, CPU, and network resources by Office Add-ins is governable to ensure that good performance and reliability are maintained.

> **Note**
>
> In some scenarios, different features of an add-in run in separate runtimes. For simplicity, this article uses the singular "runtime." For more information, see Runtimes in Office Add-ins.

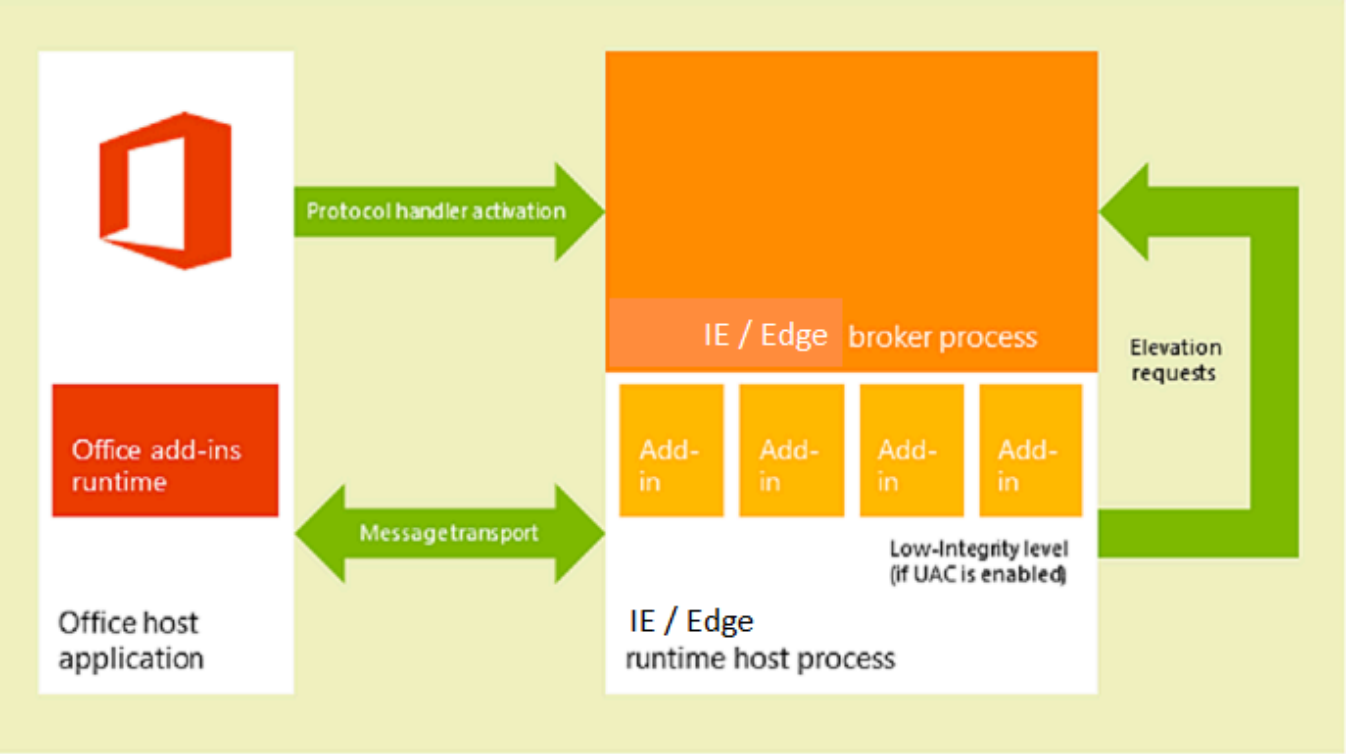
The following sections briefly describe how the runtime architecture supports running add-ins in Office clients on Windows-based devices, on Mac OS X devices, and in web browsers.

Clients on Windows and OS X devices

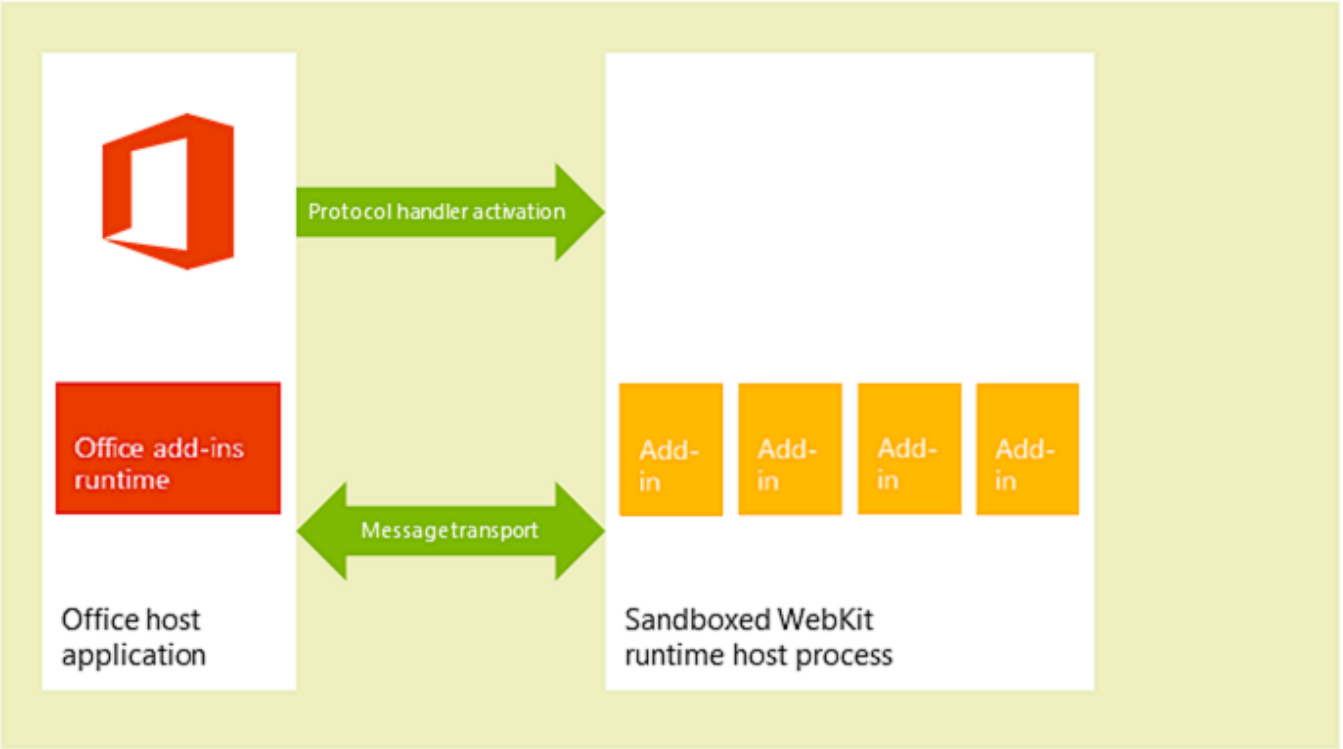
In supported clients for desktop and tablet devices, such as Excel on Windows, and Outlook on Windows (classic) and on Mac, Office Add-ins are supported by integrating an in-process component, the Office Add-ins runtime, which manages the add-in lifecycle and enables

interoperability between the add-in and the client application. The add-in webpage itself is hosted out-of-process. On a Windows desktop or tablet device, the add-in webpage is hosted inside an Internet Explorer or Microsoft Edge control which, in turn, is hosted inside an add-in runtime process that provides security and performance isolation.

On Windows desktops, Protected Mode in Internet Explorer must be enabled for the Restricted Site Zone. This is typically enabled by default. If it is disabled, an error will occur when you try to launch an add-in.



On a macOS desktop, the add-in web page is hosted inside a sandboxed WebKit runtime host process which helps provide similar level of security and performance protection.



The Office Add-ins runtime manages interprocess communication, the translation of JavaScript API calls and events into native ones, as well as UI remoting support to enable the add-in to be rendered inside the document, in a task pane, or adjacent to an email message, meeting request, or appointment.

Web clients

In supported web clients, Office Add-ins are hosted in an **iframe** that runs using the HTML5 **sandbox** attribute. ActiveX components or navigating the main page of the web client are not allowed. Office Add-ins support is enabled in the web clients by the integration of the JavaScript API for Office. In a similar way to the desktop client applications, the JavaScript API manages the add-in lifecycle and interoperability between the add-in and the web client. This interoperability is implemented by using a special cross-frame post message communication infrastructure. The same JavaScript library (Office.js) that is used on desktop clients is available to interact with the web client. The following figure shows the infrastructure that supports add-ins in Office running in the browser, and the relevant components (the web client, **iframe**, Office Add-ins runtime, and JavaScript API for Office) required to support them.

Add-in integrity in AppSource

You can make your Office Add-ins available to the public by publishing them to AppSource. AppSource enforces the following measures to maintain the integrity of add-ins.

- Requires the host server of an Office Add-in to always use Secure Sockets Layer (SSL) to communicate.
- Requires a developer to provide proof of identity, a contractual agreement, and a compliant privacy policy to submit add-ins.
- Supports a user-review system for available add-ins to promote a self-policing community.

Optional connected experiences

End users and IT admins can turn off optional connected experiences in Office desktop and mobile clients. For Office Add-ins, the impact of disabling the **Optional connected experiences** setting is that users can no longer access add-ins or the Microsoft 365 and Copilot store through these clients. However, certain Microsoft add-ins that are considered essential or business-critical, and add-ins deployed by an organization's IT admin through Centralized Deployment will still be available. In Outlook, the availability of add-ins when the **Optional**

connected experiences setting is turned off varies depending on the client. For more information, see Optional connected experiences in Outlook.

Note that if an IT admin disables the use of connected experiences in Office, it has the same effect on add-ins as turning off just optional connected experiences.

Optional connected experiences in Outlook

The following table describes the availability of add-ins on Outlook clients when optional connected experiences is turned off.

Expand table

Client	Behavior when optional connected experiences is turned off
• Web browser • new Outlook on Windows	Admin-deployed add-ins remain visible and usable. While users can still access the Microsoft 365 and Copilot store and install add-ins, these add-ins can't be used until the **Optional connected experiences** setting is turned on. Additionally, sideloaded add-ins can't be used.
• Windows (classic)	All add-ins remain visible and usable. However, the Microsoft 365 and Copilot store is inaccessible.
• Mac	Add-ins don't appear in the ribbon and the Microsoft 365 and Copilot store is inaccessible.
• Android • iOS	The **Get Add-ins** dialog shows only admin-deployed add-ins.

Addressing end users' privacy concerns

This section describes the protection offered by the Office Add-ins platform from the customer's (end user's) perspective, and provides guidelines for how to support users' expectations and how to securely handle users' personally identifiable information (PII).

End users' perspective

Office Add-ins are built using web technologies that run in a browser control or **iframe**. Because of this, using add-ins is similar to browsing to web sites on the Internet or intranet. Add-ins can be external to an organization (if you acquire the add-in from AppSource) or internal (if you acquire the add-in from an Exchange Server add-in catalog, SharePoint app catalog, or file share on an organization's network). Add-ins have limited access to the network

and most add-ins can read or write to the active document or mail item. The add-in platform applies certain constraints before a user or administrator installs or starts an add-in. But as with any extensibility model, users should be cautious before starting an unknown add-in.

> **Note**
>
> Users may see a security prompt to trust the domain the first time an add-in is loaded. This will happen if the add-in's domain host is outside of the domain of Exchange on-premise or Office Online Server.

The add-in platform addresses end users' privacy concerns in the following ways.

- Data communicated with the web server that hosts a content, Outlook or task pane add-in as well as communication between the add-in and any web services it uses must be encrypted using the Secure Socket Layer (SSL) protocol.
- Before a user installs an add-in from AppSource, the user can view the privacy policy and requirements of that add-in. In addition, Outlook add-ins that interact with users' mailboxes surface the specific permissions that they require; the user can review the terms of use, requested permissions and privacy policy before installing an Outlook add-in.
- When sharing a document, users also share add-ins that have been inserted in or associated with that document. If a user opens a document that contains an add-in that the user hasn't used before, the Office client application prompts the user to grant permission for the add-in to run in the document. In an organizational environment, the Office client application also prompts the user if the document comes from an external source.
- Add-ins running in the following Office applications are blocked from accessing a user's device capabilities.
    - 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

  A user's device capabilities include their camera, geolocation, and microphone. To learn more, see View, manage, and install add-ins for Excel, PowerPoint, and Word.
- Users can enable or disable the access to AppSource. For content and task pane add-ins, users manage access to trusted add-ins and catalogs from the **Trust Center** on the host

Office client (opened from **File** > **Options** > **Trust Center** > **Trust Center Settings** > **Trusted Add-in Catalogs**).

In Outlook, access to manage add-ins depends on the user's Outlook client. To learn more, see Use add-ins in Outlook.

Administrators can also manage access to AppSource through the admin center.

- The design of the add-in platform provides security and performance for end users in the following ways.
  - An Office Add-in runs in a web browser control that is hosted in an add-in runtime environment separate from the Office client application. This design provides both security and performance isolation from the client application.
  - Running in a web browser control allows the add-in to do almost anything a regular web page running in a browser can do but, at the same time, restricts the add-in to observe the same-origin policy for domain isolation and security zones.

End users' perspective in Outlook

The following points address end users' privacy concerns specific to Outlook.

- End user's messages that are protected by Outlook's Information Rights Management (IRM) won't interact with add-ins in the following instances.
  - When the IRM-protected message is accessed from Outlook on mobile devices.
  - When the IRM-protected message contains a sensitivity label with the **Allow programmatic access** custom policy option set to `false`.

  For more information on IRM support in add-ins, see Mail items protected by IRM.
- Granting the **restricted** permission allows the Outlook add-in to have limited access on only the current item. Granting the **read item** permission allows the Outlook add-in to access personal identifiable information, such as sender and recipient names and email addresses, on only the current item. For more information on Outlook add-in permissions, see Understanding Outlook add-in permissions.
- Manifest files of installed Outlook add-ins are secured in the user's email account.
- Outlook on Windows (classic) and on Mac monitor the performance of installed Outlook add-ins, exercise governance control, and make add-ins unavailable when they exceed limits in the following areas.

    - Response time to activate
    - Number of failures to activate or reactivate
    - Memory usage
    - CPU usage

Developer guidelines to handle PII

The following lists some specific PII protection guidelines for you as a developer of Office Add-ins.

- The Settings object is intended for persisting add-in settings and state data across sessions for a content or task pane add-in, but don't store passwords and other sensitive PII in the **Settings** object. The data in the **Settings** object isn't visible to end users, but it is stored as part of the document's file format which is readily accessible. You should limit your add-in's use of PII and store any PII required by your add-in on the server hosting your add-in as a user-secured resource.
- Using some applications can reveal PII. Make sure that you securely store data for your users' identity, location, access times, and any other credentials so that data won't become available to other users of the add-in.
- If your add-in is available in AppSource, the AppSource requirement for HTTPS protects PII transmitted between your web server and the client computer or device. However, if you re-transmit that data to other servers, make sure you observe the same level of protection.
- If you store users' PII, make sure you reveal that fact, and provide a way for users to inspect and delete it. If you submit your add-in to AppSource, you can outline the data you collect and how it's used in the privacy statement.

Developers' permission choices and security practices

Follow these general guidelines to support the security model of Office Add-ins, and drill down on more details for each add-in type.

Request the necessary permissions

The add-in platform provides a permissions model that your add-in uses to declare the level of access to a user's data that it requires for its features. Each permission level corresponds to the subset of the JavaScript API for Office your add-in is allowed to use for its features. For example, the **write document** permission for content and task pane add-ins allows access to the Document.setSelectedDataAsync method that lets an add-in write to the user's document, but doesn't allow access to any of the methods for reading data from the document. This permission level makes sense for add-ins that only need to write to a document, such as an add-in where the user can query for data to insert into their document.

As a best practice, you should request permissions based on the principle of *least privilege*. That is, you should request permission to access only the minimum subset of the API that your add-in requires to function correctly. For example, if your add-in needs only to read data in a user's document for its features, you should request no more than the **read document** permission. (But, keep in mind that requesting insufficient permissions will result in the add-in platform blocking your add-in's use of some APIs and will generate errors at run time.)

You specify permissions in the manifest of your add-in, as shown in the example in this section below, and end users can see the requested permission level of an add-in before they decide to install or activate the add-in for the first time. Additionally, Outlook add-ins that request the **read/write mailbox** permission require explicit administrator privilege to install.

To see an example of how to request permissions in the manifest, open the tab for the 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.

The following example shows how a task pane add-in specifies the **read document** permission in its manifest. To keep permissions as the focus, other elements in the manifest aren't displayed.

JSON

```
"authorization": {
  "permissions": {
    "resourceSpecific": [
      ...
      {
        "name": "Document.Read.User",
```

```
        "type": "Delegated"
      },
    ]
  }
}
```

For more information about permissions for task pane and content add-ins, see Requesting permissions for API use in add-ins.

For more information about permissions for Outlook add-ins, see Understanding Outlook add-in permissions.

Follow the same-origin policy

Because Office Add-ins are webpages that run in a web browser control, they must follow the same-origin policy enforced by the browser. By default, a webpage in one domain can't make XmlHttpRequest web service calls to another domain other than the one where it is hosted.

One way to overcome this limitation is to use JSON/P -- provide a proxy for the web service by including a **script** tag with a **src** attribute that points to some script hosted on another domain. You can programmatically create the **script** tags, dynamically creating the URL to which to point the **src** attribute, and passing parameters to the URL via URI query parameters. Web service providers create and host JavaScript code at specific URLs, and return different scripts depending on the URI query parameters. These scripts then execute where they are inserted and work as expected.

The following is an example of JSON/P in the Outlook add-in example.

JavaScript

```
// Dynamically create an HTML SCRIPT element that obtains the details for the
specified video.
function loadVideoDetails(videoIndex) {
    // Dynamically create a new HTML SCRIPT element in the webpage.
    const script = document.createElement("script");
    // Specify the URL to retrieve the indicated video from a feed of a current
list of videos,
    // as the value of the src attribute of the SCRIPT element.
    script.setAttribute("src", "https://gdata.youtube.com/feeds/api/videos/" +
        videos[videoIndex].Id + "?alt=json-in-
script&amp;callback=videoDetailsLoaded");
    // Insert the SCRIPT element at the end of the HEAD section.
    document.getElementsByTagName('head')[0].appendChild(script);
}
```

Exchange and SharePoint provide client-side proxies to enable cross-domain access. In general, same origin policy on an intranet isn't as strict as on the Internet. For more information, see Same Origin Policy Part 1: No Peeking and Addressing same-origin policy limitations in Office Add-ins.

Prevent malicious cross-site scripting

A bad actor could attack the origin of an add-in by entering malicious script through the document or fields in the add-in. A developer should process user input to avoid executing a malicious user's JavaScript within their domain. The following are some good practices to follow to handle user input from a document or mail message, or via fields in an add-in.

- Instead of the DOM property innerHTML, use the innerText and textContent properties where appropriate. Do the following for Internet Explorer and Firefox cross-browser support.

  ```JavaScript
  var text = x.innerText || x.textContent
  ```

  For information about the differences between **innerText** and **textContent**, see Node.textContent. For more information about DOM compatibility across common browsers, see W3C DOM Compatibility - HTML.

- If you must use **innerHTML**, make sure the user's input doesn't contain malicious content before passing it to **innerHTML**. For more information and an example of how to use **innerHTML** safely, see innerHTML property.

- If you are using jQuery, use the .text() method instead of the .html() method.

- Use the toStaticHTML method to remove any dynamic HTML elements and attributes in users' input before passing it to **innerHTML**.

- Use the encodeURIComponent or encodeURI function to encode text that is intended to be a URL that comes from or contains user input.

- See Developing secure add-ins for more best practices to create more secure web solutions.

Prevent "clickjacking"

Because Office Add-ins are rendered in an iframe when running in a browser with Office client applications, use the following tips to minimize the risk of clickjacking -- a technique used by

hackers to fool users into revealing confidential information.

First, identify sensitive actions that your add-in can perform. These include any actions that an unauthorized user could use with malicious intent, such as initiating a financial transaction or publishing sensitive data. For example, your add-in might let the user send a payment to a user-defined recipient.

Second, for sensitive actions, your add-in should confirm with the user before it executes the action. This confirmation should detail what effect the action will have. It should also detail how the user can prevent the action, if necessary, whether by choosing a specific button marked "Don't Allow" or by ignoring the confirmation.

Third, to ensure that no threat actor can hide or mask the confirmation, you should display it outside the context of the add-in (that is, not in an HTML dialog box).

The following are some examples of how you could get confirmation.

- Send an email to the user that contains a confirmation link.
- Send a text message to the user that includes a confirmation code that the user can enter in the add-in.
- Open a confirmation dialog in a new browser window to a page that cannot be iframed. This is typically the pattern that is used by login pages. Use the dialog API to create a new dialog.

Also, ensure that the address you use for contacting the user couldn't have been provided by a threat actor. For example, for payment confirmations use the address on file for the authorized user's account.

Request permission to access device capabilities (applies to Office on the web and new Outlook on Windows)

If an add-in requires access to a user's device capabilities, such as their camera, geolocation, or microphone, the developer must configure it to request permission from the user. 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

To request permission, the add-in must implement the device permission API.

For information on how the user is prompted for permission, 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.

Other security practices

Developers should also take note of the following security practices.

- Developers shouldn't use ActiveX controls in Office Add-ins as ActiveX controls don't support the cross-platform nature of the add-in platform.
- Content and task pane add-ins assume the same SSL settings that the browser uses by default, and allows most content to be delivered only by SSL. Outlook add-ins require all content to be delivered by SSL. Developers must specify in the **<SourceLocation>** element of the add-in manifest a URL that uses HTTPS, to identify the location of the HTML file for the add-in.

  To make sure add-ins aren't delivering content by using HTTP, when testing add-ins, developers should make sure the following settings are selected in **Internet Options** in **Control Panel** and no security warnings appear in their test scenarios.

  - Make sure the security setting, **Display mixed content**, for the **Internet** zone is set to **Prompt**. You can do that by selecting the following in **Internet Options**: on the **Security** tab, select the **Internet** zone, select **Custom level**, scroll to look for **Display mixed content**, and select **Prompt** if it isn't already selected.
  - Make sure **Warn if Changing between Secure and not secure mode** is selected in the **Advanced** tab of the **Internet Options** dialog box.
- To make sure that add-ins don't use excessive CPU core or memory resources and cause any denial of service on a client computer, the add-in platform establishes resource usage limits. As part of testing, developers should verify whether an add-in performs within the resource usage limits.

- Before publishing an add-in, developers should make sure that any personal identifiable information that they expose in their add-in files is secure.
- Developers shouldn't embed keys that they use to access APIs or services from Microsoft and others (such as Bing, Google, or Facebook) directly in the HTML pages of their add-in. Instead, they should create a custom web service or store the keys in some other form of secure web storage that they can then call to pass the key value to their add-in.
- Developers should do the following when submitting an add-in to AppSource.
  - Host the add-in they are submitting on a web server that supports SSL.
  - Produce a statement outlining a compliant privacy policy.
  - Be ready to sign a contractual agreement upon submitting the add-in.

Other than resource usage rules, developers for Outlook add-ins should also make sure their add-ins observe limits for specifying activation rules and using the JavaScript API. For more information, see Limits for activation and JavaScript API for Outlook add-ins.

IT administrators' control

In a corporate setting, IT administrators have ultimate authority over enabling or disabling access to AppSource and any private catalogs.

The management and enforcement of Office settings is done with group policy settings. These are configurable through the Office Deployment Tool, in conjunction with the Office Customization Tool.

Expand table

Setting name	Description
Allow Unsecure web add-ins and Catalogs	Allows users to run non-secure Office Add-ins, which are Office Add-ins that have webpage or catalog locations that are not SSL-secured (https://) and are not in users' Internet zones.
Block Web Add-ins	Allows you to prevent users from running Office Add-ins that use web technologies.
Block the Office Store	Allows you to prevent users from getting or running Office Add-ins that come from AppSource.

To specify permissions to install and manage Outlook add-ins in an organization that uses Exchange Online, configure administrative and user roles in the Exchange admin center. For more information, see Specify the administrators and users who can install and manage add-ins for Outlook in Exchange Online.

See also

- Requesting permissions for API use in add-ins
- Understanding Outlook add-in permissions
- Limits for activation and JavaScript API for Outlook add-ins
- Addressing same-origin policy limitations in Office Add-ins
- Same Origin Policy
- Same Origin Policy Part 1: No Peeking
- Same origin policy for JavaScript
- IE Protect Mode
- Privacy controls for Microsoft 365 Apps

Requesting permissions for API use in add-ins

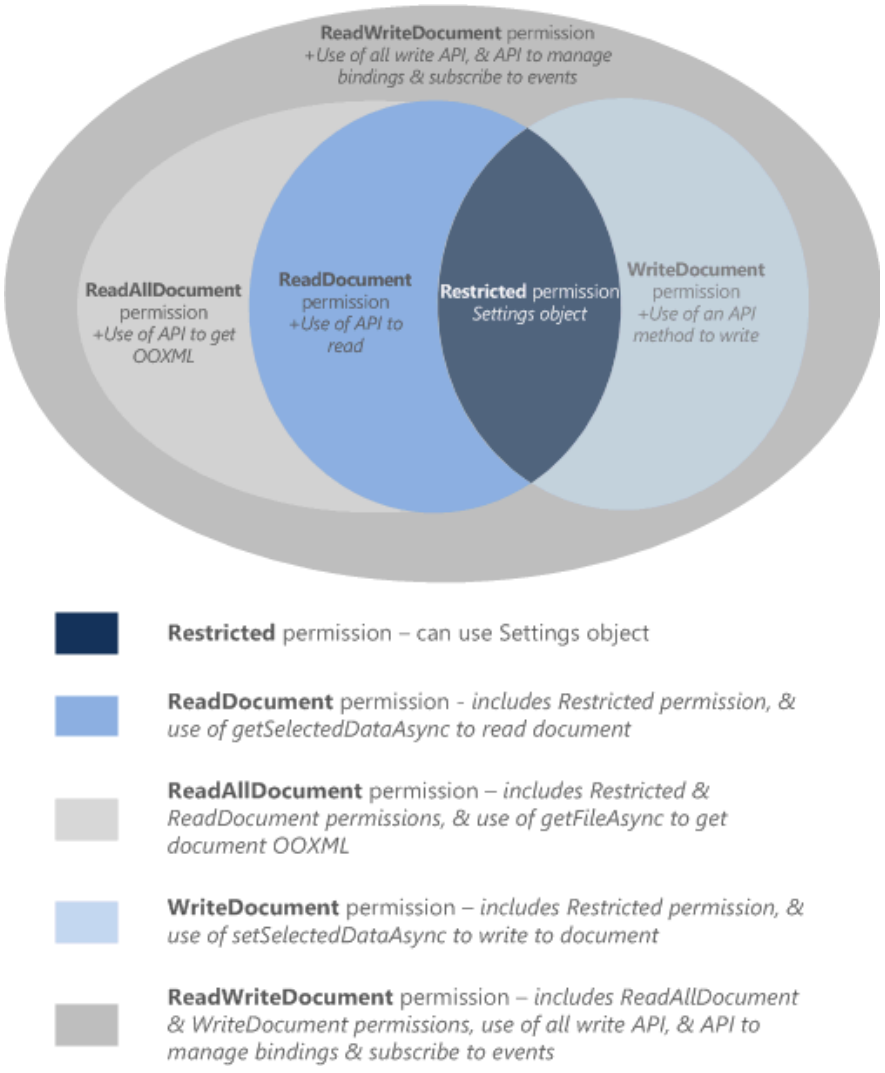
06/17/2025

This article describes the different permission levels that you declare in your add-in's manifest to specify the level of JavaScript API access your add-in requires for its features.

> **Important**
>
> This article applies to only non-Outlook add-ins. To learn about permission levels for Outlook add-ins, see Understanding Outlook add-in permissions.

Permissions model

A five-level JavaScript API access-permissions model provides the basis for privacy and security for users of your add-ins. The following figure shows the five levels of API permissions you can declare in your add-in's manifest.



These permissions specify the subset of the API that the add-in runtime allows your add-in to use when a user inserts, and then activates (trusts) your add-in. To declare the permission level your add-in requires, specify one of the permission values in the manifest. The markup varies depending on the type of manifest.

- **Unified manifest for Microsoft 365**: Use the "authorization.permissions.resourceSpecific" property. The following example requests the **write document** permission, which allows only methods that can write to (but not read) the document.

  JSON

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

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

- **Add-in only manifest**: Use the Permissions element of the manifest. The following example requests the **write document** permission, which allows only methods that can write to (but not read) the document.

  XML

  ```
  <Permissions>WriteDocument</Permissions>
  ```

As a best practice, you should request permissions based on the principle of *least privilege*. That is, you should request permission to access only the minimum subset of the API that your add-in requires to function correctly. For example, if your add-in needs only to read data in a user's document for its features, you should request no more than the **read document** permission.

The following table describes the subsets of the Common and Application-specific JavaScript APIs that are enabled by each permission level.

Expand table

Permission canonical name	Add-in only manifest name	Unified manifest name	Enabled subset of the Application-specific APIs	Enabled subset of the Common APIs
restricted	Restricted	Document.Restricted.User	None	The methods of the Settings object, and the Document.getActiveViewAsync method. This is the minimum permission level that can be requested by an add-in.

Permission canonical name	Add-in only manifest name	Unified manifest name	Enabled subset of the Application-specific APIs	Enabled subset of the Common APIs
read document	ReadDocument	Document.Read.User	All and only APIs that read the document or its properties.	In addition to the API allowed by the **restricted** permission, adds access to the API members necessary to read the document and manage bindings. This includes the use of: • The Document.getSelectedDataAsync method to get the selected text, HTML (Word only), or tabular data, but not the underlying Open Office XML (OOXML) code that contains all of the data in the document. • The Document.getFileAsync method to get all of the text in the document, but not the underlying OOXML binary copy of the document • The Binding.getDataAsync method for reading bound data in the document. • The addFromNamedItemAsync, addFromPromptAsync, and addFromSelectionAsync methods of the Bindings object for creating bindings in the document. • The getAllAsync, getByIdAsync, and releaseByIdAsync methods of the Bindings object for accessing and removing bindings in the document. • The Document.getFilePropertiesAsync method to access document file properties, such as the URL of the document. • The Document.goToByIdAsync method to navigate to named objects and locations in the document. • For task pane add-ins for Project, all of the "get" methods of the ProjectDocument object.

Permission canonical name	Add-in only manifest name	Unified manifest name	Enabled subset of the Application-specific APIs	Enabled subset of the Common APIs
read all document	ReadAllDocument	Document.ReadAll.User	Same as **read document**.	In addition to the API allowed by the **restricted** and **read document** permissions, allows the following additional access to document data. • The Document.getSelectedDataAsync and Document.getFileAsync methods can access the underlying OOXML code of the document (which in addition to the text may include formatting, links, embedded graphics, comments, revisions, and so forth).
write document	WriteDocument	Document.Write.User	All and only APIs that write to the document or its properties.	In addition to the API allowed by the **restricted** permission, adds access to the following API members. • The Document.setSelectedDataAsync method to write to the user's selection in the document.
read/write document	ReadWriteDocument	Document.ReadWrite.User	All Application-specfic APIs, including those that subscribe to events.	In addition to the API allowed by the **restricted**, **read document**, **read all document**, and **write document** permissions, includes access to all remaining API supported by add-ins, including methods for subscribing to events. You must declare the **read/write document** permission to access these additional API members: • The Binding.setDataAsync method for writing to bound regions of the document • The TableBinding.addRowsAsync method for adding rows to bound tables. • The TableBinding.addColumnsAsync method for adding columns to bound tables. • The TableBinding.deleteAllDataValuesAsync method for deleting all data in a bound table. • The setFormatsAsync, clearFormatsAsync, and setTableOptionsAsync methods of the TableBinding object for setting formatting and options on bound tables.

Permission canonical name	Add-in only manifest name	Unified manifest name	Enabled subset of the Application-specific APIs	Enabled subset of the Common APIs
				• All of the members of the CustomXmlNode, CustomXmlPart, CustomXmlParts, and CustomXmlPrefixMappings objects • All of the methods for subscribing to the events supported by add-ins, specifically the `addHandlerAsync` and `removeHandlerAsync` methods of the Binding, CustomXmlPart, Document, ProjectDocument, and Settings objects.

See also

- Privacy and security for Office Add-ins

Develop your Office Add-in to work with ITP when using third-party cookies

Article • 08/08/2024

If your Office Add-in requires third-party cookies, those cookies are blocked if the Runtime that loaded your add-in uses Intelligent Tracking Prevention (ITP). You may be using third-party cookies to authenticate users, or for other scenarios, such as storing settings.

If your Office Add-in and website must rely on third-party cookies, use the following steps to work with ITP.

1. Set up OAuth 2.0 Authorization so that the authenticating domain (in your case, the third-party that expects cookies) forwards an authorization token to your website. Use the token to establish a first-party session with a server-set Secure and HttpOnly cookie.
2. Use the Storage Access API so that the third-party can request permission to get access to its first-party cookies. Current versions of Office on Mac and Office on the web both support this API.

> **Note**
>
> If you're using cookies for purposes other than authentication, consider using `localStorage` for your scenario.
>
> However, note that starting in Version 115 of Chromium-based browsers, such as Chrome and Edge, storage partitioning is enabled 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.

The following code sample shows how to use the Storage Access API.

JavaScript

```
function displayLoginButton() {
  const button = createLoginButton();
  button.addEventListener("click", function(ev) {
    document.requestStorageAccess().then(function() {
      authenticateWithCookies();
    }).catch(function() {
      // User must have previously interacted with this domain loaded in a
```

```
top frame.
      // Also you should have previously written a cookie when domain was
loaded in the top frame.
      console.error("User cancelled or requirements were not met.");
    });
  });
}

if (document.hasStorageAccess) {
  document.hasStorageAccess().then(function(hasStorageAccess) {
    if (!hasStorageAccess) {
      displayLoginButton();
    } else {
      authenticateWithCookies();
    }
  });
} else {
    authenticateWithCookies();
}
```

About ITP and third-party cookies

Third-party cookies are cookies that are loaded in an iframe, where the domain is different from the top level frame. ITP could affect complex authentication scenarios, where a pop-up dialog is used to enter credentials and then the cookie access is needed by an add-in iframe to complete the authentication flow. ITP could also affect silent authentication scenarios, where you have previously used a pop-up dialog to authenticate, but subsequent use of the add-in tries to authenticate through a hidden iframe.

When developing Office Add-ins on Mac, access to third-party cookies is blocked by the MacOS Big Sur SDK. This is because WKWebView ITP is enabled by default on the Safari browser, and WKWebView blocks all third-party cookies. Office on Mac Version 16.44 (20121301) or later is integrated with the MacOS Big Sur SDK.

In the Safari browser, end users can toggle the **Prevent cross-site tracking** checkbox under **Preference** > **Privacy** to turn off ITP. However, ITP can't be turned off for the embedded WKWebView control.

Google Chrome third-party cookie support

Google Chrome is working to give users more control of their browsing experience. Users will be able to block third-party cookies in their Chrome browser. This will prevent your add-in from using any such cookies. This may cause issues when the add-in authenticates the user, such as multiple sign-on requests or errors.

For improved authentication experiences, see Using device state for an improved SSO experience on browsers with blocked third-party cookies.

For more information about the Google Chrome rollout, see A new path for Privacy Sandbox on the web.

See also

- Handle ITP in Safari and other browsers where third-party cookies are blocked
- Tracking Prevention in WebKit
- Chrome's "Privacy Sandbox"
- Introducing the Storage Access API

Addressing same-origin policy limitations in Office Add-ins

Article • 01/09/2024

The same-origin policy enforced by the browser or webview control prevents a script loaded from one domain from getting or manipulating properties of a webpage from another domain. This means that, by default, the domain of a requested URL must be the same as the domain of the current webpage. For example, this policy will prevent a webpage in one domain from making XmlHttpRequest web-service calls to a domain other than the one where it is hosted.

Because Office Add-ins are hosted in a webview control, the same-origin policy applies to script running in their web pages as well.

The same-origin policy can be an unnecessary handicap in many situations, such as when a web application hosts content and APIs across multiple subdomains. There are a few common techniques for securely overcoming same-origin policy enforcement. This article can only provide the briefest introduction to some of them. Please use the links provided to get started in your research of these techniques.

Use JSONP for anonymous access

One way to overcome same-origin policy limitations is to use JSONP to provide a proxy for the web service. You do this by including a `script` tag with a `src` attribute that points to some script hosted on any domain. You can programmatically create the `script` tags, dynamically create the URL to point the `src` attribute to, and then pass parameters to the URL via URI query parameters. Web service providers create and host JavaScript code at specific URLs, and return different scripts depending on the URI query parameters. These scripts then execute where they are inserted and work as expected.

The following is an example of JSONP that uses a technique that will work in any Office Add-in.

JavaScript

```
// Dynamically create an HTML SCRIPT element that obtains the details for
the specified video.
function loadVideoDetails(videoIndex) {
    // Dynamically create a new HTML SCRIPT element in the webpage.
    const script = document.createElement("script");
    // Specify the URL to retrieve the indicated video from a feed of a
current list of videos,
```

```
    // as the value of the src attribute of the SCRIPT element.
    script.setAttribute("src", "https://gdata.youtube.com/feeds/api/videos/"
+
        videos[videoIndex].Id + "?alt=json-in-
script&amp;callback=videoDetailsLoaded");
    // Insert the SCRIPT element at the end of the HEAD section.
    document.getElementsByTagName('head')[0].appendChild(script);
}
```

Implement server-side code using a token-based authorization scheme

Another way to address same-origin policy limitations is to provide server-side code that uses OAuth 2.0 flows to enable one domain to get authorized access to resources hosted on another.

Use cross-origin resource sharing (CORS)

To learn more about cross-origin resource sharing, see the many resources available on the web, such as Cross-Origin Resource Sharing (CORS).

> **Note**
>
> For information on how to use CORS in an Outlook add-in that implements event-based activation or integrated spam reporting (preview), see **Use single sign-on (SSO) or cross-origin resource sharing (CORS) in your event-based or spam-reporting Outlook add-in**.

Build your own proxy using IFRAME and POST MESSAGE (Cross-Window Messaging)

For an example of how to build your own proxy using IFRAME and POST MESSAGE, see Cross-Window Messaging.

See also

- Privacy and security for Office Add-ins

Wildcard trusted domains

Article • 05/07/2025

Besides its own domain, an add-in can access resources in certain other domains such as authentication points for major identity providers and in any domain listed in the manifest. The latter domains are specified in the AppDomains element of the add-in only manifest or the "validDomains" property of the unified manifest. Wildcards aren't allowed in the add-in only manifest. They are allowed in the unified manifest because some Teams apps and other Microsoft 365 apps honor them; but Office Add-ins don't honor `"validDomains"` that contain wildcards.

Windows administrators can make Office Add-ins, *running on Windows only*, honor domains that include a wildcard by setting the **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\WEF\AllowedAppDomains** registry key with the domain. The following is an example.

```
[HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\WEF\AllowedAppDomains]
"AppDomain1"="https://*.contoso.com"
```

Administrators can use a *.reg file to do automate the process. The following is an example of such a file.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\WEF\AllowedAppDomains]
"AppDomain1"="https://*.europe.contoso.com"

[HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\WEF\AllowedAppDomains]
"AppDomain2"="https://*.africa.contoso.com"
```

> **Note**
>
> - The domains are honored only in add-ins running on Windows desktop versions of Office. They aren't honored when an add-in is running in Office on the web even on computers where the registry change has been made.
> - The registry setting affects *all* add-ins running on the computer: they all trust the domains in the registry key.

Persist add-in state and settings

08/28/2025

Office Add-ins are essentially web applications running in the stateless environment of a browser iframe or a webview control. (For brevity hereafter, this article uses "browser control" to mean "browser or webview control".) When in use, your add-in may need to persist data to maintain the continuity of certain operations or features across sessions. For example, your add-in may have custom settings or other values that it needs to save and reload the next time it's initialized, such as a user's preferred view or default location. To do that, you can:

- Use techniques provided by the underlying browser control.
- Use the application-specific Office JavaScript APIs for Excel, Word, and Outlook that store data.

If you need to persist state across documents, such as tracking user preferences across any documents they open, you'll need to use a different approach. For example, you could use SSO to obtain the user identity, and then save the user ID and their settings to an online database.

Browser storage

Persist data across add-in instances with tools from the underlying browser control, such as browser cookies or HTML5 web storage (localStorage or sessionStorage).

Some browsers or the user's browser settings may block browser-based storage techniques. You should test for availability as documented in Using the Web Storage API.

Storage partitioning

As a best practice, any private data should be stored in partitioned `localStorage`. Office.context.partitionKey provides a key for use with local storage. This ensures that data stored in local storage is only available in the same context.

> **Note**
>
> The partition key is undefined in environments without partitioning, such as the webview controls for Office on Windows. Where it is defined, the partition key is a hash of the following two domains.
>
> - The domain that the top-level browser window is at, such as `excel.cloud.microsoft` in the case of Excel on the web.
> - The domain of the add-in, such as `myAddin.contoso.com`.

So, each of the following would be a different partition:

- `excel.cloud.microsoft` + `myAddin.contoso.com`
- `word.cloud.microsoft` + `myAddin.contoso.com`
- `word.cloud.microsoft` + `myOtherAddin.contoso.com`

The following example shows how to use the partition key with `localStorage`.

JavaScript

```
// Store the value "Hello" in local storage with the key "myKey1".
setInLocalStorage("myKey1", "Hello");

// ...

// Retrieve the value stored in local storage under the key "myKey1".
const message = getFromLocalStorage("myKey1");
console.log(message);

// ...

function setInLocalStorage(key: string, value: string) {
  const myPartitionKey = Office.context.partitionKey;

  // Check if local storage is partitioned.
  // If so, use the partition to ensure the data is only accessible by your add-
in.
  if (myPartitionKey) {
    localStorage.setItem(myPartitionKey + key, value);
  } else {
    localStorage.setItem(key, value);
  }
}

function getFromLocalStorage(key: string) {
  const myPartitionKey = Office.context.partitionKey;

  // Check if local storage is partitioned.
  if (myPartitionKey) {
    return localStorage.getItem(myPartitionKey + key);
  } else {
    return localStorage.getItem(key);
  }
}
```

Starting in Version 115 of Chromium-based browsers, such as Chrome and Edge, storage partitioning is enabled to prevent specific side-channel cross-site tracking (see also Microsoft Edge browser policies). Similar to the Office key-based partitioning, data stored by storage

APIs, such as local storage, is only available to contexts with the same origin and the same top-level site.

Application-specific settings and persistence

Excel, Word, and Outlook provide application-specific APIs to save settings and other data. Use these instead of the Common APIs mentioned later in this article so that your add-in follows consistent patterns and is optimized for the targeted application.

Settings in Excel and Word

The application-specific JavaScript APIs for Excel and for Word also provide access to the custom settings. Settings are unique to a single Excel file and add-in pairing. For more information, see Excel.SettingCollection and Word.SettingCollection.

The following example shows how to create and access a setting in Excel. The process is functionally equivalent in Word, which uses Document.settings instead of `Workbook.settings`.

JavaScript

```
await Excel.run(async (context) => {
    const settings = context.workbook.settings;
    settings.add("NeedsReview", true);
    const needsReview = settings.getItem("NeedsReview");
    needsReview.load("value");

    await context.sync();
    console.log("Workbook needs review : " + needsReview.value);
});
```

Custom XML data in Excel and Word

The Open XML **.xlsx** and **.docx** file formats let your add-in embed custom XML data in the Excel workbook or Word document. This data persists with the file, independent of the add-in.

A Word.Document and Excel.Workbook contain a `CustomXmlPartCollection`, which is a list of `CustomXmlParts`. These give access to the XML strings and a corresponding unique ID. By storing these IDs as settings, your add-in can maintain the keys to its XML parts between sessions.

The following samples show how to use custom XML parts with an Excel workbook. The first code block demonstrates how to embed XML data. It stores a list of reviewers, then uses the workbook's settings to save the XML's `id` for future retrieval. The second block shows how to

access that XML later. The "ContosoReviewXmlPartId" setting is loaded and passed to the workbook's `customXmlParts`. The XML data is then printed to the console. The process is functionally equivalent in Word, which uses Document.customXmlParts instead of `Workbook.customXmlParts`.

JavaScript

```
await Excel.run(async (context) => {
    // Add reviewer data to the document as XML
    const originalXml = "<Reviewers xmlns='http://schemas.contoso.com/review/1.0'>
<Reviewer>Juan</Reviewer><Reviewer>Hong</Reviewer><Reviewer>Sally</Reviewer>
</Reviewers>";
    const customXmlPart = context.workbook.customXmlParts.add(originalXml);
    customXmlPart.load("id");
    await context.sync();

    // Store the XML part's ID in a setting
    const settings = context.workbook.settings;
    settings.add("ContosoReviewXmlPartId", customXmlPart.id);
});
```

> **Note**
>
> `CustomXMLPart.namespaceUri` is only populated if the top-level custom XML element contains the `xmlns` attribute.

Custom properties in Excel and Word

The Excel.DocumentProperties.custom and Word.DocumentProperties.customProperties properties represent collections of key-value pairs for user-defined properties. The following Excel example shows how to create a custom property named **Introduction** with the value "Hello", then retrieve it.

JavaScript

```
await Excel.run(async (context) => {
    const customDocProperties = context.workbook.properties.custom;
    customDocProperties.add("Introduction", "Hello");
    await context.sync();
});

// ...

await Excel.run(async (context) => {
    const customDocProperties = context.workbook.properties.custom;
    const customProperty = customDocProperties.getItem("Introduction");
```

```
    customProperty.load(["key", "value"]);
    await context.sync();

    console.log("Custom key  : " + customProperty.key); // "Introduction"
    console.log("Custom value : " + customProperty.value); // "Hello"
});
```

> **Tip**
>
> In Excel, custom properties can also be set at the worksheet level with the **Worksheet.customProperties** property. These are similar to document-level custom properties, except that the same key can be repeated across different worksheets.

How to save settings in an Outlook add-in

For information about how to save settings in an Outlook add-in, see Get and set add-in metadata for an Outlook add-in and Get and set internet headers on a message in an Outlook add-in.

Common API settings and persistence

The Common APIs provide objects to save add-in state across sessions. The saved settings values are associated with the Id of the add-in that created them. Internally, the data accessed with the `Settings`, `CustomProperties`, or `RoamingSettings` objects is stored as a serialized JavaScript Object Notation (JSON) object that contains name/value pairs. The name (key) for each value must be a `string`, and the stored value can be a JavaScript `string`, `number`, `date`, or `object`, but not a **function**.

This example of the property bag structure contains three defined **string** values named `firstName`, `location`, and `defaultView`.

JSON

```
{
    "firstName":"Erik",
    "location":"98052",
    "defaultView":"basic"
}
```

After the settings property bag is saved during the previous add-in session, it can be loaded when the add-in is initialized or at any point after that during the add-in's current session. During the session, the settings are managed in entirely in memory using the `get`, `set`, and

`remove` methods of the object that corresponds to the kind of settings you're creating (**Settings**, **CustomProperties**, or **RoamingSettings**).

> **ⓘ Important**
>
> To persist any additions, updates, or deletions made during the add-in's current session to the storage location, you must call the `saveAsync` method of the corresponding object used to work with that kind of settings. The `get`, `set`, and `remove` methods operate only on the in-memory copy of the settings property bag. If your add-in is closed without calling `saveAsync`, any changes made to settings during that session will be lost.

How to save add-in state and settings per document for content and task pane add-ins

To persist state or custom settings of a content or task pane add-in for Word, Excel, or PowerPoint, use the Settings object and its methods. The property bag created with the methods of the `Settings` object are available only to the instance of the content or task pane add-in that created it, and only from the document in which it is saved.

The `Settings` object is automatically loaded as part of the Document object, and is available when the task pane or content add-in is activated. After the `Document` object is instantiated, you can access the `Settings` object with the settings property of the `Document` object. During the lifetime of the session, you can use the `Settings.get`, `Settings.set`, and `Settings.remove` methods to read, write, or remove persisted settings and add-in state from the in-memory copy of the property bag.

Because the set and remove methods operate against only the in-memory copy of the settings property bag, to save new or changed settings back to the document the add-in is associated with, you must call the Settings.saveAsync method.

Create or update a setting value

The following code example shows how to use the Settings.set method to create a setting called `'themeColor'` with a value `'green'`. The first parameter of the set method is the case-sensitive *name* (Id) of the setting to set or create. The second parameter is the *value* of the setting.

JavaScript

```
Office.context.document.settings.set('themeColor', 'green');
```

The setting with the specified name is created if it doesn't already exist, or its value is updated if it does exist. Use the `Settings.saveAsync` method to persist the new or updated settings to the document.

Get the value of a setting

The following example shows how use the Settings.get method to get the value of a setting called "themeColor". The only parameter of the `get` method is the case-sensitive *name* of the setting.

JavaScript

```
write('Current value for mySetting: ' +
Office.context.document.settings.get('themeColor'));

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

The `get` method returns the value that was previously saved for the setting *name* that was passed in. If the setting doesn't exist, the method returns **null**.

Remove a setting

The following example shows how to use the Settings.remove method to remove a setting with the name "themeColor". The only parameter of the `remove` method is the case-sensitive *name* of the setting.

JavaScript

```
Office.context.document.settings.remove('themeColor');
```

Nothing will happen if the setting doesn't exist. Use the `Settings.saveAsync` method to persist removal of the setting from the document.

Save your settings

To save any additions, changes, or deletions your add-in made to the in-memory copy of the settings property bag during the current session, you must call the Settings.saveAsync method to store them in the document. The only parameter of the `saveAsync` method is *callback*, which is a callback function with a single parameter.

JavaScript

```
Office.context.document.settings.saveAsync(function (asyncResult) {
    if (asyncResult.status == Office.AsyncResultStatus.Failed) {
        write('Settings save failed. Error: ' + asyncResult.error.message);
    } else {
        write('Settings saved.');
    }
});
// Function that writes to a div with id='message' on the page.
function write(message){
    document.getElementById('message').innerText += message;
}
```

The anonymous function passed into the `saveAsync` method as the *callback* parameter is executed when the operation is completed. The *asyncResult* parameter of the callback provides access to an `AsyncResult` object that contains the status of the operation. In the example, the function checks the `AsyncResult.status` property to see if the save operation succeeded or failed, and then displays the result in the add-in's page.

How to save custom XML to the document

A custom XML part is an available storage option for when you want to store information that has a structured character or need the data to be accessible across instances of your add-in. Note that data stored this way can also be accessed by other add-ins. You can persist custom XML markup in a task pane add-in for Word (and for Excel and Word using application-specific API as mentioned in the previous paragraph). In Word, you can use the CustomXmlPart object and its methods. The following code creates a custom XML part and displays its ID and then its content in divs on the page. Note that there must be an `xmlns` attribute in the XML string.

JavaScript

```
function createCustomXmlPart() {
    const xmlString = "<Reviewers xmlns='http://schemas.contoso.com/review/1.0'>
<Reviewer>Juan</Reviewer><Reviewer>Hong</Reviewer><Reviewer>Sally</Reviewer>
</Reviewers>";
    Office.context.document.customXmlParts.addAsync(xmlString,
        (asyncResult) => {
            $("#xml-id").text("Your new XML part's ID: " + asyncResult.value.id);
            asyncResult.value.getXmlAsync(
                (asyncResult) => {
                    $("#xml-blob").text(asyncResult.value);
                }
            );
        }
```

```
    );
}
```

To retrieve a custom XML part, use the getByIdAsync method, but the ID is a GUID that is generated when the XML part is created, so you can't know when coding what the ID is. For that reason, it's a good practice when creating an XML part to immediately store the ID of the XML part as a setting and give it a memorable key. The following method shows how to do this.

JavaScript

```
function createCustomXmlPartAndStoreId() {
   const xmlString = "<Reviewers xmlns='http://schemas.contoso.com/review/1.0'>
<Reviewer>Juan</Reviewer><Reviewer>Hong</Reviewer><Reviewer>Sally</Reviewer>
</Reviewers>";
   Office.context.document.customXmlParts.addAsync(xmlString,
       (asyncResult) => {
           Office.context.document.settings.set('ReviewersID',
asyncResult.value.id);
           Office.context.document.settings.saveAsync();
       }
   );
}
```

The following code shows how to retrieve the XML part by first getting its ID from a setting.

JavaScript

```
function getReviewers() {
   const reviewersXmlId = Office.context.document.settings.get('ReviewersID');
   Office.context.document.customXmlParts.getByIdAsync(reviewersXmlId,
       (asyncResult) => {
           asyncResult.value.getXmlAsync(
               (asyncResult) => {
                   $("#xml-blob").text(asyncResult.value);
               }
           );
       }
   );
}
```

See also

- Understanding the Office JavaScript API
- Outlook add-ins
- Get and set add-in metadata for an Outlook add-in

- Get and set internet headers on a message in an Outlook add-in
- Excel-Add-in-JavaScript-PersistCustomSettings

Overview of authentication and authorization in Office Add-ins

Article • 06/23/2023

Office Add-ins allow anonymous access by default, but you can require users to sign in to use your add-in with a Microsoft account, a Microsoft 365 Education or work account, or other common account. This task is called user authentication because it enables the add-in to know who the user is.

Your add-in can also get the user's consent to access their Microsoft Graph data (such as their Microsoft 365 profile, OneDrive files, and SharePoint data) or data in other external sources such as Google, Facebook, LinkedIn, SalesForce, and GitHub. This task is called add-in (or app) authorization, because it's the *add-in* that is being authorized, not the user.

Key resources for authentication and authorization

This documentation explains how to build and configure Office Add-ins to successfully implement authentication and authorization. However, many concepts and security technologies mentioned are outside the scope of this documentation. For example, general security concepts such as OAuth flows, token caching, or identity management are not explained here. This documentation also doesn't document anything specific to Microsoft Azure or the Microsoft identity platform. We recommend you refer to the following resources if you need more information in those areas.

- Microsoft identity platform
- Microsoft identity platform support and help options for developers
- OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform

SSO scenarios

Using Single Sign-on (SSO) is convenient for the user because they only have to sign in once to Office. They don't need to sign in separately to your add-in. SSO isn't supported on all versions of Office, so you'll still need to implement an alternative sign-in approach, by using the Microsoft identity platform. For more information on supported Office versions, see Identity API requirement sets

Get the user's identity through SSO

Often your add-in only needs the user's identity. For example, you may just want to personalize your add-in and display the user's name on the task pane. Or you might want a unique ID to associate the user with their data in your database. This can be accomplished by just getting the access token for the user from Office.

To get the user's identity through SSO, call the getAccessToken method. The method returns an access token that is also an identity token containing several claims that are unique to the current signed in user, including `preferred_username`, `name`, `sub`, and `oid`. For more information on these properties, see Microsoft identity platform ID tokens. For an example of the token returned by getAccessToken, see Example access token.

If the user isn't signed in, Office will open a dialog box and use the Microsoft identity platform to request the user to sign in. Then the method will return an access token, or throw an error if unable to sign in the user.

In a scenario where you need to store data for the user, refer to Microsoft identity platform ID tokens for information about how to get a value from the token to uniquely identify the user. Use that value to look up the user in a user table or user database that you maintain. Use the database to store user-relative information such as the user's preferences or the state of the user's account. Since you're using SSO, your users don't sign-in separately to your add-in, so you don't need to store a password for the user.

Before you begin implementing user authentication with SSO, be sure that you're thoroughly familiar with the article Enable single sign-on for Office Add-ins.

Access your Web APIs through SSO

If your add-in has server-side APIs that require an authorized user, call the getAccessToken method to get an access token. The access token provides access to your own web server (configured through a Microsoft Azure app registration). When you call APIs on your web server, you also pass the access token to authorize the user.

The following code shows how to construct an HTTPS GET request to the add-in's web server API to get some data. The code runs on the client side, such as in a task pane. It first gets the access token by calling `getAccessToken`. Then it constructs an AJAX call with the correct authorization header and URL for the server API.

JavaScript

```
function getOneDriveFileNames() {
```

```
    let accessToken = await Office.auth.getAccessToken();

    $.ajax({
        url: "/api/data",
        headers: { "Authorization": "Bearer " + accessToken },
        type: "GET"
    })
        .done(function (result) {
            //... work with data from the result...
        });
}
```

The following code shows an example /api/data handler for the REST call from the previous code example. The code is ASP.NET code running on a web server. The `[Authorize]` attribute will require that a valid access token is passed from the client, or it'll return an error to the client.

C#

```
    [Authorize]
    // GET api/data
    public async Task<HttpResponseMessage> Get()
    {
        //... obtain and return data to the client-side code...
    }
```

Access Microsoft Graph through SSO

In some scenarios, not only do you need the user's identity, but you also need to access Microsoft Graph resources on behalf of the user. For example, you may need to send an email, or create a chat in Teams on behalf of the user. These actions and more can be accomplished through Microsoft Graph. You'll need to follow these steps:

1. Get the access token for the current user through SSO by calling getAccessToken. If the user isn't signed in, Office will open a dialog box and sign in the user with the Microsoft identity platform. After the user signs in, or if the user is already signed in, the method returns an access token.
2. Pass the access token to your server-side code.
3. On the server-side, use the OAuth 2.0 On-Behalf-Of flow to exchange the access token for a new access token containing the necessary delegated user identity and permissions to call Microsoft Graph.

> ⓘ **Note**

For best security to avoid leaking the access token, always perform the On-Behalf-Of flow on the server-side. Call Microsoft Graph APIs from your server, not the client. Don't return the access token to the client-side code.

Before you begin implementing SSO to access Microsoft Graph in your add-in, be sure that you're thoroughly familiar with the following articles.

- Enable single sign-on for Office Add-ins
- Authorize to Microsoft Graph with SSO

You should also read at least one of the following articles that'll walk you through building an Office Add-in to use SSO and access Microsoft Graph. Even if you don't carry out the steps, they contain valuable information about how you implement SSO and the On-Behalf-Of flow.

- Create an ASP.NET Office Add-in that uses single sign-on which walks you through the sample at Office Add-in ASP.NET SSO.
- Create an Node.js Office Add-in that uses single sign-on which walks you through the sample at Office Add-in NodeJS SSO.

Non-SSO scenarios

In some scenarios, you may not want to use SSO. For example, you may need to authenticate using a different identity provider than the Microsoft identity platform. Also, SSO isn't supported in all scenarios. For example, older versions of Office don't support SSO. In this case, you'd need to fall back to an alternate authentication system for your add-in.

Authenticate with the Microsoft identity platform

Your add-in can sign in users using the Microsoft identity platform as the authentication provider. Once you've signed in the user, you can then use the Microsoft identity platform to authorize the add-in to Microsoft Graph or other services managed by Microsoft. Use this approach as an alternate sign-in method when SSO through Office is unavailable. Also, there are scenarios in which you want to have your users sign in to your add-in separately even when SSO is available; for example, if you want them to have the option of signing in to the add-in with a different ID from the one with which they're currently signed in to Office.

It's important to note that the Microsoft identity platform doesn't allow its sign-in page to open in an iframe. When an Office Add-in is running in *Office on the web*, the task

pane is an iframe. This means that you'll need to open the sign-in page by using a dialog box opened with the Office dialog API. This affects how you use authentication helper libraries. For more information, see Authentication with the Office dialog API.

For information about implementing authentication with the Microsoft identity platform, see Microsoft identity platform (v2.0) overview. The documentation contains many tutorials and guides, as well as links to relevant samples and libraries. As explained in Authentication with the Office dialog API, you may need to adjust the code in the samples to run in the Office dialog box.

Access to Microsoft Graph without SSO

You can get authorization to Microsoft Graph data for your add-in by obtaining an access token to Microsoft Graph from the Microsoft identity platform. You can do this without relying on SSO through Office (or if SSO failed or isn't supported). For more information, see Access to Microsoft Graph without SSO which has more details and links to samples.

Access to non-Microsoft data sources

Popular online services, including Google, Facebook, LinkedIn, SalesForce, and GitHub, let developers give users access to their accounts in other applications. This gives you the ability to include these services in your Office Add-in. For an overview of the ways that your add-in can do this, see Authorize external services in your Office Add-in.

> **Important**
>
> Before you begin coding, find out if the data source allows its sign-in page to open in an iframe. When an Office Add-in is running in *Office on the web*, the task pane is an iframe. If the data source doesn't allow its sign-in page to open in an iframe, then you'll need to open the sign-in page in a dialog box opened with the Office dialog API. For more information, see **Authentication with the Office dialog API**.

See also

- Microsoft identity platform documentation
- Microsoft identity platform access tokens
- OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform
- Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow
- JSON web token (JWT)

- JSON web token viewer