
Thunderbird WebExtension APIs

Oct 19, 2020

Contents

1	accounts	3
2	addressBooks	5
3	browserAction	9
4	cloudFile	15
5	commands	19
6	compose	21
7	composeAction	27
8	composeScripts	33
9	contacts	35
10	folders	39
11	mailingLists	41
12	mailTabs	45
13	menus	51
14	messageDisplay	57
15	messageDisplayAction	59
16	messageDisplayScripts	65
17	messages	67
18	tabs	73
19	windows	81
20	Changes to APIs	87

21 How To	95
22 Where To Get Help	101

Thunderbird WebExtensions are very similar to those of Firefox. These documents assume you have some familiarity with building a WebExtension for Firefox. If not, it is highly recommended to begin by reading some of the [MDN documentation on the subject](#).

WebExtension APIs are asynchronous, that is, they return a [Promise](#) object which resolves when ready. See [Using Promises](#) for more information about Promises.

The documents were generated automatically from the schema documents at [mail/components/extensions/schemas](#).

Note: This documentation is for pre-release versions of Thunderbird. See the [“78” version](#) for Thunderbird 78, or the [“68” version](#) for Thunderbird 68. For any problems or feature requests please [file a bug](#).

Note: In Thunderbird, all WebExtension API can be accessed through the *browser.** namespace, as with Firefox, but also through the *messenger.** namespace, which is a better fit for Thunderbird.

The accounts API first appeared in Thunderbird 66 (see [bug 1488176](#)).

1.1 Permissions

- `accountsRead` “See your mail accounts and their folders”

Note: The permission `accountsRead` is required to use `accounts`.

1.2 Functions

1.2.1 `list()`

Returns all mail accounts.

Returns a `Promise` fulfilled with:

- array of *MailAccount*

1.2.2 `get(accountId)`

Returns details of the requested account, or null if it doesn't exist.

- `accountId` (string)

Returns a `Promise` fulfilled with:

- *MailAccount*

1.2.3 setDefaultIdentity(accountId, identityId)

Added in Thunderbird 76

Sets the default identity for an account.

- `accountId` (string)
- `identityId` (string)

1.3 Types

1.3.1 MailAccount

object:

- `folders` (array of *MailFolder*) The folders for this account.
- `id` (string) A unique identifier for this account.
- `identities` (array of *MailIdentity*) The identities associated with this account. The default identity is listed first, others in no particular order. *Added in Thunderbird 76*
- `name` (string) The human-friendly name of this account.
- `type` (string) What sort of account this is, e.g. `imap`, `nntp`, or `pop3`.

1.3.2 MailIdentity

Added in Thunderbird 76

object:

- `accountId` (string) The id of the *MailAccount* this identity belongs to.
- `email` (string) The user's email address as used when messages are sent from this identity.
- `id` (string) A unique identifier for this identity.
- `label` (string) A user-defined label for this identity.
- `name` (string) The user's name as used when messages are sent from this identity.
- `organization` (string) The organization associated with this identity.
- `replyTo` (string) The reply-to email address associated with this identity.

The address books API, also including the *contacts* and *mailingLists* namespaces, first appeared in Thunderbird 64. The [Address Books](#) sample extension uses this API.

2.1 Permissions

- addressBooks “Read and modify your address books and contacts”

Note: The permission `addressBooks` is required to use `addressBooks`.

2.2 Functions

2.2.1 `openUI()`

Opens the address book user interface.

2.2.2 `closeUI()`

Closes the address book user interface.

2.2.3 `list([complete])`

Gets a list of the user’s address books, optionally including all contacts and mailing lists.

- `[complete]` (boolean) If set to true, results will include contacts and mailing lists for each address book.

Returns a [Promise](#) fulfilled with:

- array of *AddressBookNode*

2.2.4 `get(id, [complete])`

Gets a single address book, optionally including all contacts and mailing lists.

- `id` (string)
- `[complete]` (boolean) If set to true, results will include contacts and mailing lists for this address book.

Returns a `Promise` fulfilled with:

- *AddressBookNode*

2.2.5 `create(properties)`

Creates a new, empty address book.

- `properties` (object)
 - `name` (string)

Returns a `Promise` fulfilled with:

- string The ID of the new address book.

2.2.6 `update(id, properties)`

Renames an address book.

- `id` (string)
- `properties` (object)
 - `name` (string)

2.2.7 `delete(id)`

Removes an address book, and all associated contacts and mailing lists.

- `id` (string)

2.3 Events

2.3.1 `onCreated(node)`

Fired when an address book is created.

- `node` (*AddressBookNode*)

2.3.2 `onUpdated(node)`

Fired when an address book is renamed.

- `node` (*AddressBookNode*)

2.3.3 onDeleted(id)

Fired when an addressBook is deleted.

- `id` (string)

2.4 Types

2.4.1 AddressBookNode

A node representing an address book.

object:

- `id` (string) The unique identifier for the node. IDs are unique within the current profile, and they remain valid even after the program is restarted.
- `name` (string)
- `type` (*NodeType*) Always set to `addressBook`.
- `[contacts]` (array of *ContactNode*) A list of contacts held by this node's address book or mailing list.
- `[mailingLists]` (array of *MailingListNode*) A list of mailingLists in this node's address book.
- `[parentId]` (string) The `id` of the parent object.
- `[readOnly]` (boolean) Indicates if the object is read-only. Currently this returns false in all cases, as read-only address books are ignored by the API.

2.4.2 NodeType

Indicates the type of a Node, which can be one of `addressBook`, `contact`, or `mailingList`.

string

Values for `NodeType`:

- `addressBook`
- `contact`
- `mailingList`

The `browserAction` and `composeAction` APIs first appeared in Thunderbird 64. They are very similar to Firefox's `browserAction` API.

Many of our [sample extensions](#) use a `browserAction`.

Use toolbar actions to put icons in the mail window toolbar. In addition to its icon, a toolbar action can also have a tooltip, a badge, and a popup. This namespace is called `browserAction` for compatibility with browser WebExtensions.

3.1 Manifest file properties

- `[browser_action]` (object)
 - `[browser_style]` (boolean)
 - `[default_area]` (string) Currently unused.
 - `[default_icon]` (IconPath)
 - `[default_popup]` (string)
 - `[default_title]` (string)
 - `[theme_icons]` (array of ThemeIcons) Specifies icons to use for dark and light themes

Note: A manifest entry named `browser_action` is required to use `browserAction`.

3.2 Functions

3.2.1 setTitle(details)

Sets the title of the toolbar action. This shows up in the tooltip.

- `details` (object)
 - `title` (string or null) The string the toolbar action should display when moused over.

3.2.2 getTitle(details)

Gets the title of the toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

3.2.3 setIcon(details)

Sets the icon for the toolbar action. The icon can be specified either as the path to an image file or as the pixel data from a canvas element, or as dictionary of either one of those. Either the **path** or the **imageData** property must be specified.

- `details` (object)
 - `[imageData]` (*ImageDataType* or object) Either an `ImageData` object or a dictionary { `size` -> `ImageData` } representing icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.imageData = foo'` is equivalent to `'details.imageData = {'19': foo}'`
 - `[path]` (string or object) Either a relative image path or a dictionary { `size` -> relative image path } pointing to icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.path = foo'` is equivalent to `'details.imageData = {'19': foo}'`

3.2.4 setPopup(details)

Sets the html document to be opened as a popup when the user clicks on the toolbar action's icon.

- `details` (object)
 - `popup` (string or null) The html file to show in a popup. If set to the empty string (`''`), no popup is shown.

3.2.5 getPopup(details)

Gets the html document set as the popup for this toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

3.2.6 `setBadgeText(details)`

Sets the badge text for the toolbar action. The badge is displayed on top of the icon.

- `details` (object)
 - `text` (string or null) Any number of characters can be passed, but only about four can fit in the space.

3.2.7 `getBadgeText(details)`

Gets the badge text of the toolbar action. If no tab nor window is specified is specified, the global badge text is returned.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

3.2.8 `setBadgeBackgroundColor(details)`

Sets the background color for the badge.

- `details` (object)
 - `color` (string or *ColorArray* or null) An array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255]. Can also be a string with a CSS value, with opaque red being #FF0000 or #F00.

3.2.9 `getBadgeBackgroundColor(details)`

Gets the background color of the toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- *ColorArray*

3.2.10 `enable([tabId])`

Enables the toolbar action for a tab. By default, toolbar actions are enabled.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

3.2.11 `disable([tabId])`

Disables the toolbar action for a tab.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

3.2.12 isEnabled(details)

Checks whether the toolbar action is enabled.

- `details` (*Details*)

3.2.13 openPopup()

Opens the extension popup window in the active window.

3.3 Events

3.3.1 onClicked(tab, [info])

Fired when a toolbar action icon is clicked. This event will not fire if the toolbar action has a popup.

- `tab` (*Tab*) Added in Thunderbird 74.0b2
- `[info]` (*OnClickData*) Added in Thunderbird 74.0b2

3.4 Types

3.4.1 ColorArray

array of integer

3.4.2 Details

Specifies to which tab or window the value should be set, or from which one it should be retrieved. If no tab nor window is specified, the global value is set or retrieved.

object:

- `[tabId]` (integer) When setting a value, it will be specific to the specified tab, and will automatically reset when the tab navigates. When getting, specifies the tab to get the value from; if there is no tab-specific value, the window one will be inherited.
- `[windowId]` (integer) When setting a value, it will be specific to the specified window. When getting, specifies the window to get the value from; if there is no window-specific value, the global one will be inherited.

3.4.3 ImageDataType

Pixel data for an image. Must be an ImageData object (for example, from a `canvas` element).

ImageData

3.4.4 OnClickData

Added in Thunderbird 74.0b2

Information sent when a browser action is clicked.

object:

- `modifiers` (array of *string*) An array of keyboard modifiers that were held while the menu item was clicked.
- `[button]` (integer) An integer value of button by which menu item was clicked.

Values for modifiers:

- Shift
- Alt
- Command
- Ctrl
- MacCtrl

The cloudFile (a.k.a. fileLink) API first appeared in Thunderbird 64, and was uplifted to Thunderbird 60.4 ESR.

From Thunderbird 68.2.1 (Thunderbird 71 beta), an extension can choose to receive data for upload as a `File` object rather than as an `ArrayBuffer`. You **should** specify which you want as the default may change in a future version.

The [DropBox Uploader](#) sample extension uses this API.

4.1 Manifest file properties

- `[cloud_file]` (object)
 - `management_url` (string) A page for configuring accounts, to be displayed in the preferences UI. **Note:** Within this UI only a limited subset of the WebExtension APIs is available: `cloudFile`, `extension`, `i18n`, `runtime`, `storage`, `test`.
 - `name` (string) Name of the cloud file service.
 - `[data_format]` (*string*) Determines the format of the data argument in `onFileUpload`. *Added in Thunderbird 71, backported to 68.2.1*
 - `[new_account_url]` (string) **Deprecated.** This property was never used.
 - `[service_url]` (string) URL to the web page of the cloud file service.

Values for `data_format`:

- `ArrayBuffer`
- `File`

Note: A manifest entry named `cloud_file` is required to use `cloudFile`.

4.2 Functions

4.2.1 `getAccount(accountId)`

Retrieve information about a single cloud file account

- `accountId` (string) Unique identifier of the account

Returns a `Promise` fulfilled with:

- *CloudFileAccount*

4.2.2 `getAllAccounts()`

Retrieve all cloud file accounts for the current add-on

Returns a `Promise` fulfilled with:

- array of *CloudFileAccount*

4.2.3 `updateAccount(accountId, updateProperties)`

Update a cloud file account

- `accountId` (string) Unique identifier of the account
- `updateProperties` (object)
 - `[configured]` (boolean) If true, the account is configured and ready to use. This property is currently ignored and all accounts are assumed to be configured.
 - `[managementUrl]` (string) A page for configuring accounts, to be displayed in the preferences UI.
 - `[spaceRemaining]` (integer) The amount of remaining space on the cloud provider, in bytes. Set to -1 if unsupported.
 - `[spaceUsed]` (integer) The amount of space already used on the cloud provider, in bytes. Set to -1 if unsupported.
 - `[uploadSizeLimit]` (integer) The maximum size in bytes for a single file to upload. Set to -1 if unlimited.

Returns a `Promise` fulfilled with:

- *CloudFileAccount*

4.3 Events

4.3.1 `onFileUpload(account, fileInfo)`

Fired when a file should be uploaded to the cloud file provider

- `account` (*CloudFileAccount*) The created account
- `fileInfo` (*CloudFile*) The file to upload

Event listeners should return:

- object
 - [aborted] (boolean) Set this to true if the file upload was aborted
 - [url] (string) The URL where the uploaded file can be accessed

4.3.2 onFileUploadAbort(account, fileId)

- account (*CloudFileAccount*) The created account
- fileId (integer) An identifier for this file

4.3.3 onFileDeleted(account, fileId)

Fired when a file previously uploaded should be deleted

- account (*CloudFileAccount*) The created account
- fileId (integer) An identifier for this file

4.3.4 onAccountAdded(account)

Fired when a cloud file account of this add-on was created

- account (*CloudFileAccount*) The created account

4.3.5 onAccountDeleted(accountId)

Fired when a cloud file account of this add-on was deleted

- accountId (string) The id of the removed account

4.4 Types

4.4.1 CloudFile

Information about a cloud file

object:

- data (*ArrayBuffer* or *File*)
- id (integer) An identifier for this file
- name (string) Filename of the file to be transferred

4.4.2 CloudFileAccount

Information about a cloud file account

object:

- configured (boolean) If true, the account is configured and ready to use. This property is currently ignored and all accounts are assumed to be configured.

- `id` (string) Unique identifier of the account
- `managementUrl` (string) A page for configuring accounts, to be displayed in the preferences UI.
- `name` (string) A user-friendly name for this account.
- `[spaceRemaining]` (integer) The amount of remaining space on the cloud provider, in bytes. Set to -1 if unsupported.
- `[spaceUsed]` (integer) The amount of space already used on the cloud provider, in bytes. Set to -1 if unsupported.
- `[uploadSizeLimit]` (integer) The maximum size in bytes for a single file to upload. Set to -1 if unlimited.

The commands API first appeared in Thunderbird 66. It's more or less the same as the [Firefox commands API](#).

Use the commands API to add keyboard shortcuts that trigger actions in your extension, for example, an action to open the browser action or send a command to the extension.

5.1 Manifest file properties

- `[commands]` (object)

Note: A manifest entry named `commands` is required to use `commands`.

5.2 Functions

5.2.1 `update(detail)`

Update the details of an already defined command.

- `detail` (object) The new description for the command.
 - `name` (string) The name of the command.
 - `[description]` (string) The new description for the command.
 - `[shortcut]` (string) An empty string to clear the shortcut, or a string matching the format of the `commands.manifest.json` key to set a new shortcut key. If the string does not match this format, the function throws an error.

5.2.2 reset(name)

Reset a command's details to what is specified in the manifest.

- `name` (string) The name of the command.

5.2.3 getAll()

Returns all the registered extension commands for this extension and their shortcut (if active).

Returns a [Promise](#) fulfilled with:

- array of *Command*

5.3 Events

5.3.1 onCommand(command)

Fired when a registered command is activated using a keyboard shortcut.

- `command` (string)

5.4 Types

5.4.1 Command

object:

- `[description]` (string) The Extension Command description
- `[name]` (string) The name of the Extension Command
- `[shortcut]` (string) The shortcut active for this command, or blank if not active.

This message composition window API first appeared in Thunderbird 67 (see [bug 1503423](#)).

6.1 Permissions

- compose “Read and modify your email messages as you compose and send them”

6.2 Functions

6.2.1 beginNew([details])

Changed in Thunderbird 77: return value added

- [details] (*ComposeDetails*)

Returns a `Promise` fulfilled with:

- *Tab*

6.2.2 beginReply(messageId, [replyType], [details])

Changed in Thunderbird 77: return value added

- messageId (integer) The message to reply to, as retrieved using other APIs.
- [replyType] (*string*)
- [details] (*ComposeDetails*) *Added in Thunderbird 76*

Returns a `Promise` fulfilled with:

- *Tab*

Values for replyType:

- replyToSender
- replyToList
- replyToAll

6.2.3 beginForward(messageId, [forwardType], [details])

Changed in Thunderbird 77: return value added

- messageId (integer) The message to forward, as retrieved using other APIs.
- [forwardType] (*string*)
- [details] (*ComposeDetails*)

Returns a [Promise](#) fulfilled with:

- *Tab*

Values for forwardType:

- forwardInline
- forwardAsAttachment

6.2.4 getComposeDetails(tabId)

Added in Thunderbird 74

Fetches the current state of a compose window. Currently only a limited amount of information is available, more will be added in later versions.

- tabId (integer)

Returns a [Promise](#) fulfilled with:

- *ComposeDetails*

Note: The permission `compose` is required to use `getComposeDetails`.

6.2.5 setComposeDetails(tabId, details)

Added in Thunderbird 74

Updates the compose window. Specify only fields that you want to change. Currently only the `to/cc/bcc/replyTo/followupTo/newsgroups` fields and the subject are implemented.

- tabId (integer)
- details (*ComposeDetails*)

Note: The permission `compose` is required to use `setComposeDetails`.

6.2.6 listAttachments(tabId)

Added in Thunderbird 78

Lists all of the attachments of the message being composed in the specified tab.

- `tabId` (integer)

6.2.7 addAttachment(tabId, data)

Added in Thunderbird 78

Adds an attachment to the message being composed in the specified tab.

- `tabId` (integer)
- `data` (object)
 - `file` (File)
 - `[name]` (string) The name, as displayed to the user, of this attachment. If not specified, the name of the `file` object is used.

6.2.8 updateAttachment(tabId, attachmentId, data)

Added in Thunderbird 78

Renames and/or replaces the contents of an attachment to the message being composed in the specified tab.

- `tabId` (integer)
- `attachmentId` (integer)
- `data` (object)
 - `[file]` (File)
 - `[name]` (string) The name, as displayed to the user, of this attachment. If not specified, the name of the `file` object is used.

6.2.9 removeAttachment(tabId, attachmentId)

Added in Thunderbird 78

Removes an attachment from the message being composed in the specified tab.

- `tabId` (integer)
- `attachmentId` (integer)

6.3 Events

6.3.1 onBeforeSend(tab, details)

Added in Thunderbird 74

Fired when a message is about to be sent from the compose window.

- `tab` (*Tab*) Added in Thunderbird 74.0b2
- `details` (*ComposeDetails*) The current state of the compose window. This is functionally the same as the *getComposeDetails(tabId)* function.

Event listeners should return:

- object
 - `[cancel]` (boolean) Cancels the send.
 - `[details]` (*ComposeDetails*) Updates the compose window. See the *setComposeDetails(tabId, details)* function for more information.

Note: The permission `compose` is required to use `onBeforeSend`.

6.3.2 `onAttachmentAdded(tab, attachment)`

Added in Thunderbird 78

Fired when an attachment is added to a message being composed.

- `tab` (*Tab*)
- `attachment` (*ComposeAttachment*)

6.3.3 `onAttachmentRemoved(tab, attachmentId)`

Added in Thunderbird 78

Fired when an attachment is removed from a message being composed.

- `tab` (*Tab*)
- `attachmentId` (integer)

6.3.4 `onIdentityChanged(tab, identityId)`

Added in Thunderbird 78.0b2

Fired when the user changes the identity that will be used to send a message being composed.

- `tab` (*Tab*)
- `identityId` (string)

Note: The permission `accountsRead` is required to use `onIdentityChanged`.

6.4 Types

6.4.1 `ComposeAttachment`

Added in Thunderbird 78

Represents an attachment in a message being composed.

object:

- `id` (integer) A unique identifier for this attachment.
- `name` (string) The name, as displayed to the user, of this attachment. This is usually but not always the filename of the attached file.
- `size` (integer) The size in bytes of this attachment. *Added in Thunderbird 83*
- `getFile()` Retrieves the contents of the attachment as a DOM `File` object.

6.4.2 ComposeDetails

Used by various functions to represent the state of a message being composed. Note that functions using this type may have a partial implementation.

object:

- `[attachments]` (array of object) Attachments to add to the message. Only used in the `begin*` functions. *Added in Thunderbird 82*
- `[bcc]` (*ComposeRecipientList*)
- `[body]` (string)
- `[cc]` (*ComposeRecipientList*)
- `[followupTo]` (*ComposeRecipientList*) *Added in Thunderbird 74*
- `[identityId]` (string) The ID of an identity from the *accounts* API. The settings from the identity will be used in the composed message. If `replyTo` is also specified, the `replyTo` property of the identity is overridden. The permission `accountsRead` is required to include the `identityId`. *Added in Thunderbird 76*
- `[isPlainText]` (boolean) *Added in Thunderbird 75*
- `[newsgroups]` (string or array of string) *Added in Thunderbird 74*
- `[plainTextBody]` (string) *Added in Thunderbird 75*
- `[replyTo]` (*ComposeRecipientList*)
- `[subject]` (string)
- `[to]` (*ComposeRecipientList*)

6.4.3 ComposeRecipient

string: A name and email address in the format “Name <email@example.com>”, or just an email address.

OR

object:

- `id` (string) The ID of a contact or mailing list from the *contacts* and *mailingLists* APIs.
- `type` (*string*) Which sort of object this ID is for.

Values for type:

- `contact`
- `mailingList`

6.4.4 ComposeRecipientList

Added in Thunderbird 74

string: A name and email address in the format “Name <email@example.com>”, or just an email address.

OR

array of *ComposeRecipient*:

composeAction

The *browserAction* and *composeAction* APIs first appeared in Thunderbird 64. They are very similar to Firefox's *browserAction* API.

Use toolbar actions to put icons in the message composition toolbars. In addition to its icon, a toolbar action can also have a tooltip, a badge, and a popup.

7.1 Manifest file properties

- [compose_action] (object)
 - [browser_style] (boolean)
 - [default_area] (*string*) Defines the location the composeAction will appear by default. The default location is maintoolbar.
 - [default_icon] (IconPath)
 - [default_popup] (string)
 - [default_title] (string)
 - [theme_icons] (array of ThemeIcons) Specifies icons to use for dark and light themes

Values for default_area:

- maintoolbar
- formattoolbar

Note: A manifest entry named `compose_action` is required to use `composeAction`.

7.2 Functions

7.2.1 setTitle(details)

Sets the title of the toolbar action. This shows up in the tooltip.

- `details` (object)
 - `title` (string or null) The string the toolbar action should display when moused over.

7.2.2 getTitle(details)

Gets the title of the toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

7.2.3 setIcon(details)

Sets the icon for the toolbar action. The icon can be specified either as the path to an image file or as the pixel data from a canvas element, or as dictionary of either one of those. Either the **path** or the **imageData** property must be specified.

- `details` (object)
 - `[imageData]` (*ImageDataType* or object) Either an `ImageData` object or a dictionary `{size -> ImageData}` representing icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.imageData = foo'` is equivalent to `'details.imageData = {'19': foo}'`
 - `[path]` (string or object) Either a relative image path or a dictionary `{size -> relative image path}` pointing to icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.path = foo'` is equivalent to `'details.imageData = {'19': foo}'`

7.2.4 setPopup(details)

Sets the html document to be opened as a popup when the user clicks on the toolbar action's icon.

- `details` (object)
 - `popup` (string or null) The html file to show in a popup. If set to the empty string (`''`), no popup is shown.

7.2.5 getPopup(details)

Gets the html document set as the popup for this toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

7.2.6 `setBadgeText(details)`

Sets the badge text for the toolbar action. The badge is displayed on top of the icon.

- `details` (object)
 - `text` (string or null) Any number of characters can be passed, but only about four can fit in the space.

7.2.7 `getBadgeText(details)`

Gets the badge text of the toolbar action. If no tab nor window is specified is specified, the global badge text is returned.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

7.2.8 `setBadgeBackgroundColor(details)`

Sets the background color for the badge.

- `details` (object)
 - `color` (string or *ColorArray* or null) An array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255]. Can also be a string with a CSS value, with opaque red being #FF0000 or #F00.

7.2.9 `getBadgeBackgroundColor(details)`

Gets the background color of the toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- *ColorArray*

7.2.10 `enable([tabId])`

Enables the toolbar action for a tab. By default, toolbar actions are enabled.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

7.2.11 `disable([tabId])`

Disables the toolbar action for a tab.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

7.2.12 isEnabled(details)

Checks whether the toolbar action is enabled.

- `details` (*Details*)

7.2.13 openPopup()

Opens the extension popup window in the active window.

7.3 Events

7.3.1 onClicked(tab, [info])

Fired when a toolbar action icon is clicked. This event will not fire if the toolbar action has a popup.

- `tab` (*Tab*) Added in Thunderbird 74.0b2
- `[info]` (*OnClickData*) Added in Thunderbird 74.0b2

7.4 Types

7.4.1 ColorArray

array of integer

7.4.2 Details

Specifies to which tab or window the value should be set, or from which one it should be retrieved. If no tab nor window is specified, the global value is set or retrieved.

object:

- `[tabId]` (integer) When setting a value, it will be specific to the specified tab, and will automatically reset when the tab navigates. When getting, specifies the tab to get the value from; if there is no tab-specific value, the window one will be inherited.
- `[windowId]` (integer) When setting a value, it will be specific to the specified window. When getting, specifies the window to get the value from; if there is no window-specific value, the global one will be inherited.

7.4.3 ImageDataType

Pixel data for an image. Must be an ImageData object (for example, from a `canvas` element).

ImageData

7.4.4 OnClickData

Added in Thunderbird 74.0b2

Information sent when a compose action is clicked.

object:

- `modifiers` (array of *string*) An array of keyboard modifiers that were held while the menu item was clicked.
- `[button]` (integer) An integer value of button by which menu item was clicked.

Values for modifiers:

- Shift
- Alt
- Command
- Ctrl
- MacCtrl

composeScripts

This compose scripts API first appeared in Thunderbird 77. Functionally it is the same as the `content scripts` API except that it works on the document of email messages during composition. See the MDN documentation for a more in-depth explanation and *Changes in Thunderbird 77* for examples.

See also *executeScript*, *insertCSS*, *removeCSS*, and *messageDisplayScripts*.

Note: Registering a compose script through `manifest.json` is not possible at this point.

Note: The permission `compose` is required to use `composeScripts`.

8.1 Functions

8.1.1 register(composeScriptOptions)

Register a compose script programmatically

- `composeScriptOptions` (*RegisteredComposeScriptOptions*)

8.2 Types

8.2.1 RegisteredComposeScript

An object that represents a compose script registered programmatically

object:

- `unregister()` Unregister a compose script registered programmatically

8.2.2 RegisteredComposeScriptOptions

Details of a compose script registered programmatically

object:

- `[css]` (array of `ExtensionFileOrCode`) The list of CSS files to inject
- `[js]` (array of `ExtensionFileOrCode`) The list of JavaScript files to inject

The address books API, also including the *addressBooks* and *mailingLists* namespaces, first appeared in Thunderbird 64. The `quickSearch` function was added in Thunderbird 68.

The `Address Books` sample extension uses this API.

9.1 Permissions

- `addressBooks` “Read and modify your address books and contacts”

Note: The permission `addressBooks` is required to use `contacts`.

9.2 Functions

9.2.1 `list(parentId)`

Gets all the contacts in the address book with the id `parentId`.

- `parentId` (string)

Returns a `Promise` fulfilled with:

- array of *ContactNode*

9.2.2 `quickSearch([parentId], searchString)`

Gets all contacts matching `searchString` in the address book with the id `parentId`.

- `[parentId]` (string) The ID of the address book to search. If not specified, all address books are searched.

- `searchString` (string) One or more space-separated terms to search for.

Returns a `Promise` fulfilled with:

- array of `ContactNode`

9.2.3 `get(id)`

Gets a single contact.

- `id` (string)

Returns a `Promise` fulfilled with:

- `ContactNode`

9.2.4 `create(parentId, [id], properties)`

Adds a new contact to the address book with the id `parentId`.

- `parentId` (string)
- `[id]` (string) Assigns the contact an id. If an existing contact has this id, an exception is thrown.
- `properties` (`ContactProperties`)

Returns a `Promise` fulfilled with:

- string The ID of the new contact.

9.2.5 `update(id, properties)`

Edits the properties of a contact. To remove a property, specify it as `null`.

- `id` (string)
- `properties` (`ContactProperties`)

9.2.6 `delete(id)`

Removes a contact from the address book. The contact is also removed from any mailing lists it is a member of.

- `id` (string)

9.3 Events

9.3.1 `onCreated(node, id)`

Fired when a contact is created.

- `node` (`ContactNode`)
- `id` (string)

9.3.2 onUpdated(node, changedProperties)

Fired when a contact is changed.

- `node` (*ContactNode*)
- `changedProperties` (*PropertyChange*) *Added in Thunderbird 83*

9.3.3 onDeleted(parentId, id)

Fired when a contact is removed from an address book.

- `parentId` (string)
- `id` (string)

9.4 Types

9.4.1 ContactNode

A node representing a contact in an address book.

object:

- `id` (string) The unique identifier for the node. IDs are unique within the current profile, and they remain valid even after the program is restarted.
- `properties` (*ContactProperties*)
- `type` (*NodeType*) Always set to `contact`.
- `[parentId]` (string) The `id` of the parent object.
- `[readOnly]` (boolean) Indicates if the object is read-only. Currently this returns `false` in all cases, as read-only address books are ignored by the API.

9.4.2 ContactProperties

A set of properties for a particular contact. For a complete list of properties that Thunderbird uses, see <https://hg.mozilla.org/comm-central/file/tip/mailnews/addrbook/public/nsIAbCard.idl>

It is also possible to store custom properties. The custom property name however may only use a to z, A to Z, 1 to 9 and underscores.

object

9.4.3 PropertyChange

Added in Thunderbird 83

A dictionary of changed properties. Keys are the property name that changed, values are an object containing `oldValue` and `newValue`. Values can be either a string or null.

object

The folders API first appeared in Thunderbird 68 (see [bug 1531591](#)) as a part of the *accounts* API. They were later moved here.

10.1 Permissions

- `accountsFolders` “Create, rename, or delete your mail account folders”

Note: The permission `accountsFolders` is required to use `folders`.

10.2 Functions

10.2.1 `create(parentFolder, childName)`

Creates a new subfolder of `parentFolder`.

- `parentFolder` (*MailFolder*)
- `childName` (string)

10.2.2 `rename(folder, newName)`

Renames a folder.

- `folder` (*MailFolder*)
- `newName` (string)

10.2.3 delete(folder)

Deletes a folder.

- `folder` (*MailFolder*)

10.3 Types

10.3.1 MailFolder

A folder object, as returned by the `list` and `get` methods. Use the `accountId` and `path` properties to refer to a folder object:

- `accountId` (string) The account this folder belongs to.
- `path` (string) Path to this folder in the account. Although paths look predictable, never guess a folder's path, as there are a number of reasons why it may not be what you think it is.
- `[name]` (string) The human-friendly name of this folder.
- `[subFolders]` (array of *MailFolder*) *Added in Thunderbird 74*
- `[type]` (*string*) The type of folder, for several common types.

Values for type:

- `inbox`
- `drafts`
- `sent`
- `trash`
- `templates`
- `archives`
- `junk`
- `outbox`

The address books API, also including the *addressBooks* and *contacts* namespaces, first appeared in Thunderbird 64. The *Address Books* sample extension uses this API.

11.1 Permissions

- addressBooks “Read and modify your address books and contacts”

Note: The permission `addressBooks` is required to use `mailingLists`.

11.2 Functions

11.2.1 list(parentId)

Gets all the mailing lists in the address book with id `parentId`.

- `parentId` (string)

Returns a `Promise` fulfilled with:

- array of *MailingListNode*

11.2.2 get(id)

Gets a single mailing list.

- `id` (string)

Returns a `Promise` fulfilled with:

- *MailingListNode*

11.2.3 create(parentId, properties)

Creates a new mailing list in the address book with id `parentId`.

- `parentId` (string)
- `properties` (object)
 - `name` (string)
 - `[description]` (string)
 - `[nickName]` (string)

Returns a [Promise](#) fulfilled with:

- string The ID of the new mailing list.

11.2.4 update(id, properties)

Edits the properties of a mailing list.

- `id` (string)
- `properties` (object)
 - `name` (string)
 - `[description]` (string)
 - `[nickName]` (string)

11.2.5 delete(id)

Removes the mailing list.

- `id` (string)

11.2.6 addMember(id, contactId)

Adds a contact to the mailing list with id `id`. If the contact and mailing list are in different address books, the contact will also be copied to the list's address book.

- `id` (string)
- `contactId` (string)

11.2.7 listMembers(id)

Gets all contacts that are members of the mailing list with id `id`.

- `id` (string)

Returns a [Promise](#) fulfilled with:

- array of *ContactNode*

11.2.8 removeMember(id, contactId)

Removes a contact from the mailing list with id `id`. This does not delete the contact from the address book.

- `id` (string)
- `contactId` (string)

11.3 Events

11.3.1 onCreated(node)

Fired when a mailing list is created.

- `node` (*MailingListNode*)

11.3.2 onUpdated(node)

Fired when a mailing list is changed.

- `node` (*MailingListNode*)

11.3.3 onDeleted(parentId, id)

Fired when a mailing list is deleted.

- `parentId` (string)
- `id` (string)

11.3.4 onMemberAdded(node)

Fired when a contact is added to the mailing list.

- `node` (*ContactNode*)

11.3.5 onMemberRemoved(parentId, id)

Fired when a contact is removed from the mailing list.

- `parentId` (string)
- `id` (string)

11.4 Types

11.4.1 MailingListNode

A node representing a mailing list.

object:

- `description` (string)
- `id` (string) The unique identifier for the node. IDs are unique within the current profile, and they remain valid even after the program is restarted.
- `name` (string)
- `nickName` (string)
- `type` (*NodeType*) Always set to `mailingList`.
- `[contacts]` (array of *ContactNode*) A list of contacts held by this node's address book or mailing list.
- `[parentId]` (string) The `id` of the parent object.
- `[readOnly]` (boolean) Indicates if the object is read-only. Currently this returns `false` in all cases, as read-only address books are ignored by the API.

The messages API first appeared in Thunderbird 66 (see [bug 1499617](#)).

The [Filter](#) and [Layout](#) sample extensions use this API.

12.1 Functions

12.1.1 `query(queryInfo)`

Gets all mail tabs that have the specified properties, or all mail tabs if no properties are specified.

- `queryInfo` (object)
 - `[active]` (boolean) Whether the tabs are active in their windows.
 - `[currentWindow]` (boolean) Whether the tabs are in the current window.
 - `[lastFocusedWindow]` (boolean) Whether the tabs are in the last focused window.
 - `[windowId]` (integer) The ID of the parent window, or `WINDOW_ID_CURRENT` for the current window.

Returns a `Promise` fulfilled with:

- array of *MailTab*

12.1.2 `update([tabId], updateProperties)`

Modifies the properties of a mail tab. Properties that are not specified in `updateProperties` are not modified.

- `[tabId]` (integer) Defaults to the active tab of the current window.
- `updateProperties` (object)
 - `[displayedFolder]` (*MailFolder*) Sets the folder displayed in the tab. The extension must have an `accounts` permission to do this.

- [folderPaneVisible] (boolean) Shows or hides the folder pane.
- [layout] (*string*) Sets the arrangement of the folder pane, message list pane, and message display pane. Note that setting this applies it to all mail tabs.
- [messagePaneVisible] (boolean) Shows or hides the message display pane.
- [sortOrder] (*string*) Sorts the list of messages. `sortType` must also be given.
- [sortType] (*string*) Sorts the list of messages. `sortOrder` must also be given.

Values for layout:

- standard
- wide
- vertical

Values for sortOrder:

- none
- ascending
- descending

Values for sortType:

- none
- date
- subject
- author
- id
- thread
- priority
- status
- size
- flagged
- unread
- recipient
- location
- tags
- junkStatus
- attachments
- account
- custom
- received
- correspondent

12.1.3 `getSelectedMessages([tabId])`

Lists the selected messages in the current folder. A messages permission is required to do this.

- `[tabId]` (integer) Defaults to the active tab of the current window.

Returns a `Promise` fulfilled with:

- `MessageList`

Note: The permission `messagesRead` is required to use `getSelectedMessages`.

12.1.4 `setQuickFilter([tabId], properties)`

Sets the Quick Filter user interface based on the options specified.

- `[tabId]` (integer) Defaults to the active tab of the current window.
- `properties` (object)
 - `[attachment]` (boolean) Shows only messages with attachments.
 - `[contact]` (boolean) Shows only messages from people in the address book.
 - `[flagged]` (boolean) Shows only flagged messages.
 - `[show]` (boolean) Shows or hides the Quick Filter bar.
 - `[tags]` (boolean or `TagsDetail`) Shows only messages with tags on them.
 - `[text]` (`QuickFilterTextDetail`) Shows only messages matching the supplied text.
 - `[unread]` (boolean) Shows only unread messages.

12.2 Events

12.2.1 `onDisplayedFolderChanged(tab, displayedFolder)`

Fired when the displayed folder changes in any mail tab.

- `tab` (`Tab`) *Changed in Thunderbird 76, previously just the tab's ID*
- `displayedFolder` (`MailFolder`)

Note: The permission `accountsRead` is required to use `onDisplayedFolderChanged`.

12.2.2 `onSelectedMessagesChanged(tab, selectedMessages)`

Fired when the selected messages change in any mail tab.

- `tab` (`Tab`) *Changed in Thunderbird 76, previously just the tab's ID*
- `selectedMessages` (`MessageList`)

Note: The permission `messagesRead` is required to use `onSelectedMessagesChanged`.

12.3 Types

12.3.1 MailTab

object:

- `active` (boolean)
- `displayedFolder` (*MailFolder*) The `accountsRead` permission is required.
- `folderPaneVisible` (boolean)
- `id` (integer)
- `layout` (*string*)
- `messagePaneVisible` (boolean)
- `sortOrder` (*string*)
- `sortType` (*string*)
- `windowId` (integer)

Values for `layout`:

- `standard`
- `wide`
- `vertical`

Values for `sortOrder`:

- `none`
- `ascending`
- `descending`

Values for `sortType`:

- `none`
- `date`
- `subject`
- `author`
- `id`
- `thread`
- `priority`
- `status`
- `size`
- `flagged`

- unread
- recipient
- location
- tags
- junkStatus
- attachments
- account
- custom
- received
- correspondent

12.3.2 QuickFilterTextDetail

object:

- `text` (string) String to match against the recipients, author, subject, or body.
- `[author]` (boolean) Shows messages where `text` matches the author.
- `[body]` (boolean) Shows messages where `text` matches the message body.
- `[recipients]` (boolean) Shows messages where `text` matches the recipients.
- `[subject]` (boolean) Shows messages where `text` matches the subject.

The menus API first appeared in Thunderbird 66 (see [bug 1503421](#)). It is basically the same as the [Firefox menus API](#), but modified to suit Thunderbird. Note that the similar `contextMenus` API will not be added to Thunderbird.

Use the `browser.menus` API to add items to the browser's menus. You can choose what types of objects your context menu additions apply to, such as images, hyperlinks, and pages.

13.1 Permissions

- `menus`
- `menus.overrideContext`

Note: The permission `menus` is required to use `menus`.

13.2 Functions

13.2.1 `create(createProperties, [callback])`

Creates a new context menu item. Note that if an error occurs during creation, you may not find out until the creation callback fires (the details will be in `runtime.lastError`).

- `createProperties` (object)
 - `[checked]` (boolean) The initial state of a checkbox or radio item: `true` for selected and `false` for unselected. Only one radio item can be selected at a time in a given group of radio items.
 - `[command]` (string) Specifies a command to issue for the context click. Currently supports internal command `_execute_browser_action`.

- [contexts] (array of *ContextType*) List of contexts this menu item will appear in. Defaults to ['page'] if not specified.
 - [documentUrlPatterns] (array of string) Lets you restrict the item to apply only to documents whose URL matches one of the given patterns. (This applies to frames as well.) For details on the format of a pattern, see [Match Patterns](#).
 - [enabled] (boolean) Whether this context menu item is enabled or disabled. Defaults to true.
 - [icons] (object)
 - [id] (string) The unique ID to assign to this item. Mandatory for event pages. Cannot be the same as another ID for this extension.
 - [onclick] (function) A function that will be called back when the menu item is clicked. Event pages cannot use this.
 - [parentId] (integer or string) The ID of a parent menu item; this makes the item a child of a previously added item.
 - [targetUrlPatterns] (array of string) Similar to documentUrlPatterns, but lets you filter based on the src attribute of img/audio/video tags and the href of anchor tags.
 - [title] (string) The text to be displayed in the item; this is *required* unless type is 'separator'. When the context is 'selection', you can use %s within the string to show the selected text. For example, if this parameter's value is "Translate '%s' to Pig Latin" and the user selects the word "cool", the context menu item for the selection is "Translate 'cool' to Pig Latin". To specify an access key for the new menu entry, include a & before the desired letter in the title. For example "&Help".
 - [type] (*ItemType*) The type of menu item. Defaults to 'normal' if not specified.
 - [viewTypes] (array of *ViewType*) List of view types where the menu item will be shown. Defaults to any view, including those without a viewType.
 - [visible] (boolean) Whether the item is visible in the menu.
- [callback] (function) Called when the item has been created in the browser. If there were any problems creating the item, details will be available in `runtime.lastError`.

Returns a [Promise](#) fulfilled with:

- integer or string The ID of the newly created item.

13.2.2 update(id, updateProperties)

Updates a previously created context menu item.

- id (integer or string) The ID of the item to update.
- updateProperties (object) The properties to update. Accepts the same values as the create function.
 - [checked] (boolean)
 - [contexts] (array of *ContextType*)
 - [documentUrlPatterns] (array of string)
 - [enabled] (boolean)
 - [icons] (object)
 - [onclick] (function)
 - [parentId] (integer or string) Note: You cannot change an item to be a child of one of its own descendants.

- [targetUrlPatterns] (array of string)
- [title] (string)
- [type] (*ItemType*)
- [viewTypes] (array of *ViewType*)
- [visible] (boolean) Whether the item is visible in the menu.

13.2.3 remove(menuItemId)

Removes a context menu item.

- menuItemId (integer or string) The ID of the context menu item to remove.

13.2.4 removeAll()

Removes all context menu items added by this extension.

13.2.5 overrideContext(contextOptions)

Show the matching menu items from this extension instead of the default menu. This should be called during a 'contextmenu' DOM event handler, and only applies to the menu that opens after this event.

- contextOptions (object)
 - [context] (*string*) ContextType to override, to allow menu items from other extensions in the menu. Currently only 'tab' is supported. showDefaults cannot be used with this option.
 - [showDefaults] (boolean) Whether to also include default menu items in the menu.
 - [tabId] (integer) Required when context is 'tab'. Requires 'tabs' permission.

Values for context:

- tab

Note: The permission `menus.overrideContext` is required to use `overrideContext`.

13.2.6 refresh()

Updates the extension items in the shown menu, including changes that have been made since the menu was shown. Has no effect if the menu is hidden. Rebuilding a shown menu is an expensive operation, only invoke this method when necessary.

13.3 Events

13.3.1 onClicked(info, [tab])

Fired when a context menu item is clicked.

- info (*OnClickData*) Information about the item clicked and the context where the click happened.

- `[tab]` (*Tab*) The details of the tab where the click took place. If the click did not take place in a tab, this parameter will be missing.

13.3.2 onShown(info, tab)

Fired when a menu is shown. The extension can add, modify or remove menu items and call `menus.refresh()` to update the menu.

- `info` (object) Information about the context of the menu action and the created menu items. For more information about each property, see `OnClickData`. The following properties are only set if the extension has host permissions for the given context: `linkUrl`, `linkText`, `srcUrl`, `pageUrl`, `frameUrl`, `selectionText`.
 - `contexts` (array of *ContextType*) A list of all contexts that apply to the menu.
 - `editable` (boolean)
 - `menuIds` (array of `None`) A list of IDs of the menu items that were shown.
 - `[attachments]` (array of *ComposeAttachment*) *Added in Thunderbird 83*
 - `[frameUrl]` (string)
 - `[linkText]` (string)
 - `[linkUrl]` (string)
 - `[mediaType]` (string)
 - `[pageUrl]` (string)
 - `[selectionText]` (string)
 - `[srcUrl]` (string)
 - `[targetElementId]` (integer)
 - `[viewType]` (*ViewType*)
- `tab` (*Tab*) The details of the tab where the menu was opened.

13.3.3 onHidden()

Fired when a menu is hidden. This event is only fired if `onShown` has fired before.

13.4 Properties

13.4.1 ACTION_MENU_TOP_LEVEL_LIMIT

The maximum number of top level extension items that can be added to an extension action context menu. Any items beyond this limit will be ignored.

13.5 Types

13.5.1 ContextType

The different contexts a menu can appear in. Specifying 'all' is equivalent to the combination of all other contexts except for 'tab'.

string

Values for ContextType:

- all
- page
- frame
- selection
- link
- editable
- password
- image
- video
- audio
- browser_action
- tab
- message_list
- folder_pane
- compose_attachments *Added in Thunderbird 83*

13.5.2 ItemType

The type of menu item.

string

Values for ItemType:

- normal
- checkbox
- radio
- separator

13.5.3 OnClickData

Information sent when a context menu item is clicked.

object:

- `editable` (boolean) A flag indicating whether the element is editable (text input, textarea, etc.).

- `menuItemId` (integer or string) The ID of the menu item that was clicked.
- `modifiers` (array of *string*) An array of keyboard modifiers that were held while the menu item was clicked.
- `[attachments]` (array of *ComposeAttachment*) The selected attachments of a message being composed. *Added in Thunderbird 83*
- `[button]` (integer) An integer value of button by which menu item was clicked.
- `[checked]` (boolean) A flag indicating the state of a checkbox or radio item after it is clicked.
- `[displayedFolder]` (*MailFolder*) The displayed folder, if the context menu was opened in the message list. The `accountsRead` permission is required.
- `[frameId]` (integer) The id of the frame of the element where the context menu was clicked.
- `[frameUrl]` (string) The URL of the frame of the element where the context menu was clicked, if it was in a frame.
- `[linkText]` (string) If the element is a link, the text of that link.
- `[linkUrl]` (string) If the element is a link, the URL it points to.
- `[mediaType]` (string) One of 'image', 'video', or 'audio' if the context menu was activated on one of these types of elements.
- `[pageUrl]` (string) The URL of the page where the menu item was clicked. This property is not set if the click occurred in a context where there is no current page, such as in a launcher context menu.
- `[parentMenuItemId]` (integer or string) The parent ID, if any, for the item clicked.
- `[selectedFolder]` (*MailFolder*) The selected folder, if the context menu was opened in the folder pane. The `accountsRead` permission is required.
- `[selectedMessages]` (*MessageList*) The selected messages, if the context menu was opened in the message list. The `messagesRead` permission is required.
- `[selectionText]` (string) The text for the context selection, if any.
- `[srcUrl]` (string) Will be present for elements with a 'src' URL.
- `[targetElementId]` (integer) An identifier of the clicked element, if any. Use `menus.getTargetElement` in the page to find the corresponding element.
- `[viewType]` (*ViewType*) The type of view where the menu is clicked. May be unset if the menu is not associated with a view.
- `[wasChecked]` (boolean) A flag indicating the state of a checkbox or radio item before it was clicked.

Values for modifiers:

- `Shift`
- `Alt`
- `Command`
- `Ctrl`
- `MacCtrl`

The message display API first appeared in Thunderbird 70 and was backported to Thunderbird 68.2.

A message can be displayed in either a 3-pane tab, a tab of its own, or in a window of its own. All are referenced by `tabId` in this API. Display windows are considered to have exactly one tab, which has limited functionality compared to tabs from the main window.

More functions are planned for this API for adding to the user interface, as well as a message display action (similar to *browserAction* and *composeAction*).

Note: The permission `messagesRead` is required to use `messageDisplay`.

14.1 Functions

14.1.1 `getDisplayedMessage(tabId)`

Gets the currently displayed message in the specified tab. It returns null if no messages are selected, or if multiple messages are selected.

- `tabId` (integer)

Returns a `Promise` fulfilled with:

- *MessageHeader*

14.1.2 `getDisplayedMessages(tabId)`

Added in Thunderbird 81

Gets an array of the currently displayed messages in the specified tab. The array is empty if no messages are displayed.

- `tabId` (integer)

Returns a *Promise* fulfilled with:

- array of *MessageHeader*

14.2 Events

14.2.1 onMessageDisplayed(tab, message)

Fired when a message is displayed, whether in a 3-pane tab, a message tab, or a message window.

- `tab` (*Tab*) *Changed in Thunderbird 76, previously just the tab's ID*
- `message` (*MessageHeader*)

14.2.2 onMessagesDisplayed(tab, messages)

Added in Thunderbird 81

Fired when either a single message is displayed or when multiple messages are displayed, whether in a 3-pane tab, a message tab, or a message window.

- `tab` (*Tab*)
- `messages` (array of *MessageHeader*)

messageDisplayAction

The `messageDisplayAction` API was added in Thunderbird 71, and was uplifted to Thunderbird 68.3 ESR. It is similar to Firefox's `browserAction` API and can be combined with the `messageDisplay` API to determine the currently displayed message.

Use toolbar actions to put icons in the message display toolbar. In addition to its icon, a toolbar action can also have a tooltip, a badge, and a popup.

15.1 Manifest file properties

- `[message_display_action]` (object)
 - `[browser_style]` (boolean)
 - `[default_area]` (string) Currently unused.
 - `[default_icon]` (IconPath)
 - `[default_popup]` (string)
 - `[default_title]` (string)
 - `[theme_icons]` (array of ThemeIcons) Specifies icons to use for dark and light themes

Note: A manifest entry named `message_display_action` is required to use `messageDisplayAction`.

15.2 Functions

15.2.1 setTitle(details)

Sets the title of the toolbar action. This shows up in the tooltip.

- `details` (object)
 - `title` (string or null) The string the toolbar action should display when moused over.

15.2.2 getTitle(details)

Gets the title of the toolbar action.

- `details` (*Details*)

Returns a [Promise](#) fulfilled with:

- string

15.2.3 setIcon(details)

Sets the icon for the toolbar action. The icon can be specified either as the path to an image file or as the pixel data from a canvas element, or as dictionary of either one of those. Either the **path** or the **imageData** property must be specified.

- `details` (object)
 - `[imageData]` (*ImageDataType* or object) Either an `ImageData` object or a dictionary { `size` -> `ImageData` } representing icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.imageData = foo'` is equivalent to `'details.imageData = {'19': foo}'`
 - `[path]` (string or object) Either a relative image path or a dictionary { `size` -> relative image path } pointing to icon to be set. If the icon is specified as a dictionary, the actual image to be used is chosen depending on screen's pixel density. If the number of image pixels that fit into one screen space unit equals `scale`, then image with size `scale * 19` will be selected. Initially only scales 1 and 2 will be supported. At least one image must be specified. Note that `'details.path = foo'` is equivalent to `'details.imageData = {'19': foo}'`

15.2.4 setPopup(details)

Sets the html document to be opened as a popup when the user clicks on the toolbar action's icon.

- `details` (object)
 - `popup` (string or null) The html file to show in a popup. If set to the empty string (`''`), no popup is shown.

15.2.5 getPopup(details)

Gets the html document set as the popup for this toolbar action.

- `details` (*Details*)

Returns a [Promise](#) fulfilled with:

- string

15.2.6 `setBadgeText(details)`

Sets the badge text for the toolbar action. The badge is displayed on top of the icon.

- `details` (object)
 - `text` (string or null) Any number of characters can be passed, but only about four can fit in the space.

15.2.7 `getBadgeText(details)`

Gets the badge text of the toolbar action. If no tab nor window is specified is specified, the global badge text is returned.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- string

15.2.8 `setBadgeBackgroundColor(details)`

Sets the background color for the badge.

- `details` (object)
 - `color` (string or *ColorArray* or null) An array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255]. Can also be a string with a CSS value, with opaque red being #FF0000 or #F00.

15.2.9 `getBadgeBackgroundColor(details)`

Gets the background color of the toolbar action.

- `details` (*Details*)

Returns a `Promise` fulfilled with:

- *ColorArray*

15.2.10 `enable([tabId])`

Enables the toolbar action for a tab. By default, toolbar actions are enabled.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

15.2.11 `disable([tabId])`

Disables the toolbar action for a tab.

- `[tabId]` (integer) The id of the tab for which you want to modify the toolbar action.

15.2.12 isEnabled(details)

Checks whether the toolbar action is enabled.

- `details` (*Details*)

15.2.13 openPopup()

Opens the extension popup window in the active window.

15.3 Events

15.3.1 onClicked(tab, [info])

Fired when a toolbar action icon is clicked. This event will not fire if the toolbar action has a popup.

- `tab` (*Tab*) Added in Thunderbird 74.0b2
- `[info]` (*OnClickData*) Added in Thunderbird 74.0b2

15.4 Types

15.4.1 ColorArray

array of integer

15.4.2 Details

Specifies to which tab or window the value should be set, or from which one it should be retrieved. If no tab nor window is specified, the global value is set or retrieved.

object:

- `[tabId]` (integer) When setting a value, it will be specific to the specified tab, and will automatically reset when the tab navigates. When getting, specifies the tab to get the value from; if there is no tab-specific value, the window one will be inherited.
- `[windowId]` (integer) When setting a value, it will be specific to the specified window. When getting, specifies the window to get the value from; if there is no window-specific value, the global one will be inherited.

15.4.3 ImageDataType

Pixel data for an image. Must be an ImageData object (for example, from a `canvas` element).

ImageData

15.4.4 OnClickData

Added in Thunderbird 74.0b2

Information sent when a message display action is clicked.

object:

- `modifiers` (array of *string*) An array of keyboard modifiers that were held while the menu item was clicked.
- `[button]` (integer) An integer value of button by which menu item was clicked.

Values for modifiers:

- Shift
- Alt
- Command
- Ctrl
- MacCtrl

messageDisplayScripts

This message display scripts API first appeared in Thunderbird 82. Functionally it is the same as the [content scripts](#) API except that it works on the document of email messages being displayed. See the MDN documentation for a more in-depth explanation and *Changes in Thunderbird 82* for examples.

See also *executeScript*, *insertCSS*, *removeCSS*, and *composeScripts*.

Note: Registering a message display script through `manifest.json` is not possible at this point.

16.1 Permissions

- `messagesModify` “Read and modify your email messages as they are displayed to you”

Note: The permission `messagesModify` is required to use `messageDisplayScripts`.

16.2 Functions

16.2.1 `register(messageDisplayScriptOptions)`

Register a message display script programmatically

- `messageDisplayScriptOptions` (*RegisteredMessageDisplayScriptOptions*)

16.3 Types

16.3.1 RegisteredMessageDisplayScript

An object that represents a message display script registered programmatically

object:

- `unregister()` Unregister a message display script registered programmatically

16.3.2 RegisteredMessageDisplayScriptOptions

Details of a message display script registered programmatically

object:

- `[css]` (array of `ExtensionFileOrCode`) The list of CSS files to inject
- `[js]` (array of `ExtensionFileOrCode`) The list of JavaScript files to inject

The messages API first appeared in Thunderbird 66.

Note: When the term `messageId` is used in these documents, it *doesn't* refer to the Message-ID email header. It is an internal tracking number that does not remain after a restart. Nor does it follow an email that has been moved to a different folder.

Warning: Some functions in this API potentially return *a lot* of messages. Be careful what you wish for! See [Working with Message Lists](#) for more information.

17.1 Permissions

- `messagesMove` “Move, copy, or delete your email messages”
- `messagesRead` “Read your email messages and mark or tag them”

Note: The permission `messagesRead` is required to use `messages`.

17.2 Functions

17.2.1 `list(folder)`

Gets all messages in a folder.

- `folder` (*MailFolder*)

Returns a [Promise](#) fulfilled with:

- *MessageList*

Note: The permission `accountsRead` is required to use `list`.

17.2.2 `continueList(messageListId)`

Returns the next chunk of messages in a list. See *Working with Message Lists* for more information.

- `messageListId` (string)

Returns a `Promise` fulfilled with:

- *MessageList*

17.2.3 `get(messageId)`

Returns a specified message.

- `messageId` (integer)

Returns a `Promise` fulfilled with:

- *MessageHeader*

17.2.4 `getFull(messageId)`

Returns a specified message, including all headers and MIME parts.

- `messageId` (integer)

Returns a `Promise` fulfilled with:

- *MessagePart*

17.2.5 `getRaw(messageId)`

Added in Thunderbird 72, backported to 68.7

Returns the unmodified source of a message.

- `messageId` (integer)

Returns a `Promise` fulfilled with:

- `string`

17.2.6 `query(queryInfo)`

Added in Thunderbird 69, backported to 68.2

Gets all messages that have the specified properties, or all messages if no properties are specified.

- `queryInfo` (object)
 - `[author]` (string) Returns only messages with this value matching the author.
 - `[body]` (string) Returns only messages with this value in the body of the mail.

- [flagged] (boolean) Returns only flagged (or unflagged if false) messages.
- [folder] (*MailFolder*) Returns only messages from the specified folder. The `accountsRead` permission is required.
- [fromDate] (*Date*) Returns only messages with a date after this value.
- [fromMe] (boolean) Returns only messages with the author matching any configured identity.
- [fullText] (string) Returns only messages with this value somewhere in the mail (subject, body or author).
- [recipients] (string) Returns only messages with this value matching one or more recipients.
- [subject] (string) Returns only messages with this value matching the subject.
- [tags] (*TagsDetail*) Returns only messages with the specified tags. For a list of available tags, call the `listTags` method. Querying for messages that must *not* have a tag does not work. *Added in Thunderbird 74*
- [toDate] (*Date*) Returns only messages with a date before this value.
- [toMe] (boolean) Returns only messages with one or more recipients matching any configured identity.
- [unread] (boolean) Returns only unread (or read if false) messages.

Returns a `Promise` fulfilled with:

- *MessageList*

17.2.7 update(messageId, newProperties)

Marks or unmarks a message as read, flagged, or tagged.

- `messageId` (integer)
- `newProperties` (object)
 - [flagged] (boolean) Marks the message as flagged or unflagged.
 - [junk] (boolean) Marks the message as junk or not junk. *Added in Thunderbird 73, backported to 68.7*
 - [read] (boolean) Marks the message as read or unread.
 - [tags] (array of string) Sets the tags on the message. For a list of available tags, call the `listTags` method.

17.2.8 move(messageIds, destination)

Moves messages to a specified folder.

- `messageIds` (array of integer) The IDs of the messages to move.
- `destination` (*MailFolder*) The folder to move the messages to.

Note: The permissions `accountsRead` and `messagesMove` are required to use `move`.

17.2.9 copy(messageIds, destination)

Copies messages to a specified folder.

- `messageIds` (array of integer) The IDs of the messages to copy.

- `destination` (*MailFolder*) The folder to copy the messages to.

Note: The permissions `accountsRead` and `messagesMove` are required to use `copy`.

17.2.10 `delete(messageIds, [skipTrash])`

Deletes messages, or moves them to the trash folder.

- `messageIds` (array of integer) The IDs of the messages to delete.
- `[skipTrash]` (boolean) If true, the message will be permanently deleted without warning the user. If false or not specified, it will be moved to the trash folder.

Note: The permission `messagesMove` is required to use `delete`.

17.2.11 `archive(messageIds)`

Archives messages using the current settings.

- `messageIds` (array of integer) The IDs of the messages to archive.

Note: The permission `messagesMove` is required to use `archive`.

17.2.12 `listTags()`

Returns a list of tags that can be set on messages, and their human-friendly name, colour, and sort order.

Returns a *Promise* fulfilled with:

- array of *MessageTag*

17.3 Events

17.3.1 `onNewMailReceived(folder, messages)`

Added in Thunderbird 75

Fired when a new message is received, and has been through junk classification and message filters.

- `folder` (*MailFolder*)
- `messages` (*MessageList*)

Note: The permission `accountsRead` is required to use `onNewMailReceived`.

17.4 Types

17.4.1 MessageHeader

object:

- `author` (string)
- `bccList` (array of string)
- `ccList` (array of string)
- `date` (date)
- `flagged` (boolean)
- `folder` (*MailFolder*) The `accountsRead` permission is required.
- `id` (integer)
- `junk` (boolean) *Added in Thunderbird 74*
- `junkScore` (integer) *Added in Thunderbird 74*
- `read` (boolean)
- `recipients` (array of string)
- `subject` (string)
- `tags` (array of string)

17.4.2 MessageList

See *Working with Message Lists* for more information.

object:

- `id` (string)
- `messages` (array of *MessageHeader*)

17.4.3 MessagePart

Represents an email message “part”, which could be the whole message

object:

- `[body]` (string) The content of the part
- `[contentType]` (string)
- `[headers]` (object) An object of part headers, with the header name as key, and an array of header values as value
- `[name]` (string) Name of the part, if it is a file
- `[partName]` (string)
- `[parts]` (array of *MessagePart*) Any sub-parts of this part
- `[size]` (integer)

17.4.4 MessageTag

object:

- `color` (string) Tag color
- `key` (string) Distinct tag identifier – use this string when referring to a tag
- `ordinal` (string) Custom sort string (usually empty)
- `tag` (string) Human-readable tag name

17.4.5 TagsDetail

Used for filtering messages by tag in various methods. Note that functions using this type may have a partial implementation.

object:

- `mode` (*string*) Whether all of the tag filters must apply, or any of them.
- `tags` (object) Object keys are tags to filter on, values are `true` if the message must have the tag, or `false` if it must not have the tag. For a list of available tags, call the *listTags()* method.

Values for mode:

- `all`
- `any`

Use the `browser.tabs` API to interact with the browser's tab system. You can use this API to create, modify, and rearrange tabs in the browser.

18.1 Permissions

- `activeTab`
- `tabs` “Access browser tabs”
- `tabHide` “Hide and show browser tabs”

18.2 Functions

18.2.1 `get(tabId)`

Retrieves details about the specified tab.

- `tabId` (integer)

Returns a `Promise` fulfilled with:

- *Tab*

18.2.2 `getCurrent()`

Gets the tab that this script call is being made from. May be undefined if called from a non-tab context (for example: a background page or popup view).

Returns a `Promise` fulfilled with:

- *Tab*

18.2.3 connect(tabId, [connectInfo])

Added in Thunderbird 82

Connects to the content script(s) in the specified tab. The `runtime.onConnect` event is fired in each content script running in the specified tab for the current extension. For more details, see [Content Script Messaging](#).

- `tabId` (integer)
- `[connectInfo]` (object)
 - `[frameId]` (integer) Open a port to a specific frame identified by `frameId` instead of all frames in the tab.
 - `[name]` (string) Will be passed into `onConnect` for content scripts that are listening for the connection event.

Returns a [Promise](#) fulfilled with:

- `runtime.Port` A port that can be used to communicate with the content scripts running in the specified tab.

18.2.4 sendMessage(tabId, message, [options])

Added in Thunderbird 82

Sends a single message to the content script(s) in the specified tab, with an optional callback to run when a response is sent back. The `runtime.onMessage` event is fired in each content script running in the specified tab for the current extension.

- `tabId` (integer)
- `message` (any)
- `[options]` (object)
 - `[frameId]` (integer) Send a message to a specific frame identified by `frameId` instead of all frames in the tab.

Returns a [Promise](#) fulfilled with:

- any The JSON response object sent by the handler of the message. If an error occurs while connecting to the specified tab, the callback will be called with no arguments and `runtime.lastError` will be set to the error message.

18.2.5 create(createProperties)

Creates a new tab.

- `createProperties` (object)
 - `[active]` (boolean) Whether the tab should become the active tab in the window. Does not affect whether the window is focused (see [update\(windowId, updateInfo\)](#)). Defaults to `true`.
 - `[index]` (integer) The position the tab should take in the window. The provided value will be clamped to between zero and the number of tabs in the window.
 - `[selected]` (boolean) **Unsupported.** Whether the tab should become the selected tab in the window. Defaults to `true`.
 - `[url]` (string) The URL to navigate the tab to initially. Fully-qualified URLs must include a scheme (i.e. `'http://www.google.com'`, not `'www.google.com'`). Relative URLs will be relative to the current page within the extension. Defaults to the New Tab Page.

- [windowId] (integer) The window to create the new tab in. Defaults to the current window.

Returns a [Promise](#) fulfilled with:

- *Tab* Details about the created tab. Will contain the ID of the new tab.

18.2.6 duplicate(tabId)

Duplicates a tab.

- tabId (integer) The ID of the tab which is to be duplicated.

Returns a [Promise](#) fulfilled with:

- *Tab* Details about the duplicated tab. The *Tab* object doesn't contain `url`, `title` and `favIconUrl` if the "tabs" permission has not been requested.

18.2.7 query(queryInfo)

Gets all tabs that have the specified properties, or all tabs if no properties are specified.

- queryInfo (object)
 - [active] (boolean) Whether the tabs are active in their windows.
 - [currentWindow] (boolean) Whether the tabs are in the current window.
 - [highlighted] (boolean) Whether the tabs are highlighted. Works as an alias of active.
 - [index] (integer) The position of the tabs within their windows.
 - [lastFocusedWindow] (boolean) Whether the tabs are in the last focused window.
 - [mailTab] (boolean) Whether the tab is a Thunderbird 3-pane tab.
 - [status] (*TabStatus*) Whether the tabs have completed loading.
 - [title] (string) Match page titles against a pattern.
 - [url] (string or array of string) Match tabs against one or more [URL Patterns](#). Note that fragment identifiers are not matched.
 - [windowId] (integer) The ID of the parent window, or *WINDOW_ID_CURRENT* for the current window.
 - [windowType] (*WindowType*) The type of window the tabs are in.

Returns a [Promise](#) fulfilled with:

- array of *Tab*

18.2.8 update([tabId], updateProperties)

Modifies the properties of a tab. Properties that are not specified in `updateProperties` are not modified.

- [tabId] (integer) Defaults to the selected tab of the current window.
- updateProperties (object)
 - [active] (boolean) Whether the tab should be active. Does not affect whether the window is focused (see [update\(windowId, updateInfo\)](#)).
 - [url] (string) A URL to navigate the tab to.

Returns a [Promise](#) fulfilled with:

- *Tab* Details about the updated tab. The *Tab* object doesn't contain `url`, `title` and `favIconUrl` if the "tabs" permission has not been requested.

18.2.9 `move(tabIds, moveProperties)`

Moves one or more tabs to a new position within its window, or to a new window. Note that tabs can only be moved to and from normal (`window.type === "normal"`) windows.

- `tabIds` (integer or array of integer) The tab or list of tabs to move.
- `moveProperties` (object)
 - `index` (integer) The position to move the window to. -1 will place the tab at the end of the window.
 - `[windowId]` (integer) Defaults to the window the tab is currently in.

Returns a [Promise](#) fulfilled with:

- *Tab* or array of *Tab* Details about the moved tabs.

18.2.10 `reload([tabId], [reloadProperties])`

Reload a tab.

- `[tabId]` (integer) The ID of the tab to reload; defaults to the selected tab of the current window.
- `[reloadProperties]` (object)
 - `[bypassCache]` (boolean) Whether using any local cache. Default is false.

18.2.11 `remove(tabIds)`

Closes one or more tabs.

- `tabIds` (integer or array of integer) The tab or list of tabs to close.

18.2.12 `executeScript([tabId], details)`

Changed in Thunderbird 77: with the "compose" permission, this now works in the document of email messages during composition.

Injects JavaScript code into a page. For details, see the [programmatic injection](#) section of the content scripts doc.

- `[tabId]` (integer) The ID of the tab in which to run the script; defaults to the active tab of the current window.
- `details` ([InjectDetails](#)) Details of the script to run.

Returns a [Promise](#) fulfilled with:

- array of any The result of the script in every injected frame.

18.2.13 insertCSS([tabId], details)

Changed in Thunderbird 77: with the “compose” permission, this now works in the document of email messages during composition.

Injects CSS into a page. For details, see the [programmatic injection](#) section of the content scripts doc.

- `tabId` (integer) The ID of the tab in which to insert the CSS; defaults to the active tab of the current window.
- `details` ([InjectDetails](#)) Details of the CSS text to insert.

18.2.14 removeCSS([tabId], details)

Changed in Thunderbird 77: with the “compose” permission, this now works in the document of email messages during composition.

Removes injected CSS from a page. For details, see the [programmatic injection](#) section of the content scripts doc.

- `tabId` (integer) The ID of the tab from which to remove the injected CSS; defaults to the active tab of the current window.
- `details` ([InjectDetails](#)) Details of the CSS text to remove.

18.3 Events

18.3.1 onCreated(tab)

Fired when a tab is created. Note that the tab’s URL may not be set at the time this event fired, but you can listen to `onUpdated` events to be notified when a URL is set.

- `tab` ([Tab](#)) Details of the tab that was created.

18.3.2 onUpdated(tabId, changeInfo, tab)

Fired when a tab is updated.

- `tabId` (integer)
- `changeInfo` (object) Lists the changes to the state of the tab that was updated.
 - `faviconUrl` (string) The tab’s new favicon URL.
 - `status` (string) The status of the tab. Can be either *loading* or *complete*.
 - `url` (string) The tab’s URL if it has changed.
- `tab` ([Tab](#)) Gives the state of the tab that was updated.

18.3.3 onMoved(tabId, moveInfo)

Fired when a tab is moved within a window. Only one move event is fired, representing the tab the user directly moved. Move events are not fired for the other tabs that must move in response. This event is not fired when a tab is moved between windows. For that, see [onDetached\(tabId, detachInfo\)](#).

- `tabId` (integer)
- `moveInfo` (object)

- `fromIndex` (`integer`)
- `toIndex` (`integer`)
- `windowId` (`integer`)

18.3.4 `onActivated(activeInfo)`

Fires when the active tab in a window changes. Note that the tab's URL may not be set at the time this event fired, but you can listen to `onUpdated` events to be notified when a URL is set.

- `activeInfo` (`object`)
 - `tabId` (`integer`) The ID of the tab that has become active.
 - `windowId` (`integer`) The ID of the window the active tab changed inside of.

18.3.5 `onDetached(tabId, detachInfo)`

Fired when a tab is detached from a window, for example because it is being moved between windows.

- `tabId` (`integer`)
- `detachInfo` (`object`)
 - `oldPosition` (`integer`)
 - `oldWindowId` (`integer`)

18.3.6 `onAttached(tabId, attachInfo)`

Fired when a tab is attached to a window, for example because it was moved between windows.

- `tabId` (`integer`)
- `attachInfo` (`object`)
 - `newPosition` (`integer`)
 - `newWindowId` (`integer`)

18.3.7 `onRemoved(tabId, removeInfo)`

Fired when a tab is closed.

- `tabId` (`integer`)
- `removeInfo` (`object`)
 - `isWindowClosing` (`boolean`) True when the tab is being closed because its window is being closed.
 - `windowId` (`integer`) The window whose tab is closed.

18.4 Properties

18.4.1 TAB_ID_NONE

An ID which represents the absence of a browser tab.

18.5 Types

18.5.1 Tab

object:

- `active` (boolean) Whether the tab is active in its window. (Does not necessarily mean the window is focused.)
- `highlighted` (boolean) Whether the tab is highlighted. Works as an alias of `active`
- `index` (integer) The zero-based index of the tab within its window.
- `selected` (boolean) **Unsupported.** Whether the tab is selected.
- `[faviconUrl]` (string) The URL of the tab's favicon. This property is only present if the extension's manifest includes the "tabs" permission. It may also be an empty string if the tab is loading.
- `[height]` (integer) The height of the tab in pixels.
- `[id]` (integer) The ID of the tab. Tab IDs are unique within a browser session. Under some circumstances a Tab may not be assigned an ID. Tab ID can also be set to `TAB_ID_NONE` for apps and devtools windows.
- `[mailTab]` (boolean) Whether the tab is a 3-pane tab.
- `[status]` (string) Either *loading* or *complete*.
- `[title]` (string) The title of the tab. This property is only present if the extension's manifest includes the "tabs" permission.
- `[url]` (string) The URL the tab is displaying. This property is only present if the extension's manifest includes the "tabs" permission.
- `[width]` (integer) The width of the tab in pixels.
- `[windowId]` (integer) The ID of the window the tab is contained within.

18.5.2 TabStatus

Whether the tabs have completed loading.

string

Values for TabStatus:

- `loading`
- `complete`

18.5.3 UpdateFilter

An object describing filters to apply to tabs.onUpdated events.

object:

- [properties] (array of *UpdatePropertyName*) A list of property names. Events that do not match any of the names will be filtered out.
- [tabId] (integer)
- [urls] (array of string) A list of URLs or URL patterns. Events that cannot match any of the URLs will be filtered out. Filtering with urls requires the "tabs" or "activeTab" permission.
- [windowId] (integer)

18.5.4 UpdatePropertyName

Event names supported in onUpdated.

string

Values for UpdatePropertyName:

- favIconUrl
- status
- title

18.5.5 WindowType

The type of window.

string

Values for WindowType:

- normal
- popup
- panel
- app
- devtools

Note: These APIs are for the main Thunderbird windows which can contain webpage tabs and also other window types like composer or address books that cannot contain webpage tabs. Make sure your code interacts with windows appropriately, depending on their type.

Use the `browser.windows` API to interact with Thunderbird. You can use this API to create, modify, and rearrange windows.

19.1 Functions

19.1.1 `get(windowId, [getInfo])`

Gets details about a window.

- `windowId` (integer)
- `[getInfo]` (*GetInfo*)

Returns a `Promise` fulfilled with:

- *Window*

19.1.2 `getCurrent([getInfo])`

Gets the current window.

- `[getInfo]` (*GetInfo*)

Returns a `Promise` fulfilled with:

- *Window*

19.1.3 getLastFocused([getInfo])

Gets the window that was most recently focused — typically the window ‘on top’.

- [getInfo] (*GetInfo*)

Returns a *Promise* fulfilled with:

- *Window*

19.1.4 getAll([getInfo])

Gets all windows.

- [getInfo] (object) Specifies properties used to filter the *Window* returned and to determine whether they should contain a list of the *Tab* objects.
 - [populate] (boolean) If true, each *Window* object will have a `tabs` property that contains a list of the *Tab* objects for that window. The *Tab* objects only contain the `url`, `title` and `favIconUrl` properties if the extension’s manifest file includes the `"tabs"` permission.
 - [windowTypes] (array of *WindowType*) If set, the *Window* returned will be filtered based on its type.

Returns a *Promise* fulfilled with:

- array of *Window*

19.1.5 create([createData])

Creates (opens) a new browser with any optional sizing, position or default URL provided.

- [createData] (object)
 - [allowScriptsToClose] (boolean) Allow scripts to close the window.
 - [focused] (boolean) **Unsupported.** If true, opens an active window. If false, opens an inactive window.
 - [height] (integer) The height in pixels of the new window, including the frame. If not specified defaults to a natural height.
 - [incognito] (boolean) Whether the new window should be an incognito window.
 - [left] (integer) The number of pixels to position the new window from the left edge of the screen. If not specified, the new window is offset naturally from the last focused window. This value is ignored for panels.
 - [state] (*WindowState*) The initial state of the window. The ‘minimized’, ‘maximized’ and ‘fullscreen’ states cannot be combined with ‘left’, ‘top’, ‘width’ or ‘height’.
 - [tabId] (integer) The id of the tab for which you want to adopt to the new window.
 - [titlePreface] (string) A string to add to the beginning of the window title.
 - [top] (integer) The number of pixels to position the new window from the top edge of the screen. If not specified, the new window is offset naturally from the last focused window. This value is ignored for panels.
 - [type] (*CreateType*) Specifies what type of browser window to create. The ‘panel’ and ‘detached_panel’ types create a popup unless the ‘-enable-panels’ flag is set.

- `[url]` (string or array of string) A URL or array of URLs to open as tabs in the window. Fully-qualified URLs must include a scheme (i.e. `'http://www.google.com'`, not `'www.google.com'`). Relative URLs will be relative to the current page within the extension. Defaults to the New Tab Page.
- `[width]` (integer) The width in pixels of the new window, including the frame. If not specified defaults to a natural width.

Returns a *Promise* fulfilled with:

- *Window* Contains details about the created window.

19.1.6 `update(windowId, updateInfo)`

Updates the properties of a window. Specify only the properties that you want to change; unspecified properties will be left unchanged.

- `windowId` (integer)
- `updateInfo` (object)
 - `[drawAttention]` (boolean) If true, causes the window to be displayed in a manner that draws the user's attention to the window, without changing the focused window. The effect lasts until the user changes focus to the window. This option has no effect if the window already has focus. Set to false to cancel a previous draw attention request.
 - `[focused]` (boolean) If true, brings the window to the front. If false, brings the next window in the z-order to the front.
 - `[height]` (integer) The height to resize the window to in pixels. This value is ignored for panels.
 - `[left]` (integer) The offset from the left edge of the screen to move the window to in pixels. This value is ignored for panels.
 - `[state]` (*WindowState*) The new state of the window. The 'minimized', 'maximized' and 'fullscreen' states cannot be combined with 'left', 'top', 'width' or 'height'.
 - `[titlePreface]` (string) A string to add to the beginning of the window title.
 - `[top]` (integer) The offset from the top edge of the screen to move the window to in pixels. This value is ignored for panels.
 - `[width]` (integer) The width to resize the window to in pixels. This value is ignored for panels.

Returns a *Promise* fulfilled with:

- *Window*

19.1.7 `remove(windowId)`

Removes (closes) a window, and all the tabs inside it.

- `windowId` (integer)

19.2 Events

19.2.1 `onCreated(window)`

Fired when a window is created.

- `window` (*Window*) Details of the window that was created.

19.2.2 onRemoved(windowId)

Fired when a window is removed (closed).

- `windowId` (integer) ID of the removed window.

19.2.3 onFocusChanged(windowId)

Fired when the currently focused window changes. Will be `WINDOW_ID_NONE` if all browser windows have lost focus. Note: On some Linux window managers, `WINDOW_ID_NONE` will always be sent immediately preceding a switch from one browser window to another.

- `windowId` (integer) ID of the newly focused window.

19.3 Properties

19.3.1 WINDOW_ID_CURRENT

The `windowId` value that represents the current window.

19.3.2 WINDOW_ID_NONE

The `windowId` value that represents the absence of a window.

19.4 Types

19.4.1 CreateType

Specifies what type of browser window to create. The `'panel'` and `'detached_panel'` types create a popup unless the `'-enable-panels'` flag is set.

string

Values for `CreateType`:

- `normal`
- `popup`
- `panel`
- `detached_panel`

19.4.2 GetInfo

Specifies whether the *Window* returned should contain a list of the *Tab* objects.

object:

- `[populate]` (boolean) If true, the *Window* returned will have a `tabs` property that contains a list of the *Tab* objects. The *Tab* objects only contain the `url`, `title` and `favIconUrl` properties if the extension's manifest file includes the "tabs" permission.
- `[windowTypes]` (array of *WindowType*) **Deprecated.** `windowTypes` is deprecated and ignored on Thunderbird.

19.4.3 Window

object:

- `alwaysOnTop` (boolean) Whether the window is set to be always on top.
- `focused` (boolean) Whether the window is currently the focused window.
- `incognito` (boolean) Whether the window is incognito.
- `[height]` (integer) The height of the window, including the frame, in pixels.
- `[id]` (integer) The ID of the window. Window IDs are unique within a session.
- `[left]` (integer) The offset of the window from the left edge of the screen in pixels.
- `[state]` (*WindowState*) The state of this browser window.
- `[tabs]` (array of *Tab*) Array of *Tab* objects representing the current tabs in the window.
- `[title]` (string) The title of the window. Read-only.
- `[top]` (integer) The offset of the window from the top edge of the screen in pixels.
- `[type]` (*WindowType*) The type of browser window this is.
- `[width]` (integer) The width of the window, including the frame, in pixels.

19.4.4 WindowState

The state of this window.

string

Values for WindowState:

- `normal`
- `minimized`
- `maximized`
- `fullscreen`
- `docked`

19.4.5 WindowType

The type of window this is. Under some circumstances a Window may not be assigned type property.

string

Values for WindowType:

- `normal`

- popup
- panel
- app
- devtools
- addressBook *Added in Thunderbird 70, backported to 68.1.1*
- messageCompose *Added in Thunderbird 70, backported to 68.1.1*
- messageDisplay *Added in Thunderbird 70, backported to 68.1.1*

The following APIs are also included and work as they do in Firefox:

- contentScripts
- downloads
- experiments
- extension
- i18n
- identity *Added in Thunderbird 78.0b2*
- management
- permissions
- pkcs11
- runtime
- storage
- theme

20.1 Changes in Thunderbird 74

20.1.1 legacy

- The legacy API has been removed. This has been covered elsewhere.

20.1.2 accounts/folders

- The *MailFolder* object gained a *subFolders* property. From now on the *accounts API functions* will return a hierarchy of folders instead of a flat list. If you still need a flat list you should traverse the folder hierarchy:

```
browser.accounts.list().then(accounts => {
  let arrayOfFolders = [];

  function traverse(folders) {
    if (!folders) {
      return;
    }
    for (let f of folders) {
      arrayOfFolders.push(f);
      traverse(f.subFolders);
    }
  }

  for (let account of accounts) {
    traverse(account.folders);
  }
  return arrayOfFolders;
});
```

This example code works with both the current API in 68 and the new version in 74 (which will be backported to 68 after some time in beta).

20.1.3 compose

- The compose API gained two new functions and an event. The *getComposeDetails* and *setComposeDetails* functions let you retrieve or change the recipients and subject of a message being composed. More details will be added later. The *onBeforeSend* event is fired when a message is about to be sent. At this point your extension can prevent sending or change the same details as in the new functions.

20.1.4 messages

- The *query* function can now query by tags.
- The *MessageHeader* object now has junk properties: `junkScore` is an integer score from 0 (not spam) to 100 (spam). `junk` shows whether or not that score is greater than the junk threshold.

20.2 Changes in Thunderbird 75

20.2.1 browserAction/composeAction/messageDisplayAction

- The `onClick` event of the action APIs now has information about the active tab and the click event. This brings the events into line with browser WebExtensions. For `messageDisplayAction`, this is a change of behaviour – previously the ID of the active tab was passed as the first argument.

This change was uplifted to 74 beta 2.

20.2.2 compose

- The *onBeforeSend* event now has information about the active tab.

This change was uplifted to 74 beta 2.

- The compose API functions can now handle the message body.

All responses from *getComposeDetails* calls and *onBeforeSend* events now return the message body. How you handle this depends on the composition mode (plain text or HTML). For plain text composition mode, the `isPlainText` field is set to `true` and `plainTextBody` should be used. For HTML composition, `isPlainText` is set to `false` and `body` should be used.

The `body` field is a string. You should parse this to an HTML document before modifying it, and serialise the document back to a string before calling `setComposeDetails`. [Here's a sample extension.](#)

20.2.3 messages

- A new event was added: *onNewMailReceived*.

Your extension can now be notified of incoming mail. This is based on the `MailServices.mfn.msgsClassified` notification.

20.3 Changes in Thunderbird 76

20.3.1 accounts

- Thunderbird 76 introduces the *MailIdentity* type for interacting with mail identities. Like the rest of the accounts API, it is mostly read-only as we believe that configuration of identities should only happen using the built-in UI.
- The *MailAccount* type now contains a list of identities associated with that account. The default identity is listed first and other identities are in no particular order.
- The accounts API now has a *setDefaultIdentity(accountId, identityId)* function.

20.3.2 compose

- The *ComposeDetails* type now has an *identityId* field for getting or setting the identity associated with a message being composed.

20.3.3 mailTabs/messageDisplay

- For consistency with other APIs and with browser WebExtensions (ie. those used in Firefox, etc.), some events that passed a numeric tab ID to listeners now pass an object representing the tab instead. *This change is not backwards-compatible.*

The affected events are:

- *onDisplayedFolderChanged(tab, displayedFolder)*
- *onSelectedMessagesChanged(tab, selectedMessages)*
- *onMessageDisplayed(tab, message)*

20.3.4 messages

- The `accountsRead` permission is now required for all functions that accept a *MailFolder* argument. The permission was already required to obtain a *MailFolder* anyway, so this change should not break extensions.

20.3.5 experiments

- For extensions with the `addressBooks` permission, a new `addressBookManager` object is available to WebExtensions experiment implementations. The `addressBookManager` provides the following functions to help you interact with the *addressBooks*, *contacts* and *mailingLists* APIs:
 - `findAddressBookById`, `findContactById`, `findMailingListById` to help you find “real” address book objects (`nsIAbCard`, `nsIAbDirectory`) for the IDs provided by the `addressBooks` API. Note that there is active development in the address book and these interfaces will be changing in the near term without public announcement.
 - `convert` to turn “real” objects back into API-friendly objects.

For more information on these functions see the [source code of the addressBooks APIs](#).

20.4 Changes in Thunderbird 77

20.4.1 compose

- The *compose* API functions *beginNew*, *beginReply*, and *beginForward* now return a *Tab* object for use with other API functions.
- Listeners to the *compose.onBeforeSend* event are now called sequentially in the order they were added. Be aware that other extensions may see this event before or after yours does.

20.4.2 tabs

- Events in the *tabs* and *windows* APIs now fire in many more situations:
 - Calendar tabs now fire *tabs.onCreated*, *tabs.onActivated*, and *tabs.onRemoved* just like other tabs do.
 - The first 3-pane tab in a main window now fires *tabs.onCreated* when the window opens.
 - Address book, composition, and message display windows now fire *tabs.onCreated* and *tabs.onRemoved* events as well as *windows.onCreated* and *windows.onRemoved* events. Each of these windows has exactly one tab in the *tabs* API and the added events reflect that.

Whether the `tabs` event happens before or after the corresponding `windows` event is non-deterministic.

20.4.3 composeScripts/tabs

- Content script functions can now operate on a compose window “tab” in the same way they do on a content tab in Thunderbird or Firefox. (Despite the fact they don’t have tabs, compose windows have one tab object under the *tabs* API.) This requires the “compose” permission.

Here are some basic examples. See [the MDN documentation](#) for a more in-depth explanation.

```
// Where tabId is the id of a compose window tab:

browser.tabs.executeScript(tabId, {
  code: `document.body.textContent = "Hey look, the script ran!";`,
});

browser.tabs.executeScript(tabId, {
  file: "compose.js",
});

browser.tabs.insertCSS(tabId, {
  code: "body { background-color: red; }",
});

browser.tabs.insertCSS(tabId, {
  file: "compose.css",
});

browser.tabs.removeCSS(tabId, {
  code: "body { background-color: red; }",
});

browser.tabs.removeCSS(tabId, {
```

(continues on next page)

(continued from previous page)

```
file: "compose.css",
});
```

- Scripts can also be registered to run automatically on composition window “tabs”, using the new *composeScripts* API. Again, this works just like the contentScripts API:

```
let registeredScripts = await browser.composeScripts.register({
  css: [
    // Any number of code or file objects could be listed here.
    { code: "body { background-color: red; }" },
    { file: "compose.css" },
  ],
  js: [
    // Any number of code or file objects could be listed here.
    { code: `document.body.textContent = "Hey look, the script ran!";` },
    { file: "compose.js" },
  ],
});
```

Added code will run immediately and CSS will be immediately applied to already-open composition windows, and any new composition windows.

The returned value, `registeredScripts` in this example, is an object with which you can unregister the code/CSS:

```
await registeredScripts.unregister();
```

Warning: This functionality has the ability to completely destroy every message being composed, with no way to undo it. Be careful!

Note: Javascript or CSS applied by these methods is *not* sent with the message. This is not a way to decorate messages or make them interactive.

20.5 Changes in Thunderbird 78

20.5.1 compose

- Attachments in the compose window can now be accessed with the compose API. These functions have been added:

- *listAttachments(tabId)*
- *addAttachment(tabId, data)*
- *updateAttachment(tabId, attachmentId, data)*
- *removeAttachment(tabId, attachmentId)*

These events have been added:

- *onAttachmentAdded(tab, attachment)*
- *onAttachmentRemoved(tab, attachmentId)*

See the documentation on those functions and events for more information.

- A new event, *onIdentityChanged(tab, identityId)* was added.

This change was made in Thunderbird 78 beta 2.

20.5.2 identity

- The `browser.identity` namespace for OAuth handling was enabled.

This change was made in Thunderbird 78 beta 2.

20.6 Changes in Thunderbird 81

20.6.1 messageDisplay

- The following API has been added to allow access to details of multiple-selection of email. Previously only a single selection function was available.

- *getDisplayedMessages(tabId)*

These events have been added:

- *onMessagesDisplayed(tab, messages)*

See the documentation on those functions and events for more information.

20.7 Changes in Thunderbird 82

20.7.1 compose

- Attachments can now be specified in the *beginNew*, *beginReply*, and *beginForward* functions.

20.7.2 tabs

- The *connect* and *sendMessage* functions now work as they do in Firefox.

20.7.3 messageDisplayScripts/tabs

- Content script functions can now operate on a message display “tab” in the same way they do on a content tab in Thunderbird or Firefox. This requires the new “messagesModify” permission.

Here are some basic examples. See [the MDN documentation](#) for a more in-depth explanation.

```
// Where tabId is the id of a message display tab:

browser.tabs.executeScript(tabId, {
  code: `document.body.textContent = "Hey look, the script ran!";`,
});

browser.tabs.executeScript(tabId, {
```

(continues on next page)

(continued from previous page)

```

    file: "display.js",
  });

  browser.tabs.insertCSS(tabId, {
    code: "body { background-color: red; }",
  });

  browser.tabs.insertCSS(tabId, {
    file: "display.css",
  });

  browser.tabs.removeCSS(tabId, {
    code: "body { background-color: red; }",
  });

  browser.tabs.removeCSS(tabId, {
    file: "display.css",
  });

```

- Scripts can also be registered to run automatically on messages being displayed, using the new *messageDisplayScripts* API. Again, this works just like the contentScripts API:

```

let registeredScripts = await browser.messageDisplayScripts.register({
  css: [
    // Any number of code or file objects could be listed here.
    { code: "body { background-color: red; }" },
    { file: "display.css" },
  ],
  js: [
    // Any number of code or file objects could be listed here.
    { code: `document.body.textContent = "Hey look, the script ran!";` },
    { file: "display.js" },
  ],
});

```

Added code will run immediately and CSS will be immediately applied to already-open message display tabs or windows, and any new message display tabs or windows.

The returned value, `registeredScripts` in this example, is an object with which you can unregister the code/CSS:

```
await registeredScripts.unregister();
```

Note: This functionality does *not* permanently alter messages, only what the user sees when they are displayed.

20.8 Changes in Thunderbird 83

20.8.1 compose

- *ComposeAttachment* objects now have a `size` property with the size of the attachment in bytes.

20.8.2 contacts

- The *onUpdated* event now calls listeners with a second argument, containing details of the changes made. For example:

```
{
  "FirstName": {
    "oldValue": "Daniel",
    "newValue": "Danielle"
  },
  "DisplayName": {
    "oldValue": "Daniel Smith",
    "newValue": "Danielle Smith"
  }
}
```

20.8.3 menus

- A `compose_attachment` context can now be used. This context applies when the user opens a context menu on selected attachments in a compose window. The selected attachments can be accessed from the `attachments` property in an *onShown* or *onClicked* listener.

21.1 WebExtension Experiments

A WebExtension experiment is an additional API that is shipped with your WebExtension. It allows your extension to interact with Thunderbird, much like earlier types of extension did. If you find the built-in Thunderbird APIs can do 80% of what you want to achieve, then WebExtension experiments are for you.

Note: Firefox does not allow WebExtension experiments on release or beta versions. Thunderbird does.

21.1.1 Adding an experiment to your extension

The full code of this example is on [GitHub](#).

Note: This is a very cut-down example. You may find the [Firefox documentation](#) helpful, particularly the pages on API schemas, implementing a function, and implementing an event.

Extension manifest

Experimental APIs are declared in the `experiment_apis` property in a WebExtension's `manifest.json` file. For example:

```
{
  "manifest_version": 2,
  "name": "Extension containing an experimental API",
  "experiment_apis": {
    "myapi": {
      "schema": "schema.json",
      "parent": {
        "scopes": ["addon_parent"],

```

(continues on next page)

(continued from previous page)

```
    "paths": [ ["myapi"] ],
    "script": "implementation.js"
  }
}
```

Schema

The schema defines the interface between your experiment API and the rest of your extension, which would use `browser.myapi` in this example. In it you describe the functions, events, and types you'll be implementing:

```
[
  {
    "namespace": "myapi",
    "functions": [
      {
        "name": "sayHello",
        "type": "function",
        "description": "Says hello to the user.",
        "async": true,
        "parameters": [
          {
            "name": "name",
            "type": "string",
            "description": "Who to say hello to."
          }
        ]
      }
    ]
  }
]
```

You can see some more-complicated schemas [in the Thunderbird source code](#).

Implementing functions

And finally, the implementation. In this file, you'll put all the code that directly interacts with Thunderbird UI or components. Start by creating an object with the same name as your api at the top level. (Remember to use `var myapi` or `this.myapi`, not `let myapi` or `const myapi`.)

The object has a function `getAPI` which returns another object containing your API. Your functions and events are members of this returned object:

```
var myapi = class extends ExtensionCommon.ExtensionAPI {
  getAPI(context) {
    return {
      myapi: {
        async sayHello(name) {
          Services.wm.getMostRecentWindow("mail:3pane").alert("Hello " + name + "!");
        },
      },
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
};

```

(Note that the `sayHello` function is an async function, and `alert` blocks until the prompt is closed. If you call `browser.myapi.sayHello()`, it would return a `Promise` that doesn't resolve until the user closes the alert.)

Implementing events

The code for events is more complicated, but the pattern is the same every time. The interesting bit is the `register` function, with the argument named `fire` in this example. Any call to `fire.async` will notify listeners that the event fired with the arguments you used.

In `register`, add event listeners, notification observers, or whatever else is needed. `register` runs when the extension calls `browser.myapi.onToolbarClick.addListener`, and returns a function that removes the listeners and observers. This returned function runs when the extension calls `browser.myapi.onToolbarClick.removeListener`, or shuts down.

```

var myapi = class extends ExtensionCommon.ExtensionAPI {
  getAPI(context) {
    return {
      myapi: {
        onToolbarClick: new ExtensionCommon.EventManager({
          context,
          name: "myapi.onToolbarClick",
          register(fire) {
            function callback(event, id, x, y) {
              return fire.async(id, x, y);
            }

            windowListener.add(callback);
            return function() {
              windowListener.remove(callback);
            };
          },
        }).api(),
      }
    }
  }
};

```

Using folder and message types

The built-in schema define some common objects that you may wish to return, namely *MailFolder*, *MessageHeader*, and *MessageList*.

To use these types, interact with the `folderManager` or `messageManager` objects which are members of the `context.extension` object passed to `getAPI`:

```

// Get an nsIMsgFolder from a MailFolder:
let realFolder = context.extension.folderManager.get(accountId, path);

// Get a MailFolder from an nsIMsgFolder:
context.extension.folderManager.convert(realFolder);

```

(continues on next page)

(continued from previous page)

```
// Get an nsIMsgDBHdr from a MessageHeader:
let realMessage = context.extension.messageManager.get(messageId);

// Get a MessageHeader from an nsIMsgDBHdr:
context.extension.messageManager.convert(realMessage);

// Start a MessageList from an array or enumerator of nsIMsgDBHdr:
context.extension.messageManager.startMessageList(realFolder.messages);
```

Using tabs and windows

To access tabs or windows using the ID values from the built-in APIs, use the `tabManager` or `windowManager` objects. These have functions similar to, but not the same as, the APIs:

```
// Get a real tab from a tab ID:
let tabObject = context.extension.tabManager.get(tabId);
let realTab = tabObject.nativeTab;
let realTabWindow = tabObject.window;

// Get a tab ID from a real tab:
context.extension.tabManager.getWrapper(realTab).id;

// Query tabs: (note this returns a Generator, not an array like the API)
context.extension.tabManager.query(queryInfo);
```

“Tabs” are a bit weird. For a tab on the main Thunderbird window, the `nativeTab` property is the `tabInfo` object you’d get from that window’s `<tabmail>`. For a tab *not* on the main window, e.g. a “tab” representing the message composition window, both `nativeTab` and `window` properties refer to the window itself.

```
// Get a real window from a window ID:
let windowObject = context.extension.windowManager.get(windowId);
let realWindow = windowObject.window;

// Get a window ID from a real window:
context.extension.windowManager.getWrapper(realWindow).id;

// Get all windows: (note this returns a Generator, not an array like the API)
context.extension.windowManager.getAll();
```

For more things you could use on `tabObject` or `windowObject` in the examples above, see the [Tab](#), [TabMailTab](#), and [Window](#) classes in the source code.

21.1.2 Experiment API Generator

Try the [Experiment Generator](#) to quickly get started making a WebExtension experiment. It doesn’t cover all the possibilities, but should be useful for most use-cases or learning how things work.

21.1.3 Getting your API added to Thunderbird

If you think your API would be useful to other extensions, consider having it added to Thunderbird. [File a bug in the Extensions API component](#) and add your schema and implementation files as attachments.

21.2 Working with Message Lists

Mail folders could contain a *lot* of messages: thousands, tens of thousands, or even hundreds of thousands.

It would be a very bad idea to deal with that many messages at once, so the WebExtensions APIs split any response that could contain many messages into pages (or chunks, or groups, or whatever...). The default size of each page is 100 messages, although this could change and you **must not** rely on that number.

Each page is an object with two properties: `id`, and `messages`. To get the next page, call `continueList(messageListId)` with the `id` property as an argument:

```
let page = await browser.messages.list(folder);
// Do something with page.messages.

while (page.id) {
  page = await browser.messages.continueList(page.id);
  // Do something with page.messages.
}
```

You could also wrap this code in a generator for ease-of-use:

```
async function* listMessages(folder) {
  let page = await browser.messages.list(folder);
  for (let message of page.messages) {
    yield message;
  }

  while (page.id) {
    page = await browser.messages.continueList(page.id);
    for (let message of page.messages) {
      yield message;
    }
  }
}
```


CHAPTER 22

Where To Get Help

- developer.thunderbird.net for information about building Thunderbird, and creating extensions.
- [Add-ons message board](#) for Thunderbird add-on developers to ask questions and share knowledge.
- [Bugzilla](#) for bug reports and feature requests.
- developer.mozilla.org for Firefox WebExtensions, on which Thunderbird's are based.