Filter

Internationalization

In this article

The WebExtensions API has a rather handy module available for internationalizing extensions — i18n. In this article we'll explore its features and provide a practical example of how it works. The i18n system for extensions built using WebExtension APIs is similar to common JavaScript libraries for i18n such as i18n.js .

Note: The example extension featured in this article — notify-link-clicks-i18n — is available on GitHub. Follow along with the source code as you go through the sections below.

Anatomy of an internationalized extension

An internationalized extension can contain the same features as any other extension — background scripts, content scripts, etc. — but it also has some extra parts to allow it to switch between different locales. These are summarized in the following directory tree:

- extension-root-directory/
  - _locales
    - en
      - messages.json
    - English messages (strings)
    - de
      - messages.json
    - German messages (strings)
    - etc.
  - manifest.json
    - locale-dependent metadata
  - myJavascript.js
    - JavaScript for retrieving browser locale, locale-specific messages, etc.
  - myStyles.css
    - locale-dependent CSS

Let's explore each of the new features in turn — each of the below sections represents a step to follow when internationalizing your extension.

Providing localized strings in _locales

Note: You can look up language subtags using the *Find* tool on the Language subtag lookup page . Note that you need to search for the English name of the language.

Every i18n system requires the provision of strings translated into all the different locales you want to support. In extensions, these are contained within a directory called `_locales` , placed inside the extension root. Each individual locale has its strings (referred to as messages) contained within a file called `messages.json` , which is placed inside a subdirectory of `_locales` , named using the language subtag for that locale's language.

Note that if the subtag includes a basic language plus a regional variant, then the language and variant are conventionally separated using a hyphen: for example, "en-US". However, in the directories under `_locales`, **the separator must be an underscore**: "en_US".

So for example, in our sample app we have directories for "en" (English), "de" (German), "nl" (Dutch), and "ja" (Japanese). Each one of these has a `messages.json` file inside it.

Let's now look at the structure of one of these files (_locales/en/messages.json):

JSON

```
{
  "extensionName": {
    "message": "Notify link clicks i18n",
    "description": "Name of the extension."
  },

  "extensionDescription": {
    "message": "Shows a notification when the user clicks on links.",
    "description": "Description of the extension."
  },

  "notificationTitle": {
    "message": "Click notification",
    "description": "Title of the click notification."
  },

  "notificationContent": {
    "message": "You clicked $URL$.",
    "description": "Tells the user which link they clicked.",
    "placeholders": {
      "url": {
        "content": "$1",
        "example": "https://developer.mozilla.org"
      }
    }
  }
}
```

This file is standard JSON — each one of its members is an object with a name, which contains a `message` and a `description`. All of these items are strings; `$URL$` is a placeholder, which is replaced with a substring at the time the `notificationContent` member is called by the extension. You'll learn how to do this in the Retrieving message strings from JavaScript section.

> **Note:** You can find much more information about the contents of `messages.json` files in our Locale-Specific Message reference.

Internationalizing manifest.json

There are a couple of different tasks to carry out to internationalize your manifest.json.

Retrieving localized strings in manifests

Your manifest.json includes strings that are displayed to the user, such as the extension's name and description. If you internationalize these strings and put the appropriate translations of them in messages.json, then the correct translation of the string will be displayed to the user, based on the current locale, like so.

To internationalize strings, specify them like this:

JSON

```
"name": "__MSG_extensionName__",
"description": "__MSG_extensionDescription__",
```

Here, we are retrieving message strings dependent on the browser's locale, rather than just including static strings.

To call a message string like this, you need to specify it like this:

1. Two underscores, followed by
2. The string "MSG", followed by
3. One underscore, followed by
4. The name of the message you want to call as defined in `messages.json`, followed by
5. Two underscores

```
__MSG_ + messageName + __
```

Specifying a default locale

Another field you should specify in your manifest.json is default_locale:

JSON

```
"default_locale": "en"
```

This specifies a default locale to use if the extension doesn't include a localized string for the browser's current locale. Any message strings that are not available in the browser locale are taken from the default locale instead. There are some more details to be aware of in terms of how the browser selects strings — see Localized string selection.

Locale-dependent CSS

Note that you can also retrieve localized strings from CSS files in the extension. For example, you might want to construct a locale-dependent CSS rule, like this:

CSS

```
header {
  background-image: url(../images/__MSG_extensionName__/header.png);
}
```

This is useful, although you might be better off handling such a situation using Predefined messages.

Retrieving message strings from JavaScript

So, you've got your message strings set up, and your manifest. Now you just need to start calling your message strings from JavaScript so your extension can talk the right language as much as possible. The actual i18n API is pretty simple, containing just four main methods:

- You'll probably use `i18n.getMessage()` most often — this is the method you use to retrieve a specific language string, as mentioned above. We'll see specific usage examples of this below.
- The `i18n.getAcceptLanguages()` and `i18n.getUILanguage()` methods could be used if you needed to customize the UI depending on the locale — perhaps you might want to show preferences specific to the users' preferred languages higher up in a prefs list, or display cultural information relevant only to a certain language, or format displayed dates appropriately according to the browser locale.
- The `i18n.detectLanguage()` method could be used to detect the language of user-submitted content, and format it appropriately.

In our notify-link-clicks-i18n example, the background script contains the following lines:

JS

```
let title = browser.i18n.getMessage("notificationTitle");
let content = browser.i18n.getMessage("notificationContent", message.url);
```

The first one just retrieves the `notificationTitle message` field from the available `messages.json` file most appropriate for the browser's current locale. The second one is similar, but it is being passed a URL as a second parameter. What gives? This is how you specify the content to replace the `$URL$` placeholder we see in the `notificationContent message` field:

JSON

```
"notificationContent": {
  "message": "You clicked $URL$.",
  "description": "Tells the user which link they clicked.",
  "placeholders": {
    "url" : {
      "content" : "$1",
      "example" : "https://developer.mozilla.org"
    }
  }
}
```

The `"placeholders"` member defines all the placeholders, and where they are retrieved from. The `"url"` placeholder specifies that its content is taken from `$1`, which is the first value given inside the second parameter of `getMessage()`. Since the placeholder is called `"url"`, we use `$URL$` to call it inside the actual message string (so for `"name"` you'd use `$NAME$`, etc.) If you have multiple placeholders, you can provide them inside an array that is given to `i18n.getMessage()` as the second parameter — `[a, b, c]` will be available as `$1`, `$2`, and `$3`, and so on, inside `messages.json`.

Let's run through an example: the original `notificationContent` message string in the `en/messages.json` file is

```
You clicked $URL$.
```

Let's say the link clicked on points to `https://developer.mozilla.org`. After the `i18n.getMessage()` call, the contents of the second parameter are made available in messages.json as `$1`, which replaces the `$URL$` placeholder as defined in the `"url"` placeholder. So the final message string is

```
You clicked https://developer.mozilla.org.
```

Direct placeholder usage

It is possible to insert your variables (`$1`, `$2`, `$3`, etc.) directly into the message strings, for example we could rewrite the above `"notificationContent"` member like this:

JSON

```
"notificationContent": {
  "message": "You clicked $1.",
  "description": "Tells the user which link they clicked."
}
```

This may seem quicker and less complex, but the other way (using `"placeholders"`) is seen as best practice. This is because having the placeholder name (e.g. `"url"`) and example helps you to remember what the placeholder is for — a week after you write your code, you'll probably forget what `$1` – `$8` refer to, but you'll be more likely to know what your placeholder names refer to.

Hardcoded substitution

It is also possible to include hardcoded strings in placeholders, so that the same value is used every time, instead of getting the value from a variable in your code. For example:

JSON

```
"mdn_banner": {
  "message": "For more information on web technologies, go to $MDN$.",
  "description": "Tell the user about MDN",
  "placeholders": {
    "mdn": {
      "content": "https://developer.mozilla.org/"
    }
  }
}
```

In this case we are just hardcoding the placeholder content, rather than getting it from a variable value like `$1`. This can sometimes be useful when your message file is very complex, and you want to split up different values to make the strings more readable in the file, plus then these values could be accessed programmatically.

In addition, you can use such substitutions to specify parts of the string that you don't want to be translated, such as person or business names.

Localized string selection

Locales can be specified using only a language code, like `fr` or `en`, or they may be further qualified with a region code, like `en_US` or `en_GB`, which describes a regional variant of the same basic language. When you ask the i18n system for a string, it will select a string using the following algorithm:

1. if there is a `messages.json` file for the exact current locale, and it contains the string, return it.
2. Otherwise, if the current locale is qualified with a region (e.g. `en_US`) and there is a `messages.json` file for the regionless version of that locale (e.g. `en`), and that file contains the string, return it.
3. Otherwise, if there is a `messages.json` file for the `default_locale` defined in the `manifest.json`, and it contains the string, return it.
4. Otherwise return an empty string.

Take the following example:

- extension-root-directory/
  - _locales
    - en_GB
      - messages.json
        - `{ "colorLocalized": { "message": "colour", "description": "Color." }, /* … */ }`

      en
      - messages.json
        - `{ "colorLocalized": { "message": "color", "description": "Color." }, /* … */ }`
    - fr
      - messages.json
        - `{ "colorLocalized": { "message": "couleur", "description": "Color." }, /* … */}`

Suppose the `default_locale` is set to `fr`, and the browser's current locale is `en_GB`:

- If the extension calls `getMessage("colorLocalized")`, it will return "colour".
- If "colorLocalized" were not present in `en_GB`, then `getMessage("colorLocalized")`, would return "color", not "couleur".

Predefined messages

The i18n module provides us with some predefined messages, which we can call in the same way as we saw earlier in Retrieving localized strings in manifests and Locale-dependent CSS. For example:

```
__MSG_extensionName__
```

Predefined messages use exactly the same syntax, except with `@@` before the message name, for example

```
__MSG_@@ui_locale__
```

The following table shows the different available predefined messages:

Message name	Description
`@@extension_id`	The extension's internally-generated UUID. You might use this string to construct URLs for resources inside the extension. Even unlocalized extensions can use this message. You can't use this message in a manifest file. Also note that this ID is *not* the add-on ID returned by `runtime.id`, and that can be set using the browser_specific_settings key in manifest.json. It's the generated UUID that appears in the add-on's URL. This means that you can't use this value as the `extensionId` parameter to `runtime.sendMessage()`, and can't use it to check against the `id` property of a `runtime.MessageSender` object.
`@@ui_locale`	The current locale; you might use this string to construct locale-specific URLs.
`@@bidi_dir`	The text direction for the current locale, either "ltr" for left-to-right languages such as English or "rtl" for right-to-left languages such as Arabic.
`@@bidi_reversed_dir`	If the `@@bidi_dir` is "ltr", then this is "rtl"; otherwise, it's "ltr".
`@@bidi_start_edge`	If the `@@bidi_dir` is "ltr", then this is "left"; otherwise, it's "right".
`@@bidi_end_edge`	If the `@@bidi_dir` is "ltr", then this is "right"; otherwise, it's "left".

Going back to our earlier example, it would make more sense to write it like this:

CSS

```
header {
  background-image: url(../images/__MSG_@@ui_locale__/header.png);
}
```

Now we can just store our local specific images in directories that match the different locales we are supporting — en, de, etc. — which makes a lot more sense.

Let's look at an example of using `@@bidi_*` messages in a CSS file:

CSS

```
body {
  direction: __MSG_@@bidi_dir__;
}

div#header {
  margin-bottom: 1.05em;
  overflow: hidden;
  padding-bottom: 1.5em;
  padding-__MSG_@@bidi_start_edge__: 0;
  padding-__MSG_@@bidi_end_edge__: 1.5em;
```

```
  position: relative;
}
```

For left-to-right languages such as English, the CSS declarations involving the predefined messages above would translate to the following final code lines:

CSS

```
direction: ltr;
padding-left: 0;
padding-right: 1.5em;
```

For a right-to-left language like Arabic, you'd get:

CSS

```
direction: rtl;
padding-right: 0;
padding-left: 1.5em;
```

Testing out your extension

To test your extension's localization, you use Firefox or Firefox Beta , the Firefox builds in which you can install language packs.

Then, for each locale supported in the extension you want to test, follow the instructions to Use Firefox in another language to switch the Firefox UI language. (If you know your way around Settings, under Language, use Set Alternatives.)

Once Firefox is running in your test language, install the extension temporarily . After installing your extension, in `about:debugging` , if you've set up your extension correctly, you see the extension listed with its icon, name, and description in the chosen language. You can also see the localized extension details in `about:addons` . Now exercise the extension's features to ensure the translations you need are in place.

If you'd like to try this process out, you can use the notify-link-clicks-i18n extension. Set up Firefox to display one of the languages supported in this example (German, Dutch, or Japanese). Load the extension and go to a website. Click a link to see the translated version of the notification reporting the link's URL.

Found a content problem with this page?

- Edit the page on GitHub.
- Report the content issue.
- View the source on GitHub.

Want to get more involved? Learn how to contribute.

This page was last modified on Aug 2, 2023 by MDN contributors.

mdn
Your blueprint for a better internet.

MDN

About
Blog
Careers
Advertise with us

Support

Product help
Report an issue

Our communities

MDN Community
MDN Forum
MDN Chat

Developers

Web Technologies
Learn Web Development
MDN Plus
Hacks Blog

Website Privacy Notice
Cookies
Legal
Community Participation Guidelines
Visit Mozilla Corporation's not-for-profit parent, the Mozilla Foundation.
Portions of this content are ©1998–2024 by individual mozilla.org contributors. Content available under a Creative Commons license.