Persona Switcher
Embedded WebExtension Specifications Document

The Team That Shall Not Be Named (T^3SNBN)

Jason Gould

Jim Kim

Stefany Maldonado

Simeon Martinez

Trever Mock

Dustin Porter

Luz Rodriguez

Steve Beaty

CS4250-001 Software Engineering

03/01/17

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to define the processes involved in embedding a WebExtension within *Persona Switcher*.

1.2 Scope

This document will define the process of embedding the WebExtension within the bootstrapped add-on and the process of migrating functionality from the add-on to the embedded WebExtension.

1.3 References

[1] Mozilla Developer Network. “Embedded WebExtensions.” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Embedded_WebExtensions

[2] Mozilla Developer Network. “Runtime.” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/runtime

[3] Mozilla Developer Network. “Internationalization.” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Internationalization#Anatomy_of_an_internationalized_WebExtension

[4] Mozilla Developer Network “Implement a settings page” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Implement_a_settings_page

[5] Mozilla Developer Network “browser_action” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/browser_action

[6] Mozilla Developer Network “storage” [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/storage

1.4 Overview

The remainder of this document includes 4 chapters. The second chapter defines the process of setting up the embedded WebExtension within *Persona Switcher*. The third chapter defines the process of migrating *Persona Switcher*s' functionality to the embedded WebExtension. The fourth chapter addresses the issue of functionality that cannot be ported to a web extension and the underlying factors which prevent migration.

2. Setting Up the Embedded WebExtension

This section defines the process of setting up the embedded WebExtension within *Persona Switcher*. The online resource by Mozilla titled "Embedded WebExtensions" [1] is used as the primary reference for all information in this section.

2.1 Add a line in install.rdf

Include "*hasEmbeddedWebExtension*" in the install.rdf file containing the value *true* as shown below:

```
<em:hasEmbeddedWebExtension>true</em:hasEmbeddedWebExtension>
```

2.2 Create the WebExtension directory

The embedded WebExtension lives in a top-level directory called "webextension" inside the add-on as illustrated by the example below:

```
my-boostrapped-addon/
  chrome/
        webextension/
        manifest.json
        background.js
        ...
        bootstrap.js
        chrome.manifest
        install.rdf
```

2.3 Starting the embedded WebExtension

The embedded WebExtension must be started by the legacy add-on. This is done by modifying the bootstrap.js startup() function.

The data argument passed to the startup() function gets the webExtension property. The example below shows how the startup() function needs to be modified:

```
// bootstrapped add-on
function startup({webExtension}) {
...
```

2.4 Exchanging Messages

The runtime APIs [2] can be used to exchange messages between the legacy add-on and embedded webExtension. There are two ways of exchanging messages using the runtime APIs:

- Connectionless Messaging
- Connection-Oriented Messaging

Connectionless Messaging is suited for one-off messages where the legacy add-on recieves (and optionally responds to) a message. Connection-Oriented Messaging is suited for longer lived connections where the legacy add-on can accept connection requests and communicate with the WebExtension over runtime.Port.

The legacy add-on can't initiate communications for either of these methods. The WebExtension must be the one to initiate a communication by sending a message or requesting a connection to the legacy add-on.

Connectionless Messaging

The embedded WebExtension can use *runtime.sendMessage()* to send a single message to the legacy add-on:

```
browser.runtime.sendMessage("message-from-webextension").then(reply => {
  if (reply) {
        console.log("response from legacy add-on: " + reply.content);
  }
});
```

The legacy add-on can then receive (and optionally respond to) this message using the *runtime.onMessage* object:

```
// bootstrapped add-on
function startup({webExtension}) {
  // Start the embedded webextension.
  webExtension.startup().then(api => {
        const {browser} = api;
    browser.runtime.onMessage.addListener((msg, sender, sendReply) => {
        if (msg == "message-from-webextension") {
        sendReply({
        content: "reply from legacy add-on"
        });
        }
        });
  });
}
```

Connection-Oriented Messaging

To set up a longer-lived connection, the WebExtension can use *runtime.connect()*:

```
var port = browser.runtime.connect({name: "connection-to-legacy"});

port.onMessage.addListener(function(message) {
  console.log("Message from legacy add-on: " + message.content);
});
```

The legacy add-on can listen for connection attempts using *runtime.onConnect* and both sides can use the resulting *runtime.Port* to exchange messages:

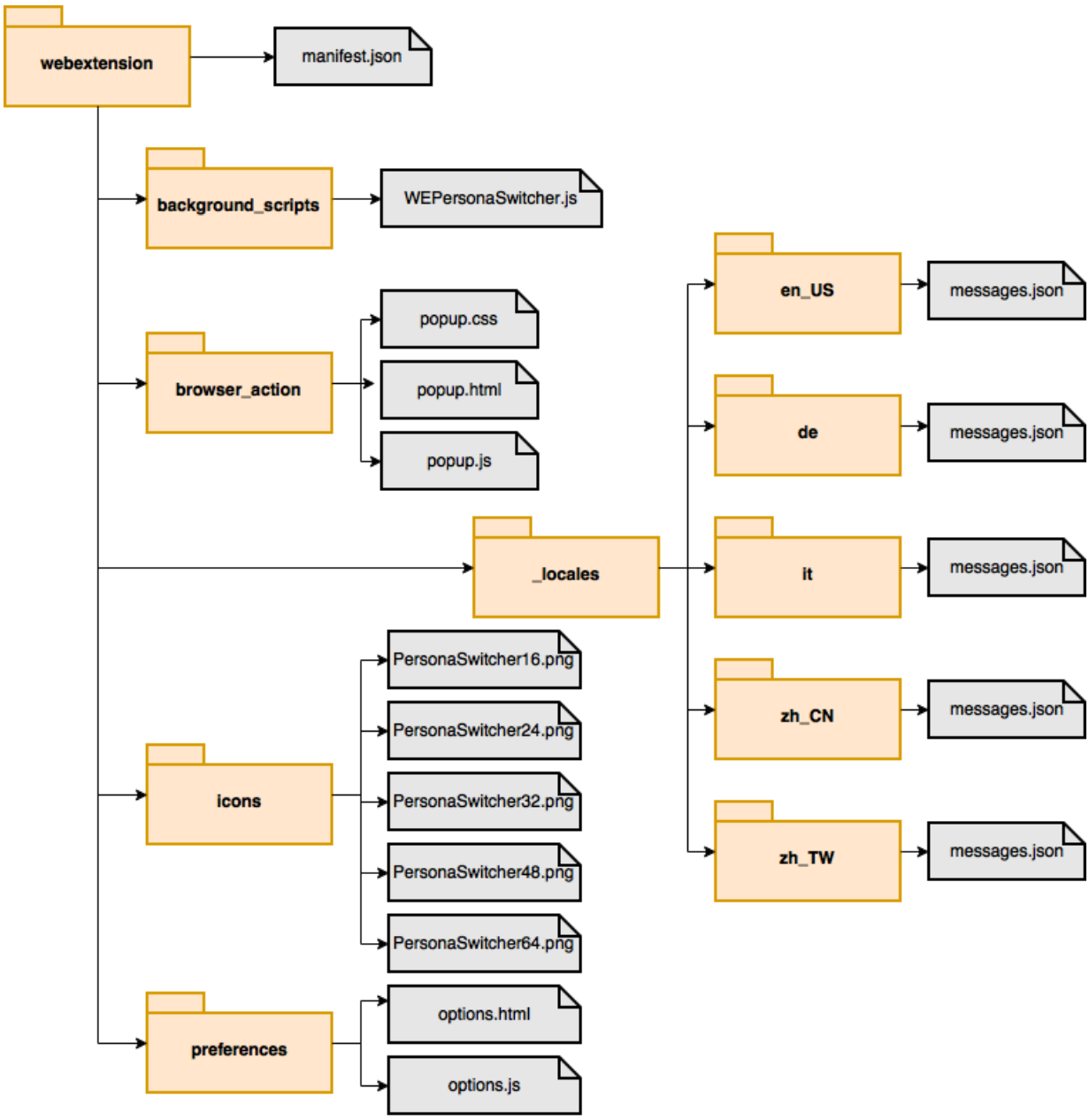
```
function startup({webExtension}) {
  // Start the embedded webextension.
  webExtension.startup().then(api => {
        const {browser} = api;
        browser.runtime.onConnect.addListener((port) => {
        port.postMessage({
        content: "content from legacy add-on"
        });
        });
```

```
});
}
```

3. Functionality Porting

This section defines the various aspects of migrating *Persona Switchers'* functionality to the embedded WebExtension.

3.1 Web Extension Directory Structure and Included Files



webextension
The base directory for the embedded WebExtension and home of the manifest.json file.

background_scripts
These scripts are loaded as the extension is loaded and stay until the add-on is uninstalled or disabled.

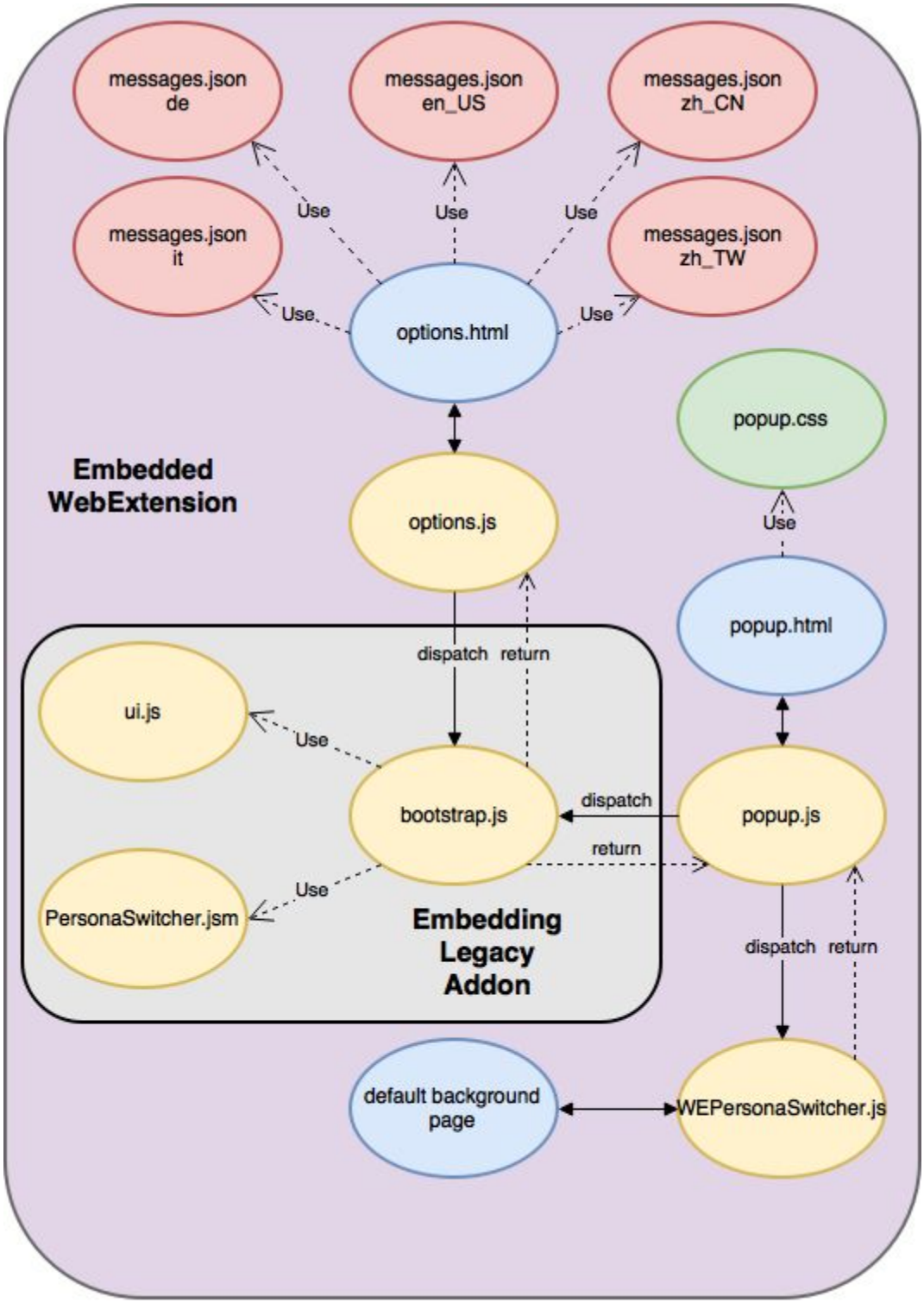
browser_action
This is a button that can be added to the browser toolbar. The end user can then click this button to interact with the extension. A popup can be defined using HTML, CSS, and JavaScript. The popup will be shown when the button is clicked. The user will then be able to interact with the popup and will close when the user clicks anywhere outside the popup.

preferences
This page defines the preferences also known as the options for the add-on or web extension that an end-user can change. The access to the options page can be reached by the end user through the add-ons manager.

_locales
Contains the files necessary for internationalization. Each supported language appears as a subdirectory containing a single messages.json file that defines all of the localized strings used by Persona Switcher for that locale.

icons
All of the icons used by Persona Switcher. This includes the icon used for the browser action as well as the icon that appears when Persona Switcher is displayed in the addons page.

3.2 Internal Control Flow



3.3 Manifest

The manifest.json file is the only file that every WebExtension must contain. The file specifies basic metadata about the extension such as the name, version, and specific aspects of the extension's functionality such as background and content scripts along with browser actions. The manifest.json file fills the roles of both the chrome.manifest and install.rdf files in the legacy code for the embedded WebExtension. As is currently typical for numerous aspects of WebExtensions, the manifest.json file is significantly simpler than it's predecessors and it's functionality is in some ways more limited. An example of this limited functionality is that there is no method for listing contributors or translators and the developer/author key is limited to a single entry. Likewise, an example of the simplification inherent in the manifest.json file is the removal of the localized tags found in the legacy install.rdf file. These tags have been abstracted away and localization of the manifest is handled by the localization API i18n.

As for the structure of the manifest.json file there are only a few keys that are mandatory, and they are "manifest_version", "version", and "name". Additionally, the "default_locale" key must be present if the "_locales" directory is present. The last (sometimes) mandatory key is the "Applications" key, and it is a special case. This key is not supported in Google Chrome but is mandatory in Firefox before Firefox 48. However, as the minimum version for numerous aspects of the functionality that will be employed in Persona Switcher is Firefox version 52, this does not pertain to Persona Switcher. The more relevant detail is that the "Applications" key is used to define the addon-ID. Because Firefox does not treat embedded WebExtensions as separate addons, the addon-ID specified by the Applications key will be ignored if added in the current architecture. However, once Persona Switcher is fully ported to the WebExtension and removed from the embedding legacy addon, this key must be added (if not already present) with the value of the ID the same as the ID of the legacy addon. It is in this way that Persona Switcher can be recognized by addons.mozilla.org as an update of the legacy addon.

> **Note:** This is a one way street and once the ID has been associated with the addon as a WebExtension, it can not be undone. For this reason, it may be desirable to register the WebExtension version of Persona Switcher with a new addon-ID in order to allow the legacy addon to remain accessible from mozilla.addons.org

"Manifest_version": a mandatory key whose type is a number. The current value of the type must always be 2. *Example:* "manifest_version": 2

“Name”: a mandatory key whose type is a string.

“Version”: a mandatory key whose type is a string. The syntax for Chrome’s version is more restrictive than Firefox and the values from Chrome’s version will always stay valid for Firefox. However, values that are valid for Firefox will not be valid for Chrome. Example: “version”: “0.1”

3.4 Toolbar Button

Because WebExtensions do not have direct access to the underlying XUL of the browser’s chrome, they must rely on a set of JavaScript API’s to add content to the chrome. The browserAction javascript API provides the ability to add an addon specified button to the navigation bar and will be replacing the toolbar button used in the legacy code. In order to add the browser action to the navigation toolbar, the manifest.json must contain the browser_action key. The browser_action folder will contain new files popup.css, popup.html and popup.js in which the old legacy code functionality will be updated and implemented in the popup.js file. Since we are supplying the popup files, we should be able to handle all associated logic of the user interactions of the toolbar-button in these files instead of relying on the background scripts (as mentioned from the above link).

-The popup.css file will be used for styling the browser_action window which will be populated with html elements containing all the addon names (and optionally icons).

-The popup.html file will be a basic html document that references the popup.css and loads the popup.js file.

-The popup.js file will populate the browser action window with the names (and icons) of installed themes as html elements. It will also handle the user’s interactions once the browser action button is clicked. This logic will be (in part) converted from the ui.js file of the legacy code.

Sample code:

```
 "browser_action": {
        "browser_style": true,
   "default_icon": {
     "16": "icons/PersonaSwitcher16.png",
     "32": "icons/PersonaSwitcher32.png",
```

```
    "64": "icons/PersonaSwitcher64.png"
  },
  "default_title": "PersonaSwitcherMenu",
  "default_popup": "browser_action/popup.html"
 },
```

3.5 Internationalization

This section will give relevant information for migrating Persona Switcher's legacy code internationalization to the embedded WebExtension. Information for this section was taken from the mozilla website internationalization section [3].

3.5.1 Creating the directories and messages.json files

1. Create a *_locales* directory inside the WebExtension root directory.
2. Inside the *_locales* directory should contain directories for each region. Language and variant are conventionally separated using an underscore: for example "en_US".
3. Each directory contains a messages.json file which will contain all of the string translations for the directories regional language.

The following image outlines the directory setup:

3.5.2 messages.json files

The messages.json files are structured as seen in the image below:

```
{
  "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"
      }
    }
  }
}
```

Each member of the messages.json file is named after the name of the message string being localized.

The "*message*" section contains the internationalized text to be displayed. It is the only required field.

The "*description*" section contains a description of the internationalized text. It is optional.

The "*placeholders*" section is optional and will not be used in migration of Persona Switchers internationalization.

3.5.3 Migrating internationalization functionality

In Persona Switcher's legacy code, internationalization is done in .dtd and .properties files that correspond to each region. Each string translation of these files will be ported to the corresponding messages.json file in the embedded WebExtension.

For example, here is a localized string in about.dtd for the word 'about':
<!ENTITY about "Über:">

The messages.json file will include this localized string as follows (description optional):
"about" { "messages": "Über", "description": "The word 'About'." },

The following tables map all labels from the .dtd and.properties files to the new messages.json file.

Personaswitcher.properties	**messages.json**
personaswitcher.noPersonas	ps_noPersonas
personaswitcher-menu.label	ps_menuLabel
personaswitcher-button.label	ps_buttonLabel
personaswitcher.tooltip	ps_tooltip
personaswitcher.default	ps_default

about.dtd	**messages.json**
about	about
version	version
createdBy	createdBy
homepage	homepage

options.dtd	**messages.json**

options.title	options_title
options.pane1	options_pane1
options.kbshortcuts.title	options_kbshortcuts_title
options.kbshortcuts.default	options_kbshortcuts_default
options.kbshortcuts.rotate	options_kbshortcuts_rotate
options.kbshortcuts.toggle	options_kbshortcuts_toggle
options.kbshortcuts.auto	options_kbshortcuts_auto
options.kbshortcuts.menushortcuts	options_kbshortcuts_menushortcuts
options.kbshortcuts.access	options_kbshortcuts_access
options.kbshortcuts.activate	options_kbshortcuts_activate
options.kbshortcuts.key	options_kbshortcuts_key
options.kbshortcuts.shift	options_kbshortcuts_shift
options.kbshortcuts.control	options_kbshortcuts_control
options.kbshortcuts.alt	options_kbshortcuts_alt
options.kbshortcuts.meta	options_kbshortcuts_meta
options.kbshortcuts.accel	options_kbshortcuts_accel
options.kbshortcuts.os	options_kbshortcuts_os
options.options.title	options_options_title
options.autoevery.switch	options_autoevery_switch
options.autoevery.minutes	options_autoevery_minutes
options.random.desc	options_random_desc
options.startup-switch.desc	options_startupSwitch_desc
options.toolbox-minheight.desc	options_toolboxMinHeight_desc

options.preview.desc	options_preview_desc
options.preview-delay.desc	options_previewDelay_desc
options.icon-preview.desc	options_iconPreview_desc
options.notification-workaround.desc	options_notificationWorkaround_desc
options.menus.title	options_menus_title
options.tools-submenu	options_toolsSubmenu
options.main-menubar	options_mainMenuBar

Internationalization information from install.rdf will also be migrated to messages.json. The name and description of the WebExtension will be declared as "*extensionName*", "*extensionShortName*", and "*extensionDescription*" in messages.json.

3.5.4 Internationalizing manifest.json

The manifest.json includes strings that are displayed to the user, such as the WebExtensions name and description. A certain format must be used to internationalize these strings within manifest.json.

```
__MSG_ + messageName + __
```

The two examples below show localized message "*extensionName*" being called from manifest.json:

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

The format can be described as follows:

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

A default locale should also be specified in the manifest.json file.

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

This example will set the default locale to "*en*" assuming a folder does not exist for some browsers current locale.

3.5.5 Retrieving message strings from JavaScript

The i18n API contains four methods in total. The *i18n.getMessage()* method will be used to retrieve a specific language string from JavaScript code.

```
var title = browser.i18n.getMessage("notificationTitle");
```

The i18n API's three other methods, *i18n.getAcceptLanguages()*, *i18n.getUILangguage()*, and *i18n.detectLanguage()* which will not be used in Persona Switchers migration to embedded WebExtension.

3.5.6 Implementing localization in HTML

There currently isn't a direct method of showing localized strings from the HTML portions of a WebExtension. For this reason we will go with the following implementation of internationalization of HTML.

The following function must be called on startup of the corresponding Javascript file for an HTML file implementing internationalization:

```
function localizeHtmlPage()
{
   var objects = document.getElementsByName("i18n");
   for (var j = 0; j < objects.length; j++)
   {
      var obj = objects[j];
      obj.innerHTML = browser.i18n.getMessage(obj.id.toString());
   }
}
```

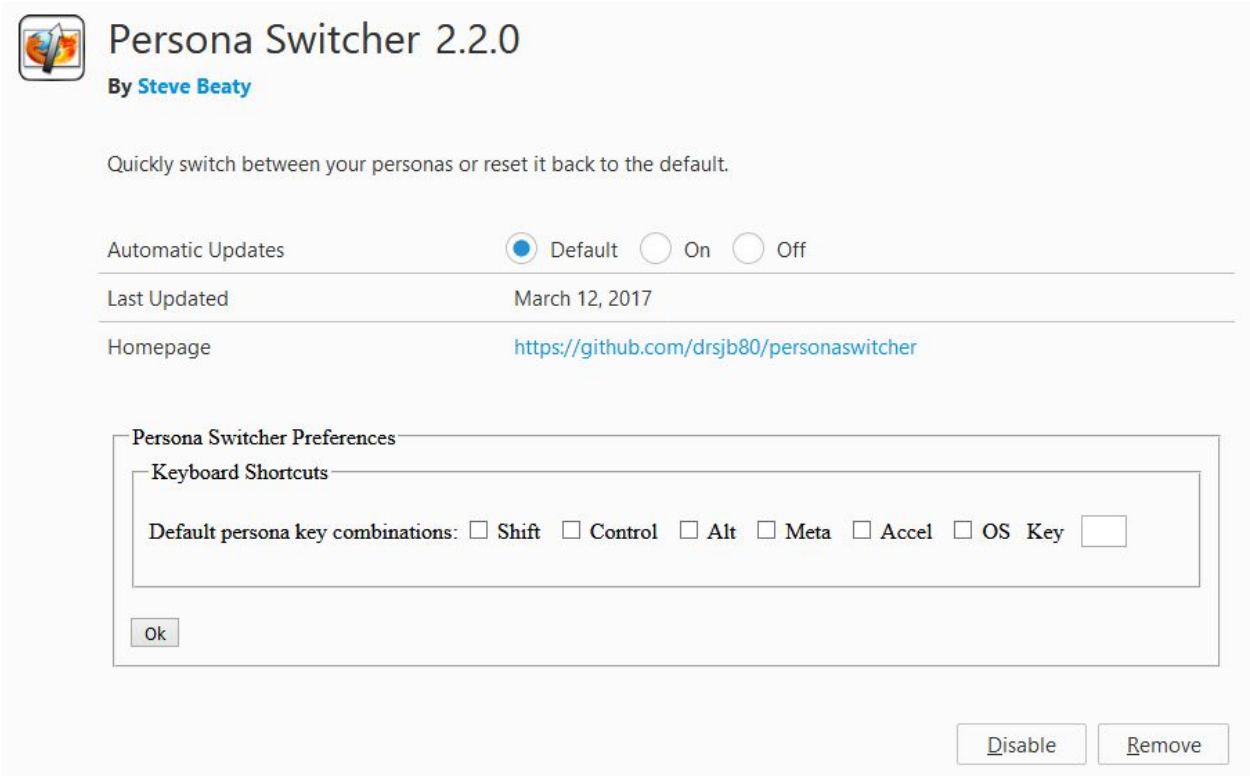
This function gets all the elements containing the name "i18n" and uses the id for each element, which corresponds to a localized string, to populate the element with the correct localized string. The following HTML element acts as an example for putting a localized string in HTML.

<element name="i18n" id="localized_string_name"></element>

It's important to note that everything inside the element will be rewritten upon being localized. For this reason, the contents of these elements should be empty and any formatting or other HTML information should be done outside of this element.

3.6 Preferences

For a WebExtension, the preferences menu is referred to as the options page or settings page. Options pages define the user interface (UI) for the users to view and enable you to define the preferences for the WebExtension that the user will change. Information for this section was taken from the mozilla website settings page implementation section[4] and the mozilla website storage API section[6]. The options page will show up in the browser's add-on manager, alongside the add-on's name and description. The following image displays an example of what the options page will potentially look like:



3.6.1 Implementing the Options page

Since WebExtensions do not have direct access to the underlying XUL of the browser's chrome, the options.xul file can not be used to display the preferences menu to the user. The following defines the process that will take place to implement the options page:

1. **\chrome\content\options.xul → \webextension\preferences\options.html**

    - The formatting of the preferences menu in the .xul file will have to be converted to an HTML file that displays it and lets the user change the preferences.
2. **\chrome\content\prefs.js → \webextension\preferences\options.js**
    - This file will populate the preferences from storage and update the stored preferences when the user changes them.
    - The storage API will be used to get and set the preferences.
3. **options_ui key and permissions in \webextension\manifest.json**
    - options_ui: This sets an HTML document to be the settings page (also known as the options page) for the add-on. It will show up in the browser's add-on manager, alongside the add-on's name and description.

```
"options_ui": {
  "page": "options.html"
},
```

    - permissions: the storage API needs to be used to store the preferences/settings, so we need to ask permission to use that API.

```
"permissions": ["storage"]
```

3.6.2 Storage API

The options.js file will be implementing the logic for preferences and will consist of the following methods:

- saveOptions()
    - This function will be called when a user makes a change to the preferences in the menu and saves by selecting the "ok" button. An event listener will be created to wait for the "ok" button click on the preferences menu and call the method. Within the saveOptions() method, the storage.local.set() method is called and used to store a preference into storage.

- loadOptions()
    - This function will be called when the preferences menu is loaded. The function will load the stored preferences into the menu and display to the user. An event listener will be created to wait for the "DOMContentLoaded" event and call the method. The "DOMContentLoaded" event is fired when the options.html file has been completely loaded and parsed. Within the loadOptions() method, the storage.local.get() method is called and used to retrieve a preference from storage.

The preferences will be stored and accessible through the Storage API. The following are the methods:

- storage.local.get()
    - Retrieves one or more items from the storage area. In our case, this will be used when we are fetching a value for a specific preference from storage.
    - Returns a Promise, which is a value that allows you to associate handlers with an asynchronous action's eventual success value or failure reason(error). A promise is in one of the following states:
        - Pending: initial state, not fulfilled or rejected
        - Fulfilled: operation completed successfully
        - Rejected: operation failed.
    - The then() method is a method that takes up to two arguments which are callback functions or handlers for the success and failure cases of the Promise. As you can see in the example below, if the Promise state is fulfilled, then it will call the onGot() method, otherwise it calls the onError() method.

```
let gettingItem = browser.storage.local.get(["kitten", "monster", "grapefruit"]);
gettingItem.then(onGot, onError);
```

- storage.local.set()
    - Stores one or more items in the storage area, or updates existing items. In our case, this will be used when the user submits the values of the preferences in the menu by clicking the "ok" button.
    - Returns a Promise, the then() method will handle it with no arguments if it is fulfilled. If it is rejected/failed, then the Promise is rejected with an error message. There are two ways to handle the promise, a then() method that only checks for errors, or a catch() method. The following are examples:

```
// store the objects
let setting = browser.storage.local.set({kitten, monster});
// just check for errors
setting.then(null, onError);
```

```
// store the objects
let setting = browser.storage.local.set({kitten, monster});
// just check for errors
setting.catch(onError);
```

3.6.3 Default Handling

The methods in the option.js script will only handle preferences directly on the form. This means that when Persona Switcher is installed, the values for the preferences will be empty until the user clicks on the “ok” button to save/store them. Due to this, we will handle default preferences in the background script when the add-on is installed.

3.7 Shortcuts (Commands)

https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/commands

In Embedded WebExtension you can specify the shortcut keys via commands in the manifest.json file. These command keys that define one or more keyboard shortcuts are objects themselves. These keyboard shortcuts are defined by a name, combination of keys, and a description.

When a command key is created, each object gets its own specific attributes which are the suggested_key and the description. The suggested_key attribute is also an object with specific attributes such as “default, mac, windows, etc“. Each command key is composed of shortcut values which can accept the format of a key combination (as Media keys are not supported in Firefox). Key combinations comprise of a modifier, secondary modifier, and a key.

- Modifier: is mandatory except for functions keys. Can be any of the following: “Ctrl", "Alt", "Command", “MacCtrl”
- Secondary: is optional, however if implemented, must be “Shift”
- Key: is mandatory and includes one of the following:
    - letters A-Z
    - numbers 0-9
    - Comma, Period, Home, End, PageUp, PageDown, Space, Insert, Delete, Up, Down, Left, and Right
    - Function keys are possible to use but have have a known issue and are best left out.

Because command keys are objects themselves which are immutable once they are coded into the manifest.json file they cannot be changed hence any command keys that are created will remain the same and unchangeable for users to personally customize in the settings preference page of PersonaSwitcher. This process would involve terminating part of the functionality in the ui.js file which involves the modification of shortcut keys such as makeKey() and setKeySet().

3.8 ContextMenu

The current bootstrapped version of PersonaSwitcher implements a popup window on left click of the browser toolbar button and no right click response. The implementation of the context menu into the embedded WebExtension will be an enhancement that would create access to Persona Switchers options by right clicking the browser action. In addition, once the expanded functionality is added to the API, the use of sub menus in the tools menu can be implemented into PersonaSwitcher. Please see 4.1 of this document for further explanation. Implementing context menu would manage the FR-15 (from the SRS doc) issue #4, Quick Access to Settings.

4. Unportable Functionality

This section addresses the fact that various functionalities for WebExtensions are still not fully implemented by Firefox. Thus, there remains certain functionality that cannot be ported and must remain in the legacy code. Which functionality that is, how to deal with it, and the possibility of porting it in the future are all discussed below.

4.1 Custom Menus and Sub-menu's

Currently, the addition of custom menus to the main menu toolbar as well as the addition of sub menus to existing menus on the toolbar (such as the Tools menu), is not possible with the javascript APIs currently available to Firefox WebExtensions. For this reason, it is suggested that these features be left intact in the legacy code. Moving forward, as Firefox continues to expand the javascript APIs available to WebExtension developers, as well as their functionality, these features may become eligible for porting. Notably, there is a bug/enhancement being developed that will expand the functionality of the contextMenu API to allow the addition of sub menu's to the Toolbar menu. This bug/enhancement can be found at https://bugzilla.mozilla.org/show_bug.cgi?id=1268020 . However, there does not currently seem to be any work towards allowing the addition of custom menu's to the main menu toolbar.

4.2 Installed Theme Acquisition and Switching

Currently, the acquisition and activation/deactivation of installed themes is not possible with the javascript APIs currently available to Firefox WebExtensions. For this reason, it is suggested that these features be left intact in the legacy code. Moving forward, as Firefox continues to expand the javascript APIs available to WebExtension developers, as well as their functionality, these features may become eligible for porting. There are two APIs that are scheduled to be developed/enhanced to address these concerns and they can be found at the following links. https://bugzilla.mozilla.org/show_bug.cgi?id=1271481 ,https://bugzilla.mozilla.org/show_bug.cgi?id=1336908

4.3 Shortcut Preferences

Because WebExtensions do not have access to the browser's chrome directly, they are not able to inject the custom keyset needed to implement custom shortcuts. The current solution for WebExtensions is the command API which allows a WebExtension to define new shortcuts in the manifest.json file. However, shortcuts defined in this way are immutable. Thus, in order to preserve the current functionality of allowing users to specify their own shortcut key values via the preferences menu, it is recommended that the shortcut implementation remain in the legacy code at present.

One notable exception is the shortcut for the browser action button. Currently there is no shortcut for the toolbar button in the legacy code (which the browser action is replacing). It is recommended that a shortcut be added for this purpose. Due to the inability of browser actions to be opened programmatically, the only method for implementing this shortcut is with the commands API. Moving forward, as Firefox continues to expand the javascript APIs available to WebExtension developers, as well as their functionality, the ability to port shortcuts while retaining their mutability may become possible. There are two bug reports that are approved which address shortcut mutability and they can be found at the following links.
https://bugzilla.mozilla.org/show_bug.cgi?id=1303384
https://bugzilla.mozilla.org/show_bug.cgi?id=1215061