Web Application Extension Framework

WebExtension

User Guide

Revision History

Version	Date	Author	Description
0.0.1	2/3/2020	Masayuki Otoshi	Document Created

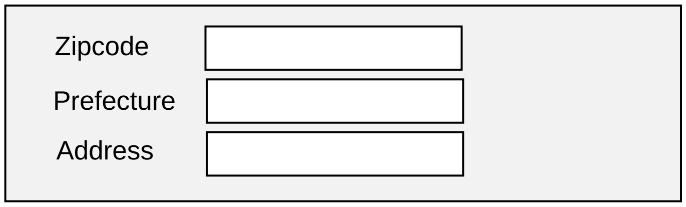
Table of Contents

1 Overview

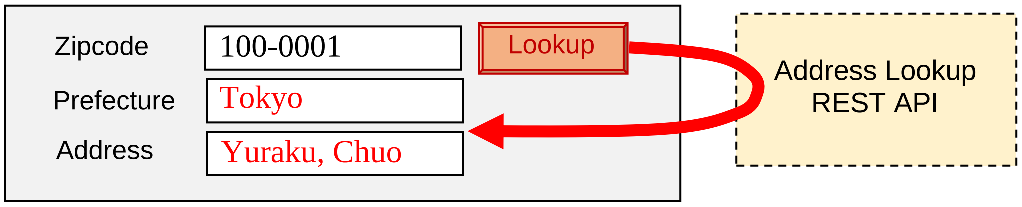
This document is for system integrators and developers who are starting to enhance an existing web application by connecting with other web applications, REST APIs.

1.1 What is WebExtension?

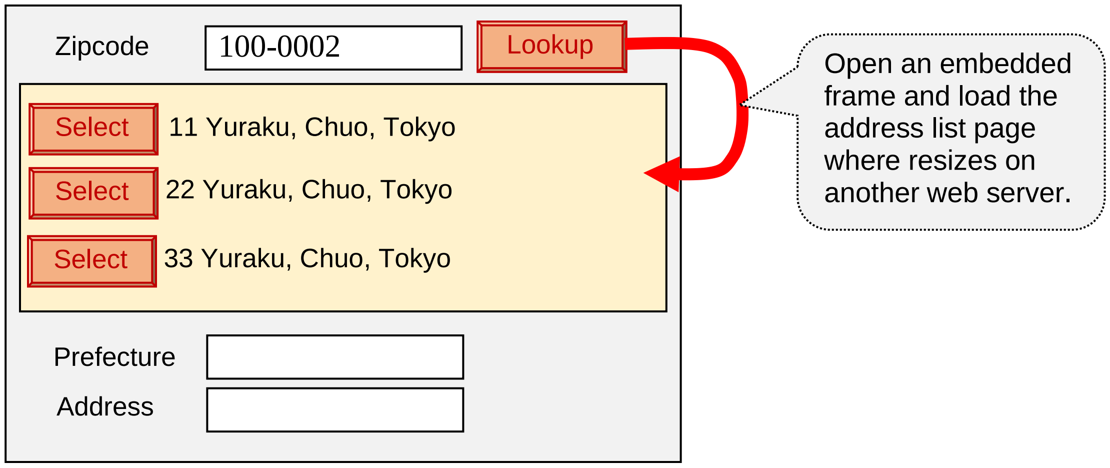
WebExtension is a lightweight framework to customize the browsing web pages. It allows you to enhance the application without modifying source code of the page contents. Suppose there is an address input form in your web application, and you are going to add an address autofill feature on it.



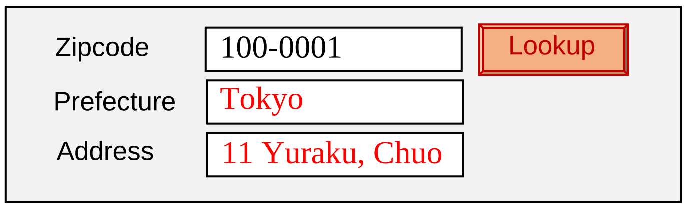
To implement the feature, you will need to change the source code of the page to add a new button labeled "Lookup". The button calls a REST API to lookup an address from the zip code entered. When an address data is responded, the address is populated into the address form.



Normally, the enhancement can be done by modifying the existing code. However, WebExtension enables you to add the feature without modifying the source code of the page. WebExtension only requires inserting <script> tags to import your enhancement code into the base HTML file.
WebExtension can also integrate with another web application having pages. Suppose you have a web application that searches addresses from a zip code and displays a list of addresses matched with the given zip code. You can add a Lookup button, which opens a frame and displays the search results in it.

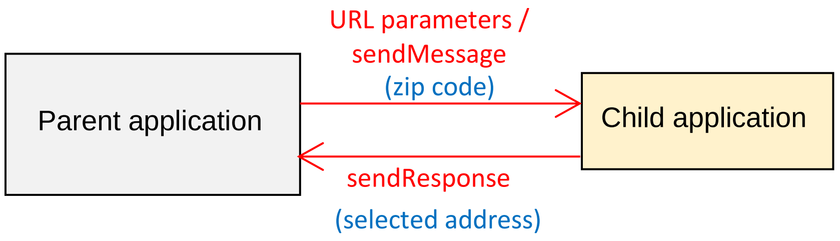


When user selects one of the Select button in the frame, the frame is closed and the selected address is populated into the parent address form.

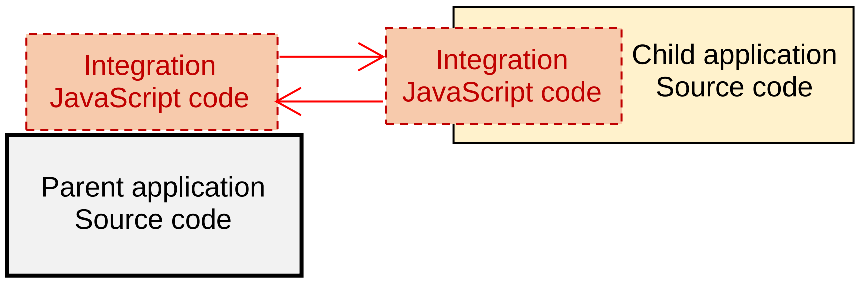


Zipcode	100-0001	Lookup
Prefecture	Tokyo	
Address	11 Yuraku, Chuo	

This type of enhancement can be also done without modifying the source code of the page. WebExtension connects the two applications; parent application (address form) and child application (address list) and allows them to pass messages each other.



The parent application can pass initial data to the child via URL parameters. Also it can send data by calling sendMessage API. The child application can send back data by calling snedResponse API.
On the child application side, integration code to send a response needs to be implemented inside the application. However, parent application can be managed integration JavaScript code separately. It only needs to include <script> tags to load the integration code.



By managing the integration code separately from original application code, you can keep the parent application vendor original without your custom code. It makes you easy apply patches and upgrade the application.

2 Installation

2.1 Prerequisites

Ensure that you have installed the following software on your machine:

- **Java SE Version 8** or above
  https://www.oracle.com/technetwork/java/javase/
- **Chrome browser**
  https://www.google.com/chrome/
- **Apache Tomcat Version 9.0.30** or above (or equivalent web application server)
  http://tomcat.apache.org/

2.2 Download WebExtension

Download a master ZIP file from the WebExtension website below:

https://github.com/web-extension/master/archive/master.zip

Unzip it:

```
master/
├──── doc/
├──── samples/
├──── webextension.jar
├──── WebExtension.js
├──── WebExtensionClient.js
└──── README.md
```

2.3 Setup HttpProxy servlet

Next, configure web.xml in your web application server to enable HttpProxy servlet.

1. Go to /webapps/ROOT/WEB-INF/
2. Open web.xml with your favorite text editor.
3. Add the following entries:

```
<servlet>
   <servlet-name>webextension-httpproxy-servlet</servlet-name>
   <servlet-class>webextension.HttpProxy</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>webextension-httpproxy-servlet</servlet-name>
   <url-pattern>/WebExtensionHttpProxy</url-pattern>
</servlet-mapping>
```

2.4 Run samples

That is all for the installation. As the next step, open sample pages to verify that the installation has done successfully.

1. Copy master/samples into /webapps/ROOT/ folder.
2. Start the TOMCAT server.
3. Open http://loclahost:8080/samples/HelloWorld/index.html with Chrome browser.

You will see HelloWorld sample page and Say Hello button will be added on the page.

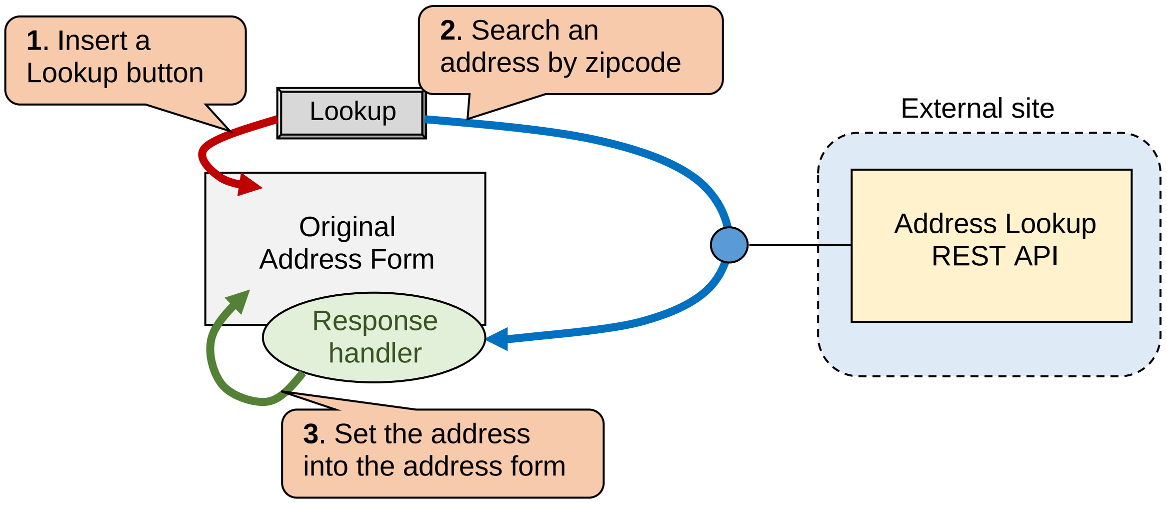
3 Extension with REST API

This chapter shows an example of how to extend your web application with a REST API call provided by third party.

Sample Site:
https://web-extension.github.io/AddressLookup/index.html

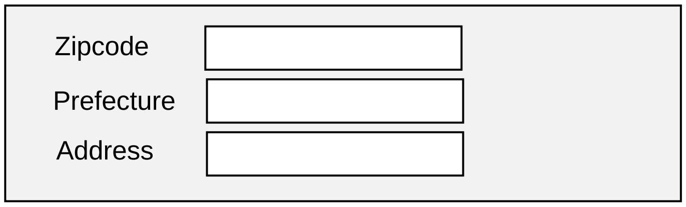
Code:
https://github.com/web-extension/master/tree/master/samples/AddressLookup

This sample site has an address form (Zipcode, Prefecture and Address fields).
Step 1: WebExtension dynamically adds a Lookup button in the form when the page is loaded.
Step 2: After user enters a zipcode and clicks on the Lookup button, it makes a HTTP request against an external site to execute their Address Lookup REST API. The API returns an address in JSON format.
Step 3: When a response is returned, response hander function (a callback function) is fired and set the address returned from REST API into the address form.


1. Insert a Lookup button
2. Search an address by zipcode
Lookup
External site
Original Address Form
Address Lookup REST API
Response handler
3. Set the address into the address form

3.1 Address Form and REST API

Suppose your application has an address form as shown below:



The HTML code of the page is below:

```
<html>
<body>
Zipcode <input id="zipcode" type="text" value=""><br>
Prefecture <input id="pref" type="text" value=""><br>
Address <input id="address" type="text" value="">
</body>
```

```
</html>
```

And you want to extend the form that the prefecture and address are automatically populated when user enters the zipcode. And you found the REST API to lookup an address from a zipcode.

```
https://api.zipaddress.net/?lang=rome&zipcode=<zipcode>
```

The REST API returns, for example, the following address data in JSON format for zipcode 100-0006.

```
{
	"code":200,
	"data":{
		"pref":"TOKYO",
		"address":"CHIYODA-KU YURAKUCHO",
		"fullAddress":"TOKYO CHIYODA-KU YURAKUCHO"
	}
}
```

3.2 Extend the Address Form

WebExtension does not require to change the body HTML code. You just need to add <script> tags in the HTML.

```
<html>
<head>
   <script src="WebExtension.js"></script>
   <script src="AddressLookup.js"></script>
</head>
<body>
Zipcode <input id="zipcode" type="text" value=""><br>
Prefecture <input id="pref" type="text" value=""><br>
Address <input id="address" type="text" value="">
</body>
</html>
```

Your integration code can be described in the AddressLookup.js. Below is a sample implementation to lookup address from a given zipcode.

AddressLookup.js

```
WebExtension.addExtensionRegister(() => {
   const target = document.getElementById('zipcode');
   if (!target) return;
   const button = document.getElementById('myButton');
   if (!button) {
      target.insertAdjacentHTML('afterend', "<button id='myButton'
onClick='lookup();return false;'>Lookup</button>");
   }
});

function lookup() {
   const zipcode = document.getElementById('zipcode').value;
   const url = 'https://api.zipaddress.net/?lang=rome&zipcode='+zipcode;
```

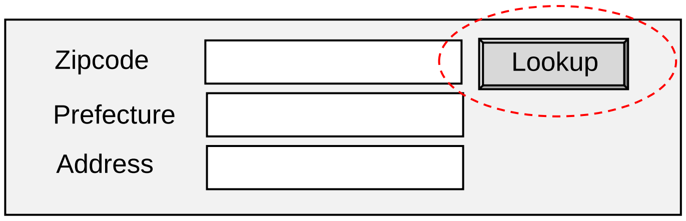
```
    WebExtension.sendHttpRequest('get', url)
      .then(response => handleResponse(response));
}

function handleResponse(response) {
    const address = response.data;
    document.getElementById('pref').value = address.pref;
    document.getElementById('address').value = address.address;
}
```

There are three parts in the above code:

- Add an extension register
- Call REST API (lookup address)
- Handle the response

Step 1: Add an extension register

What you need to do first is to register an extension register function by calling WebExtension.addExtensionRegister(). In the function, you need to check to see if your custom HTML element has not added yet. If not yet, add the custom element on the page. In this example, we want to add a button right next to zipcode text field, in order to invoke the REST API. The custom button (id=myButton) will be inserted when zipcode exists and myButton does not exist yet. And the myButton has onClick attribute with a JavaScript to call lookup function.
With this extension register, user will see the custom button "Lookup" when the page is loaded.



Step 2: Call REST API

When user clicks the Lookup button, lookup function is fired. To make a REST API call, WebExtension provides the following function:

```
WebExtension.sendHttpRequest('get', url).then(response =>
handleResponse(response));
```

The lookup function gets the current value from the zipcode field, and generates a URL with REST API URL and zipcode parameter. And then it calls the sendHttpRequest function. When the HTTP request has been called successfully, then handleResponse function is invoked with the response from the REST API.

Step 3: Handle the response

Finally, the handleResponse function is the function to process the response from the REST API. It extracts address data from the response and sets prefecture and address into each input field on the HTML page.

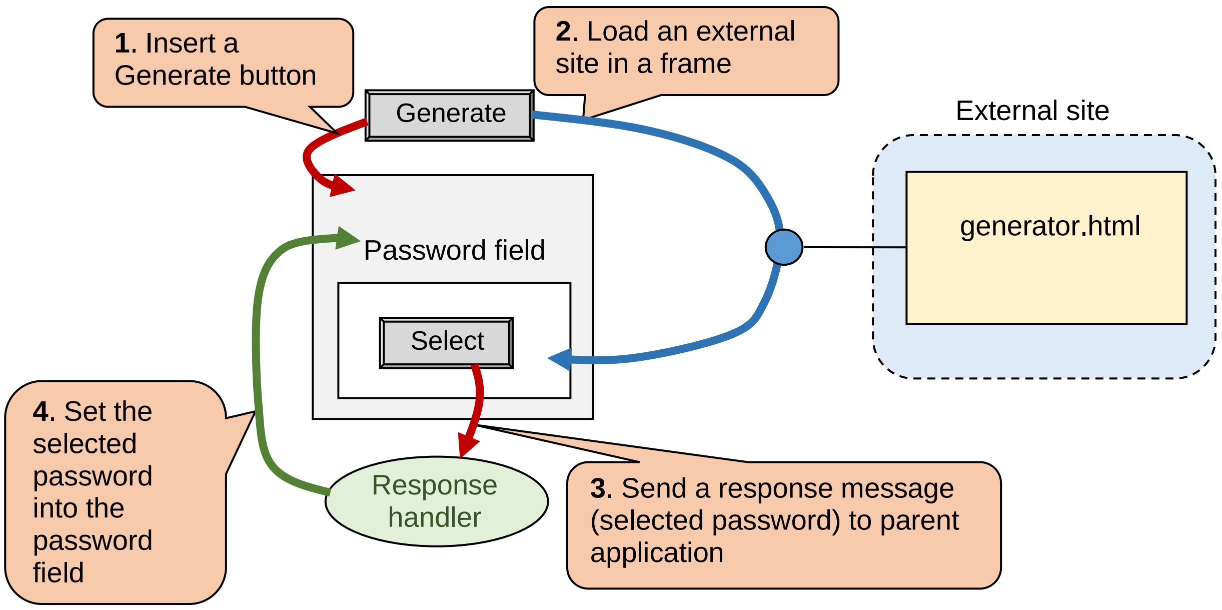
4 Extension with Custom Page

This chapter shows another example of the case that an application to be called has a screen. We assume that you have another web application to handle a specific job, and you want to integrate your base application with the web app so that user can do the specific job on your base application.

Sample Site:
https://web-extension.github.io/FrameClient/index.html

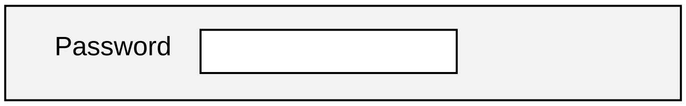
Code:
https://github.com/web-extension/master/tree/master/samples/FrameClient

This sample site has an user profile from (Password field only).
Step 1: WebExtension dynamically adds a Generate button in the form when the page is loaded.
Step 2: When user clicks on the Generate button, WebExtension opens a new frame on the same page and loads the external web page (generator.html). There are three suggested passwords with select button on the page.
Step 3: When user selects a password, WebExtension sends a response message (selected password text) to the parent application.
Step 4: When the parent application received the response, response hander function is fired and set the new password into the password field.


1. Insert a Generate button
2. Load an external site in a frame
Generate
External site
generator.html
Password field
Select
4. Set the selected password into the password field
Response handler
3. Send a response message (selected password) to parent application

4.1 Profile Form and Password Generator

Suppose your application has an user profile form as shown below:

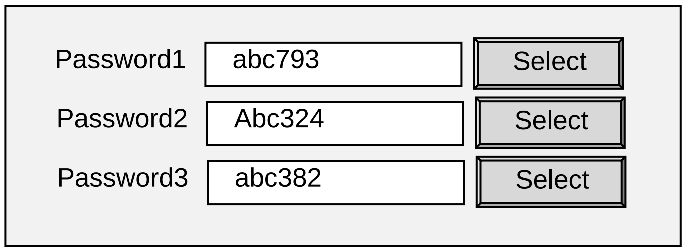


To simplify the example, there is a password field only on that page.
The HTML code of the page is below:

```
<html>
<body>
Password <input id="password" type="text" value="">
</body>
</html>
```

And your team also has another web application which generates three sample passwords so that user does not have to think of new password phrase by themselves.

generator.html



Password1	abc793	Select
Password2	Abc324	Select
Password3	abc382	Select

Now we want to add a new custom button next to the password field on the user profile page, and open the generator HTML page in a new frame. Once user selects one of Select buttons, close the frame and the selected password will be set in the password field on the user profile page.

4.2 Extend the User Profile Form

As you saw the previous example, WebExtension does not require to change the body HTML code. You just need to add <script> tags in the HTML.

```
<html>
<head>
  <script src="WebExtension.js"></script>
  <script src="PasswordGenerator.js"></script>
</head>
<body>
Password <input id="password" type="text" value="">
</body>
</html>
```

Your integration code can be described in the PasswordGenerator.js. Below is a sample implementation to open another application in a frame and get a response from the app.

PasswordGenerator.js

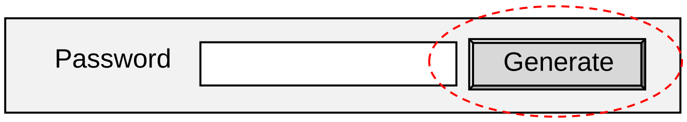
```
WebExtension.addExtensionRegister(() => {
  const target = document.getElementById('password');
  if (!target) return;
  const button = document.getElementById('myButton');
  if (!button) {
    target.insertAdjacentHTML('afterend', "<button id='myButton'
onClick='generate();return false;'>Generate</button>");
  }
});

function generate() {
  const password = document.getElementById('password').value;
  const button = document.getElementById('myButton');
  const url = 'generator.html?'+encodeURIComponent(password);
  WebExtension.openFrame(url, button);
}

WebExtension.addResponseHandler('passwordGenerator', response => {
  document.getElementById('password').value = response;
});
```

There are three parts in the above code:

- Add an extension register
- Open a new frame
- Add a response handler

Step 1: Add an extension register

We here also need to register an extension register function as the first step. As you saw in AddressLookup example, what to do here are same. First, call the WebExtension.addExtensionRegister(), and check to see if your custom HTML element has not added yet. If not yet, add the custom element on the page.
In this example, we want to add a button right next to password text field, in order to invoke a custom event handler (generate function). The custom button (id=myButton) will be inserted when password field exists and myButton does not exist yet. And the myButton has onClick attribute with a JavaScript to call the generate function.
With this extension register, user will see the custom button “Generate” when the page is loaded.



Step 2: Open a new frame

When user clicks the Generate button, generate function is fired. To open a new frame, WebExtension provides the following function:

```
WebExtension.openFrame(url, button);
```

The generate function gets the current value from the password field, and generates a URL to generator.html page with the current password value in its parameter. And then it calls the openFrame function. A new frame will be opened next to the Generate button.

Step 3: Add a response handler

In this sample, the response is invoked from the generator.html which is the client application page. In order to communicate with base application from the HTML, the generator html includes WebExtensionClient.js.

generator.html

```
<html>
<head>
<script src="WebExtensionClient.js"></script>
<script>
```

```
window.onload = function() {
  const pass = (window.location.search+'aaaaaaa').substring(1,7);
  for (let i=1; i<=3; i++) {
    document.getElementById('password'+i).value = pass + (Math.floor(Math.random() *
900) + 100);
  }
};

function selectNewPassword(number) {
  const newPassword = document.getElementById('password'+number).value;
  WebExtension.sendResponse('passwordGenerator', newPassword);
}
</script>
</head>
<body>

<h3>Suggested new passwords.</h3>

Password1 <input id="password1" type="text" value=""><button
onClick='selectNewPassword(1)'>Select</button><br><br>
Password2 <input id="password2" type="text" value=""><button
onClick='selectNewPassword(2)'>Select</button><br><br>
Password3 <input id="password3" type="text" value=""><button
onClick='selectNewPassword(3)'>Select</button><br><br>

<p>
  This page generates three recommended passwords based on the original
password.<br>
  Please choose one of the passwords.
</p>

</body>
</html>
```

And when the user clicked Select button, the page calls the WebExtension.sendResponse function with the selected password as its response message. The function sends a message to the parent window. And the base application invokes registered responseHandler function with the specified responseId. The function is added according to the code below:

```
WebExtension.addResponseHandler('passwordGenerator', response => {
  document.getElementById('password').value = response;
});
```

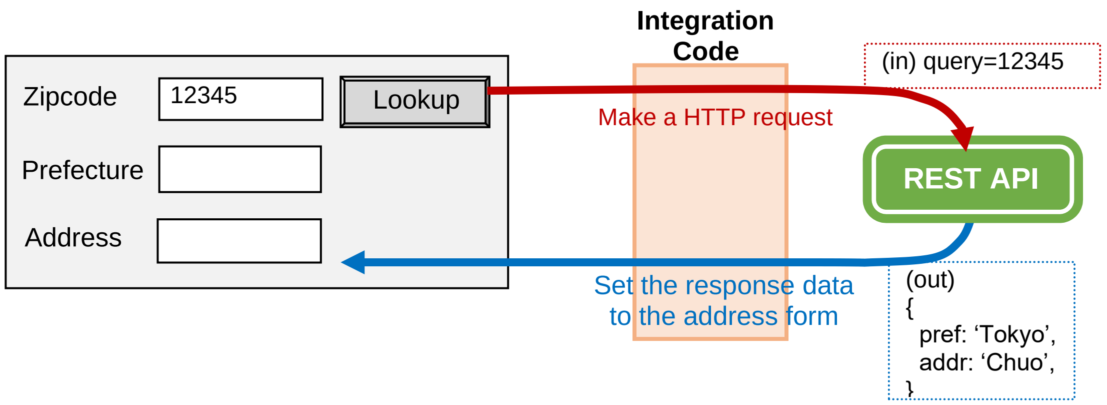
So, when WebExtension gets the response for passwordGenerator, it invokes the above function with the given new password. And the new password is populated into the password input field.

5 Extension Patterns

This chapter shows extension patterns which WebExtension can handle to integrate.

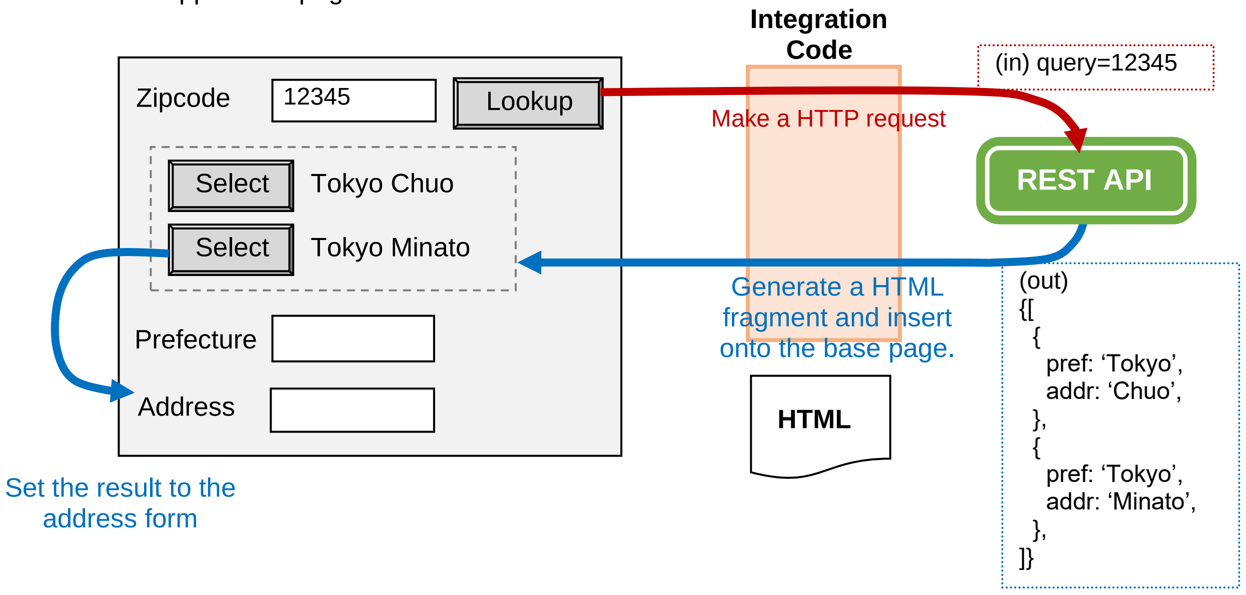
5.1 No UI

This is the simplest pattern which does not require user interface. Integration code makes a REST API call and populates the response data into the base application.



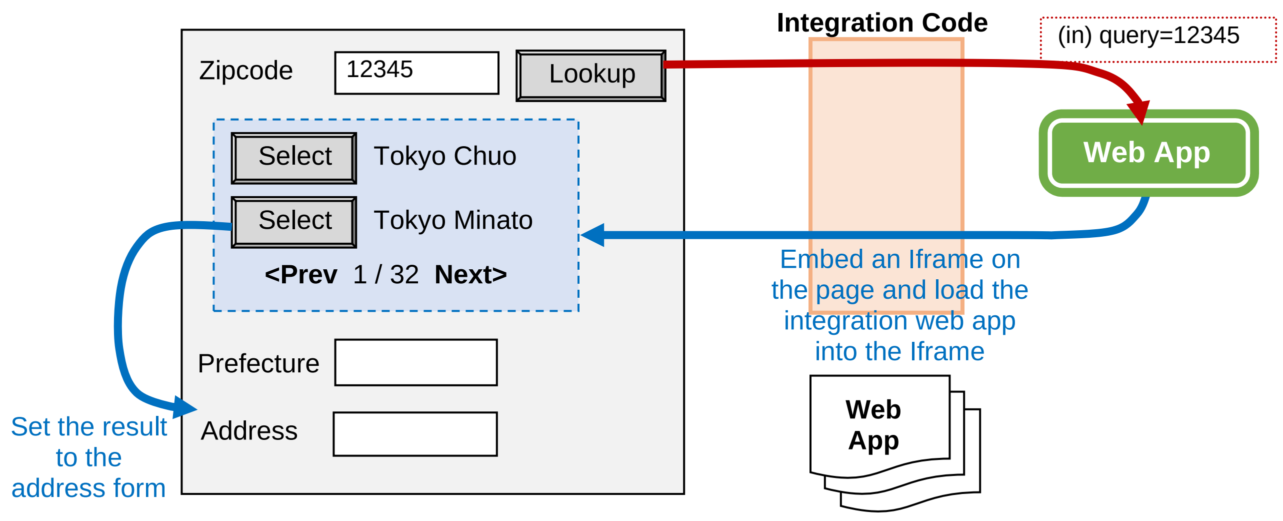
5.2 HTML Fragment

This pattern requires a HTML fragment to be embedded onto the base application. As we saw in No UI pattern, integration code makes a REST API call, but this time, it may return multiple results, and we do not know which result is the one that user wants to choose. Hence, the application needs to display a list of the results so that user can choose. To do so, the integration code generates a list of the result in HTML format and inserts onto the base application page.



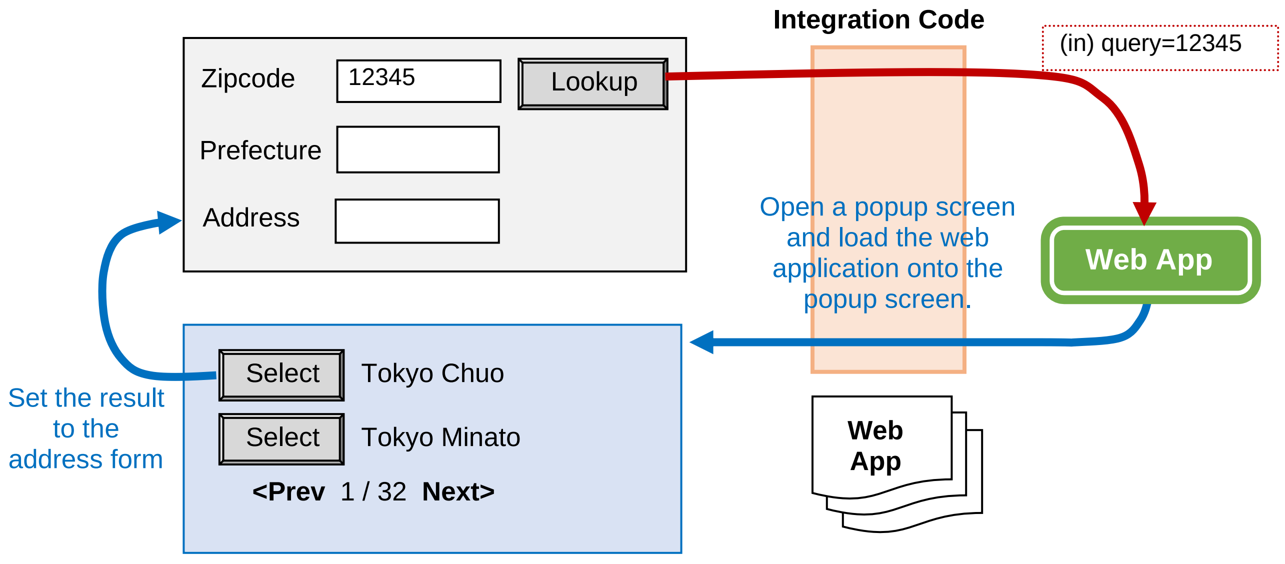
5.3 Small IFrame

If the integration needs more interactions with user (the content is more complicated than the HTML fragment), you may want to embed the web application on the page. For example, if REST API returns many results which requires to be shown in a data table with pagination, it is easier to implement it as a web application and embed the content on the base page in an Iframe tag. This is the pattern that inserts an Iframe tag on the base page and load the external web application into the Iframe. Once user selected final data, the Iframe will be closed and the selected data will be sent to the base page.



5.4 Popup IFrame

This pattern is technically same as Small Iframe. A difference is the size of the Iframe. This pattern is for web application which requires bigger screen space. This pattern opens the Iframe with full of the frame size or browser size which covers the base page. Once user selects a final result, the popped-up Iframe will be closed and the original base page is displayed again with the selected data.

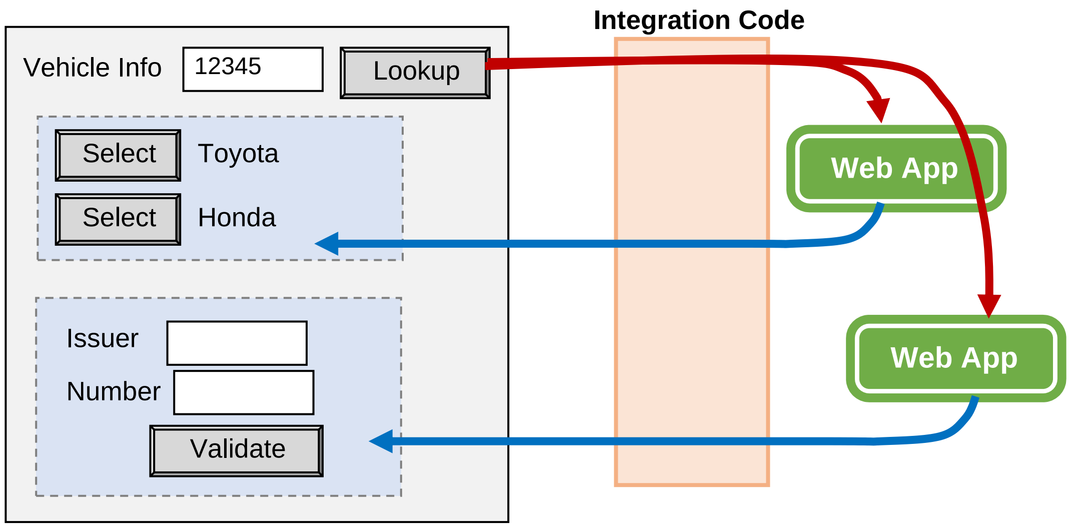


5.5 Combination Patterns

The above patterns show basic flows to integrate. But they may not be enough for complicated scenarios. We need to combine patterns to implement such a complicated flow.

No UI + HTML Fragment
For example, Address Lookup REST API returns multiple addresses according to the given zipcode. If the zipcode is associated with only one address, the integration can simply populate the found address on the base page. But if the zipcode is associated with multiple addresses, we will get multiple results from the API. We cannot determine which address is the one to be chosen. Thus, we have no choice to display the addresses returned from API on the page to ask user which address should be populated.
That is, if only one address is found, it can be implemented with No UI pattern. But if multiple results are returned, it has to be handled with HTML Fragment pattern.

Multiple IFrames
Another example is to use multiple small Iframes. If an integration needs to work with two web applications, two Iframe elements need to be inserted on a same page and each Iframe must be able to communicate with parent application.

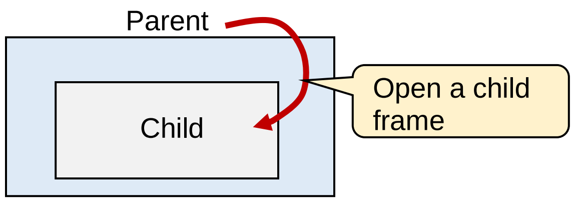


6 API Reference – WebExtension.js

This chapter lists WebExtension JavaScript APIs.

6.1 openFrame

Opens a new WebExtension frame at the extension point and load a child page in the frame. If you do not specify the extension point, the frame will be opened with the browser size.



```
WebExtension.openFrame ( src , extensionPoint , options );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
src	URL	URL to be loaded in the frame	Yes
extensionPoint	HTML element	HTML element object where the frame is inserted. By default, the frame is added right after the tag.	No
options	Object	Specifies option parameters. **frameId** Specifies a unique identifier of a new frame to be opened. You do not have to specify the ID if you use only one frame on a page. **width** Specifies the width of the frame. **height** Specifies the height of the frame. **style** Specifies CSS styles to be applied on the frame. For example, style: 'max-height: max-content' **position** Specifies a position where the frame is inserted. Choose one of these: **beforebegin, afterbegin, beforeend** or **afterend**. **frameSize** Specifies a size of the frame. Choose one of these: **full** : full size of the section,	No

		browser : full size of the browser, **'' (empty)** : content size **proxy** A URL of proxy servlet. By default, '/WebExtensionHttpProxy' handles all requests. **useProxy** Specifies true, if you want to load the src page through http proxy so that you can access the IFRAME document. **onload** Specifies a function to b fired when the src page is loaded in the frame.	

Example 1: Open the web page given by the source parameter with browser screen size.

```
WebExtension.openFrame('http://myapp/landingPage');
```

Example 2: Open the web page right after the HTML element given by extensionPoint parameter.

```
WebExtension.openFrame(
        'http://myapp/landingPage',
        document.getElementById('keyword')
);
```

Example 3: Insert the frame inside the extensionPoint tag.

```
WebExtension.openFrame(
        'http://myapp/landingPage',
        document.getElementById('main-section'),
        { position: afterbegin }
);
```

6.2 closeFrame

Closes the WebExtension frame.

```
WebExtension.closeFrame ( frameId );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
frameId	String	Specifies the frameId given to openFrame API. If it is not specified when opened, the frameId can be omitted.	No

By default, the frame is automatically closed when QuckConnect received a response from QuckConnect frame. So, you won't call the API in normal cases.
When you set closeFrame option to false and send back your response with sendResponse API, you need to close the frame with this API.

6.3 getFrameElement

Returns a HTML element object of WebExtension frame.

```
const frameTag = WebExtension.getFrameElement ( frameId );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
frameId	String	Specifies the frameId given to openFrame API. If it is not specified when opened, the frameId can be omitted.	No

6.4 isFrameOpen

Returns true if the WebExtension is opened. Otherwise, return false.

```
const boolVal = WebExtension.isFrameOpen ( frameId );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
frameId	String	Specifies the frameId given to openFrame API. If it is not specified when opened, the frameId can be omitted.	No

6.5 sendHttpRequest

Sends a HTTP request to the specified URL, and returns a Promise object to handle response from the server.

```
WebExtension.sendHttpRequest (
        method , url , options
);
```

Parameters: This API accepts the following parameters:

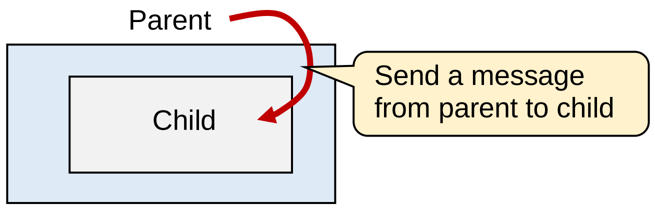
Name	Value Type	Description	Required
method	String	Specifies a HTTP method. **GET** or **POST**.	Yes
url	String	Specifies a URL to send the request to.	Yes
options	Object	Specifies option parameters. **body** Specifies body text or JSON object to be sent when POST method. **proxy** A URL of proxy servlet. By default, '/WebExtensionHttpProxy' handles all requests. **useProxy** Specifies false, if you send a request directly without through the proxy. **responseType** Data type in response. Choose **text** (default) or **json**. **requestHandler** A function to be called before sending a request. It will get a request parameter to receive the request object. Normally, this is defined to set request headers.	No

Example 1: Call a REST API and handle the response JSON object in resolve function.

```
function lookup() {
  const zip = document.getElementById('zipcode').value;
  const url = 'https://myapp/?zip='+zipcode;
  WebExtension.sendHttpRequest('get', url).then(
    response => handleSuccess(response),
    response => handleFailure(response),
  );
}

function handleSuccess(response) {
  // process the response
}

function handleFailure(response) {
  // process the response
}
```

6.6 sendMessage

Sends a message to the child application loaded in a frame.



```
WebExtension.sendMessage ( messageId , message , options );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
messageId	String	Specifies a unique identifier specified by addMessageHandler. When you call this function, the message handler with the same messageId is invoked to process the sent message in the child application.	Yes
message	String / JSON	Specifies a text string or JSON object to be sent to child application.	Yes
options	Object	Specifies option parameters. **frameId** Specifies the target frame ID where the message will be sent. If the ID is a default ID (webExtensionFrame), this prop can be omitted.	No

Example 1: Send a JSON object holding a list of addresses to a client app.
Note that the child frame must be opened when this API is called. So, you need to call the API in **onload** event of the frame element when you open the frame.

```
WebExtension.openFrame(url, button, {
    onload: function() {
        WebExtension.sendMessage('addressList', list);
    }
});
```

6.7 sendResponse

Sends a response data to parent application from parent application.

```
WebExtension.sendResponse ( id , response , options );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
id	String	Specifies an identifier of the response. When WebExtension receives the response, the response handler function registered with the id is fired.	Yes
response	String or JSON	Specifies data to be sent	Yes
options	Object	Specifies option parameters. **closeFrame** **true** or **false** Specifies false if you keep the frame open even after the response was sent. **frameId** Specifies the **frameId** specified in options of **openFrame** API. If you use default ID, you do not specify this property.	No

Example 1: Send a value in WebExtension frame to WebExtension.

```
function selectNewPassword(number) {
  const frame = WebExtension.getFrameElement();
  const newPassword =
frame.contentWindow.document.getElementById('password'+number).value;
  WebExtension.sendResponse('passwordGenerator', newPassword);
}
```

6.8 setExtensionRegistersTimer

Sets a timer to invoke extension register functions.

```
WebExtension.setExtensionRegistersTimer ();
```

The timer is automatically set by the WebExtension, hence, you do not have to call the function by yourself. However, you may need to call it when the timer does not work for some reason.

6.9 addExtensionRegister

Adds an extension register function that is for adding or updating HTML elements on existing pages. **The added function is invoked by WebExtension every second, so it must be designed not to take longer time to execute.**

```
WebExtension.addExtensionRegister ( func );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
func	Function	Specifies a function to add your custom HTML code on existing page.	Yes

Example 1: Register a function that adds a BUTTON html element before P tag.

```
WebExtension.addExtensionRegister(() => {
  const target = document.getElementsByTagName('p')[0];
  if (!target) return;
  const button = document.getElementById('myButton');
  if (!button) {
    target.insertAdjacentHTML('beforebegin', "<button id='myButton' onClick=\"alert('Hello,
World!');return false;\">Say Hello</button>");
  }
});
```

6.10 addResponseHander

Adds a response handler function that handles a text or JSON object returned from client page.

```
WebExtension.addResponseHandler ( responseId , func );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
responseId	String	Specifies an identifier of the response message. The id must be matched with the id specified by sendResponse function.	Yes
func	Function	Specifies a function to add your custom HTML code on existing page.	Yes

Example 1: Register a function that sets a new password into a password input field.

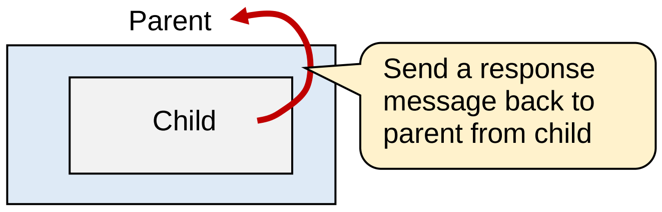
```
WebExtension.addResponseHandler('passwordGenerator', response => {
  document.getElementById('password').value = response;
});
```

7 API Reference – WebExtensionClient.js

This chapter lists WebExtensionClient JavaScript APIs.
The JavaScript file is placed in target application to be displayed in a frame opened by base application.

7.1 sendResponse

Sends a response message to the parent application, which opened the frame.



```
WebExtensionClient.sendResponse ( responseId , response , options );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
responseId	String	Specifies a unique identifier specified by addResponseHandler. When you call this function, the response handler with the same responseId is invoked to process the response message on the parent application.	Yes
response	String / JSON	Specifies a text string or JSON object to be sent to base application.	Yes
options	Object	Specifies option parameters. **frameId** Specifies a frame ID to be closed. If the ID is same as the responseId, this prop can be omitted. **closeFrame** Specifies true, if you want to close the frame after the response was processed.	No

Example 1: Send a selected name as a text to the base application.

```
WebExtension.sendResponse ( ‘myApp’, selectedName );
```

Example 2: Send response data as a JSON object, and the frame will be kept open even after the response was handled on the base app.

```
          <Button
            id="confirmBtn"
            onClick={() => WebExtension.sendResponse('bank', {
               bankName: bank.name,
                    bankCode: bank.code,
                    branchName: branch.name,
                    branchCode: branch.code,
                }, {
                    closeFrame: false
                })}
            >
```

7.2 addMessageHander

Adds a message handler function that handles a text or JSON object sent from the parent application.

```
WebExtensionClient.addMessageHandler ( messageId , func );
```

Parameters: This API accepts the following parameters:

Name	Value Type	Description	Required
messageId	String	Specifies an identifier of the sent message. The id must be matched with the id specified by sendMessage function.	Yes
func	Function	Specifies a function to handle the message on your page.	Yes

Example 1: Register a function that sets the list of addresses sent from the parent application.

```
WebExtension.addMessageHandler('addressList', message => {
  for (let i=0; i<message.length; i++) {
    addAddress(i, message[i]);
  }
});
```