Web Application Extension Framework

WebExtension

User Guide

Revision History

Version	Date	Author	Description
0.0.1	2/3/2020	Masayuki Otoshi	Document Created
0.0.2	7/1/2020	Masayuki Otoshi	Added ResponseFilter

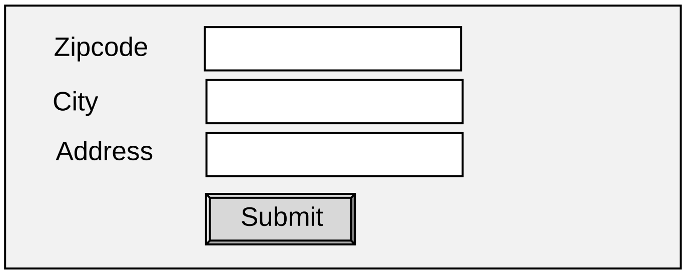
Table of Contents

1 Overview

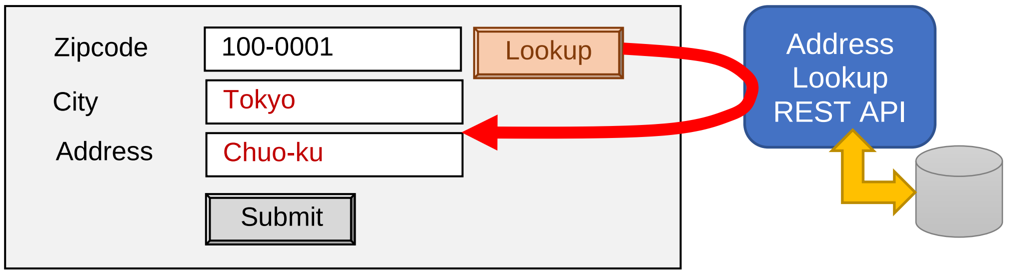
This document is for system integrators and developers who are starting to enhance existing web applications by using microservices such as REST APIs, frontend applications.

1.1 What is WebExtension?

WebExtension is a lightweight framework to customize the browsing web pages. It enables to add an additional feature on an existing page of web application. Suppose your web application has an address form and you want to add a feature to assist user to populate their address, for example, by zipcode, by scanning driver license card, etc. WebExtension enables the feature to be implemented separately from the application.
You do not have to change any code of the existing application. The address form can be kept as a simple form as shown below:



WebExtension allows you to update the page when the page was generated on server side. Using the feature, a button can be added right next to Zipcode input field before sending back to browser. An onClik event handler is attached on the Lookup button so that an Address Lookup REST API is called when user clicked.

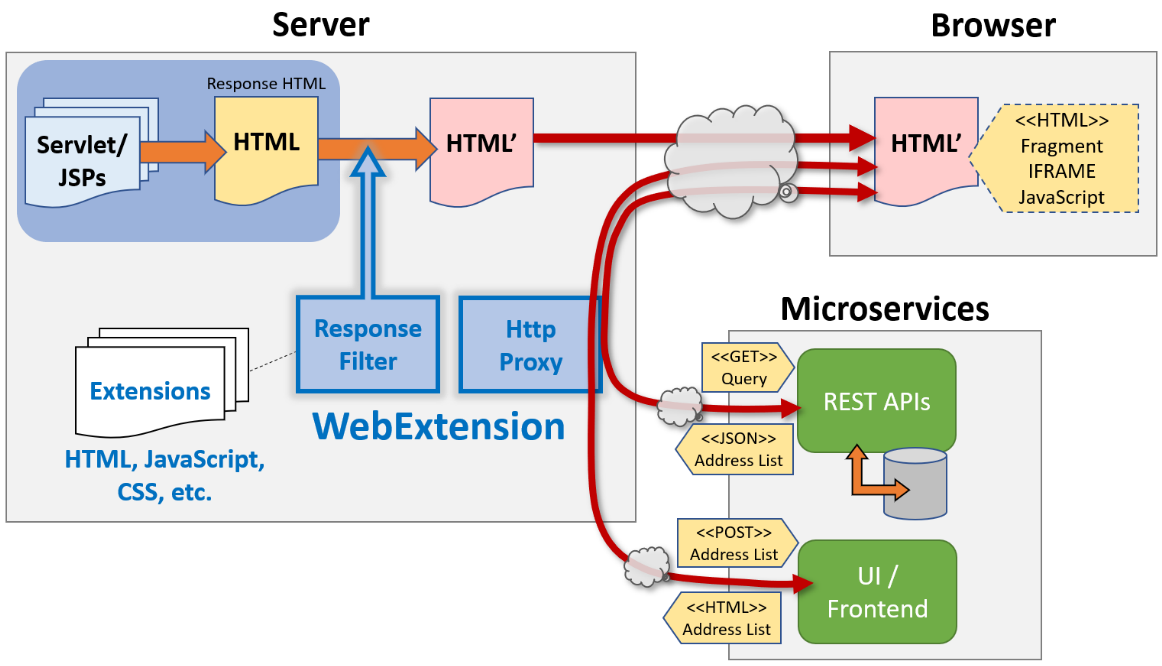


The REST API searches an address by the zipcode entered in the input field and returns an address found. And the address is set on the Address form by the handler function.
As you saw, the following three components need to be developed to implement the Address Lookup feature:

- Lookup button
- Event handler function
- Address Lookup REST API

They can be implemented separately from the existing application and we can dynamically add the feature in runtime by using WebExtension. I need to emphasize that we do not have to change any source code of the existing application to add the feature.

Why can we modify original HTML page with touching the source code? Response Filter component of WebExtension works as a web filter on the web server, and it inserts extension code into the response HTML before sending back to browser. You can add/remove any HTML elements such as buttons, links, JavaScript code and CSS. WebExtension supports XML, JSON and String data as well as HTML format.

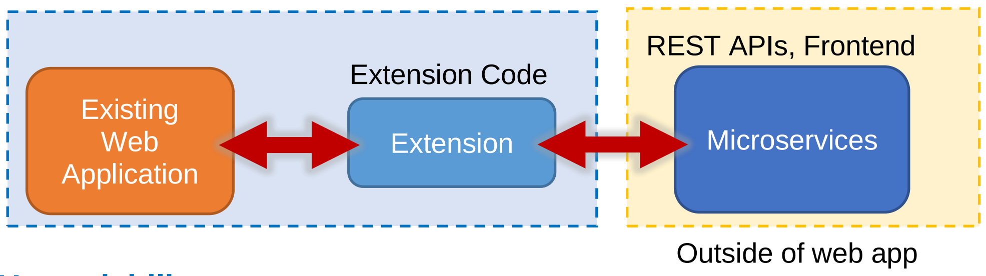


1.2 Why is WebExtension needed?

WebExtension will bring you mainly the following two benefits:

Reusability

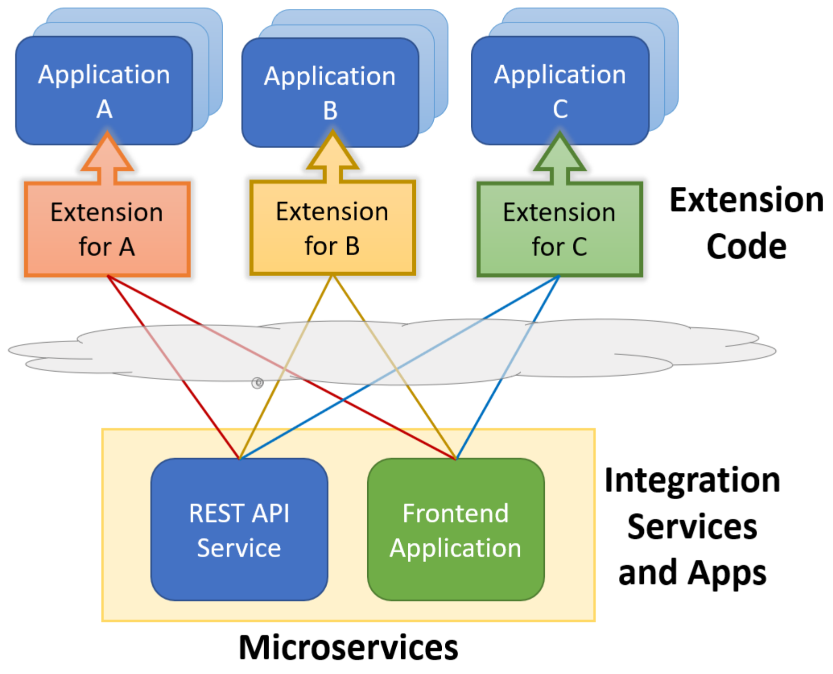
Additional functions, for example, entering an address by Address Book, Zipcode, Driver's license card, are **implemented outside** of the existing application, thus the functions can be reused among applications.



Upgradability

The extension code can be applied on the existing application **without any code changes** so that the existing application can freely upgrade as long as they keep interface.
With doing so, the extension code and the existing application are loosely coupled, and each project can proceed their development separately. Also, by developing piece by piece, each application size can be kept smaller. It also results that our development is easy to handle.

Microservices (REST APIs, Frontend apps) aim to be called from many web applications and the data and page contents are commonly used among them. Page structure, element names, CSS styles, etc depend on web apps, but the differences are absorbed by Extension Code so that we can use unified names and styles in our microservices.



Even if upgrade of Application A breaks interface, it only affects Extension A. Microservices will not be needed to change their code. Also, the extension code should be very small, so we can easily detect what to fix.

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/
- **Apache Tomcat Version 9.0.30** or above (or equivalent servlet container)
  http://tomcat.apache.org/
- **Chrome browser**
  https://www.google.com/chrome/

Below are dependency jar files:

- **Jakarta JSON Processing API Version 1.1.6** or above
  (or equivalent JSONP implementation)
  https://eclipse-ee4j.github.io/jsonp/
- **jsoup Version 1.13.1** or above
  https://jsoup.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/
└──── README.md
```

Copy the **samples** folder into **webapps** folder of Tomcat.

```
C:> copy  samples  <TOMCAT-root>/webapps
```

2.3 Setup web.xml

This is the last step of installation. There are two entries you need to configure.

ResponseFilter

ResponseFilter is a web filter to update response contents (e.g. HTML, XML, CSS, JavaScript, etc). Sample configurations have been described in sample folder.
Open the existing web.xml in the samples.

```
<TOMCAT-root>/webapps/samples/WEB-INF/web.xml
```

Replace the TOMCATROOT with your <TOMCAT-root> folder path

```
<filter>
  <filter-name>WebextensionResponseFilter</filter-name>
  <filter-class>webextension.ResponseFilter</filter-class>
  <init-param>
    <param-name>srcdir</param-name>
    <param-value>TOMCATROOT\webapps\samples</param-value>
  </init-param>
  <init-param>
    <param-name>classpath</param-name>
    <param-value>
      TOMCATROOT\webapps\samples\WEB-INF\lib
    </param-value>
  </init-param>
  <init-param>
  <param-name>updater</param-name>
    <param-value>
      HelloWorld.HelloWorldUpdater.javax
      AddressLookup.AddressLookupUpdater.java
      AddressList.AddressListUpdater.java
      AddressListFrame.AddressListFrameUpdater.java
      PasswordGenerator.PasswordGeneratorUpdater.java
      MultiFrames.MultiFramesUpdater.java
      POST.PostUpdater.java
      SPA.SpaUpdater.javax
      Updater.HtmlUpdater.javax
      Updater.XmlUpdater.javax
      Updater.JsonUpdater.javax
    </param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>WebextensionResponseFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

HttpProxy

If **metadata-complete** is set to **false** in your web.xml, you can skip this setting.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">
```

If it is set to true, you must add the following entries:

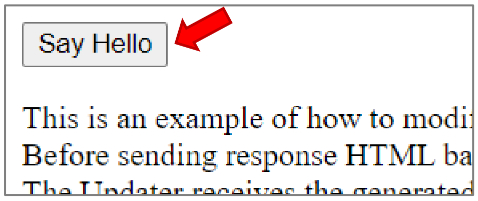
```
<servlet>
    <servlet-name>WebextensionHttpProxy</servlet-name>
    <servlet-class>webextension.HttpProxy</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>WebextensionHttpProxy</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. Start your Tomcat.
2. Open http://localhost:8080/samples/HelloWorld/index.html with Chrome browser.

You will see a "**Say Hello**" button on the shown page.



This is an example of how to modi
Before sending response HTML ba

3 Getting Started – Hello World

This chapter explains the easiest sample, Hello World, which shows how to insert a HTML button on existing application.

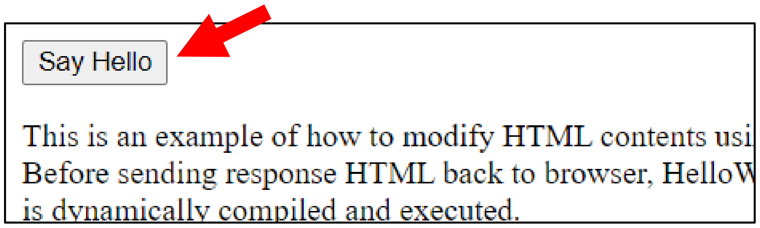
https://github.com/web-extension/master/tree/master/samples/HelloWorld

3.1 Original HTML code

Hello World is a single HTML file that is **index.html** as shown below:

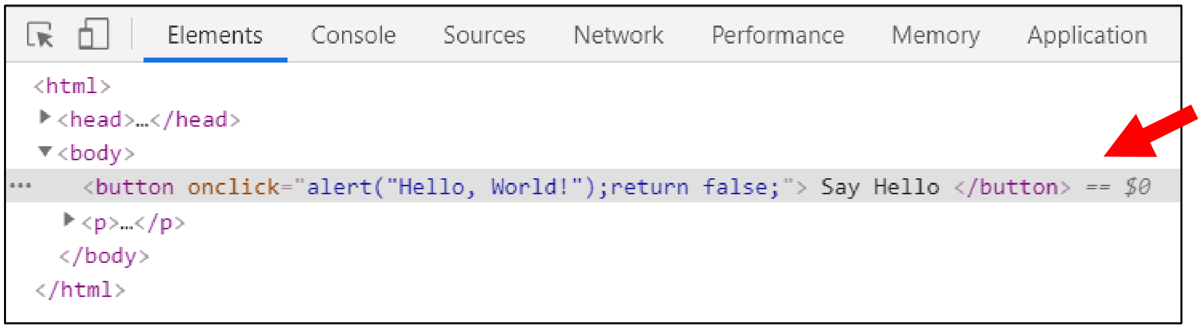
```
<html>
<head>
<title>Hello World</title>
</head>
<body>

<p>
  This is an example of how to modify HTML contents using Updater.<br>
  …
</p>

</body>
</html>
```

There is only one <p> tag in the body. However, you will see a "**Say Hello**" button on the message defined by the <p>.



This is an example of how to modify HTML contents usi
Before sending response HTML back to browser, HelloW
is dynamically compiled and executed.

If you inspect the HTML elements by using inspector, you will see that <button> tag is inserted before the <p> tag. There is no such a tag in the index.html. Where does the <button> come from?



```
<html>
 ▶<head>…</head>
 ▼<body>
    <button onclick="alert("Hello, World!");return false;"> Say Hello </button> == $0
   ▶<p>…</p>
  </body>
</html>
```

3.2 Updater Class

The <button> tag was added by Updater class invoked by ResponseFilter of WebExtension framework. The updater gives you a chance to update original HTML content before sending back to browser. Below shows the source code of updater for HelloWorld, which inserts the <button> tag before the <p> tag.

HelloWorldUpdater.javax

```
package HelloWorld;

import java.util.Map;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import webextension.ResponseHtmlUpdater;
import webextension.annotation.WebExtensionFilter;

@WebExtensionFilter(contentType="text/html", servletPath="/HelloWorld/")
public class HelloWorldUpdater extends ResponseHtmlUpdater {

  public boolean update(Map<String, String> context, Document document) throws Exception {
    Element p = document.selectFirst("p");
    if (p == null) return false;

    p.before(
      #<button onClick='alert("Hello, World!");return false;'>
        Say Hello
      </button>
    );
    return true;
  }
}
```

Updater Base Classes

Updater class needs to be extended from one of classes below depending on its ContentType:

Updater Base Class	Description
ResponseStringUpdater	The content is passed as a String
ResponseHtmlUpdater	The content is passed as a Jsoup Document object parsed by HTML parser
ResponseXmlUpdater	The content is passed as a Jsoup Document object parsed by XML parser
ResponseJsonUpdater	The content is passed as a JsonStructure object.

In this sample, the response content (index.html) is described in HTML format, so we here choose the ReponseHtmlUpdater class as a base class of HelloWorldUpdater class.

```
public class HelloWorldUpdater extends ResponseHtmlUpdater {
```

The base class requires to implement the following interface:

```
boolean update(Map<String, String> context, Document document) throws Exception
```

Through the interface, you will get context and document objects. The context holds request and response config information such as ServletPath of the request, ContentType of the response. The document is an instance of Jsoup Document class where HTML content parsed by HTML parser is stored. We here try to update HTML content by modifying the document object through Jsoup APIs. For the details of the APIs, please see https://jsoup.org/

The update method expects to return a boolean value. If you have no updates in the given document object, return false. ResponseFilter won't reconstruct the HTML content. If you made updates, return true, and ResponseFilter regenerate a new document object to reflect your changes.

Adding a <button> tag

What we want to do here is to add a <button> before the <p>, so the first step is to find he <p> tag from the given document object. It can be done by **selectFirst** API:

```
public Element selectFirst(String cssQuery)
```

It finds the first Element that matches the Selector CSS query. To find a <p> tag, we can simply pass the tag name "p" to the API.

```
    Element p = document.selectFirst("p");
    if (p == null) return false;
```

If there is a <p> tag in the document, an Element object will be returned. Otherwise, it returns null. If there is no <p> tag as the extension point, the content is not our target page. In that case, return **false** not to reconstruct the document object in WebExtension framework.
When a <p> tag is found, the <button> can be added by using the code below:

```
    p.before(
        #<button onClick='alert("Hello, World!");return false;'>
            Say Hello
        </button>
    );
    return true;
```

The Element object, p, has '**before**' API, which insert the specified HTML before this element.

```
public Element before(String html)
```

To commit the change, return **true** from this function.

"#<" Expression

The "**#<**" in the before parameter is an extended expression of javax. It can be used to describe a HTML tag as a String object. The <button> tag is pre-processed and replaced with the following String code before its compilation.

```
    p.before(
        "<button onClick='alert(\"Hello, World!\");return false;'>\n"+
"           Say Hello\n"+
"        </button>"
    );
```

@WebExtensionFilter Annotation

Updater class must be annotated by WebExtensionFilter to limit response contents to be applied.

```
@WebExtensionFilter(contentType="text/html", servletPath="/HelloWorld/index.html")
```

ResponseFilter handles all responses in text format from the server. It could be HTML, XML, JSON, CSS, JavaScript, etc. But, when you create an updater class, most of cases, you must know the content type of the target page if the application is available to access. If the contentType is text/html, you can specify the contentType as “text/html”.
Likewise, by using servletPath, the response contents to be handled can be limited.
By doing so, invoking updater against unmatched contentType or servletPath can be prevented, which improves its performance.

3.3 Web.xml

The updater class is not automatically detected by servlet container. It must be registered in ResponseFilter section of web.xml as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">

  <filter>
    <filter-name>WebextensionResponseFilter</filter-name>
    <filter-class>webextension.ResponseFilter</filter-class>
    <init-param>
      <param-name>-s</param-name>
      <param-value>C:\apache-tomcat-9.0.0\webapps\samples</param-value>
    </init-param>
    <init-param>
      <param-name>-classpath</param-name>
      <param-value>
        C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib
      </param-value>
    </init-param>
    <init-param>
      <param-name>updater</param-name>
      <param-value>
        HelloWorld.HelloWorldUpdater.javax
      </param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WebextensionResponseFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

The "C:\apache-tomcat-9.0.0" must be replaced with your TOMCAT root folder.

Updater parameter

The updater init-param is a mandatory parameter. It accepts a list of updater class name (or java/javax file path)

```
   <init-param>
     <param-name>updater</param-name>
     <param-value>
       HelloWorld.HelloWorldUpdater.javax
     </param-value>
   </init-param>
```

Javac parameters

If you specify java or javax file name in the updater init-param, you need to also specify -s and -classpath init-params. (If you specify classes only, this step can be skipped)
When java or javax file is specified as an updater, the source code is dynamically compiled in runtime just like JSP. The -s and -classpath are passed to javac command as its parameters.

```
   <init-param>
     <param-name>-s</param-name>
     <param-value>C:\apache-tomcat-9.0.0\webapps\samples</param-value>
   </init-param>
   <init-param>
     <param-name>-classpath</param-name>
     <param-value>
       C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib
     </param-value>
   </init-param>
```

The -s option is to specify a root folder of the java and javax source code.
The -classpath option is to specify a list of jar files or library folders to be added into classpath. Below shows an example of multiple jar files specified instead of a library folder:

```
   <init-param>
     <param-name>-classpath</param-name>
     <param-value>
       C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib\jakarta.json-1.x.x.jar
       C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib\ jsoup-1.x.x.jar
       C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib\webextension.jar
     </param-value>
   </init-param>
```

Now that all settings for the HelloWorld sample are done, you can open a browser and access the HelloWorld sample web site. The HelloWorldUpdater.java and class files will be dynamically generated in the same folder as its javax. And a "Say Hello" button is shown on the page if it successfully worked.

Switching to use compiled class from java/javax

The dynamic compilation can be used only for debugging purpose. Once you ensured your updater worked fine, you need to create a jar file containing the generated Java class and change the settings in web.xml to load the pre-compiled class. Here are the steps:

Go to <source root> directory where is the value of -s option (in this sample, C:\apache-tomcat-9.0.0\webapps\samples)

Run jar command with the following parameters:

```
C:> jar cvf helloworldupdater.jar HelloWorld/*.class
```

Copy the “helloworldupdater.jar” file generated into lib directory in Tomcat

```
C:> copy  helloworldupdater.jar  C:/apache-tomcat-9.0.0/webapps/samples/WEB-INF/lib
```

Open web.xml file in WEB-INF folder and change updater path names from javax to no extension. Below shows the current updater name (with javax extension)

```
   <init-param>
     <param-name>updater</param-name>
     <param-value>
       HelloWorld.HelloWorldUpdater.javax
     </param-value>
   </init-param>
```

The above value should be changed to below (without extension).

```
   <init-param>
     <param-name>updater</param-name>
     <param-value>
       HelloWorld.HelloWorldUpdater
     </param-value>
   </init-param>
```

Now that the init-params for dynamic compilation (Javac parameters starting with a dash) are not necessary to specify, so you can remove **-s** and **-classpath** init-params from web.xml. Final web.xml for production version looks below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">

  <filter>
    <filter-name>WebextensionResponseFilter</filter-name>
    <filter-class>webextension.ResponseFilter</filter-class>
    <init-param>
      <param-name>updater</param-name>
      <param-value>
        HelloWorld.HelloWorldUpdater
      </param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WebextensionResponseFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

4 Javax Extended Syntax

As you saw in some examples, javax is used as a file extension for an updater Java class. The javax accepts two expressions which Java does not support yet. This chapter explains the expressions supported in javax file.

4.1 Template literals

Template literals are string literals enclosed by backtick (` `) character instead of double quotes. It allows embedded expressions and multi-line strings.
An expression is indicated by a dollar sign and curly braces (${expression}).

```
String var1 = `string text`;

String var2 = `string text line 1
string text line 2
string text line 3`;

String var3 = `string text ${expression} string text`;
```

The above code is compiled into Java code below:

```
String var1 = "string text";

String var2 = "string text line 1\n"+
"string text line 2\n"+
"string text line 3";

String var3 = "string text "+expression+" string text";
```

4.2 XML literals

XML literals are string literals enclosed by a XML tag instead of double quotes. Like the template literals, it allows embedded expressions and multi-line strings.
The tag must be described in XML format following a # sign. HTML single tag like <BR> is not allowed. (<BR/> must be used instead)

```
    String newline = #<br class="myStyle" />;
    p.before(
        #<button onClick='alert("Hello, World!");return false;'>
            <b>Say Hello</b>${newline}
        </button>
    );
    return true;
```

The above code is compiled into Java code below:

```
    String newline = "<br class=\"myStyle\" />";
    p.before(
        "<button onClick='alert(\"Hello, World!\");return false;'>\n"+
"           <b>Say Hello</b>"+newline+"\n"+
"         </button>"
    );
```

5 Extension with REST API (No UI)

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

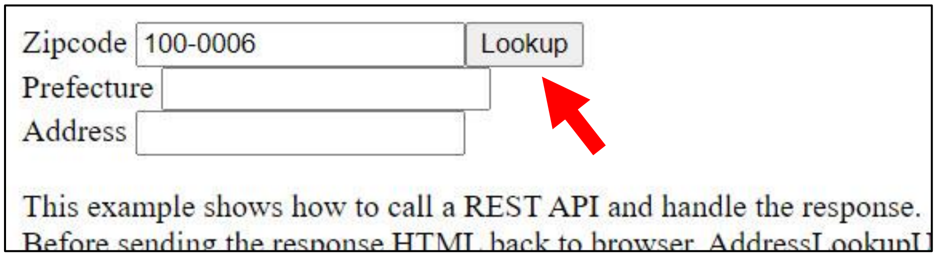
https://github.com/web-extension/master/tree/master/samples/AddressLookup

5.1 Original HTML Code

This sample has only a single HTML page that is **index.html**.

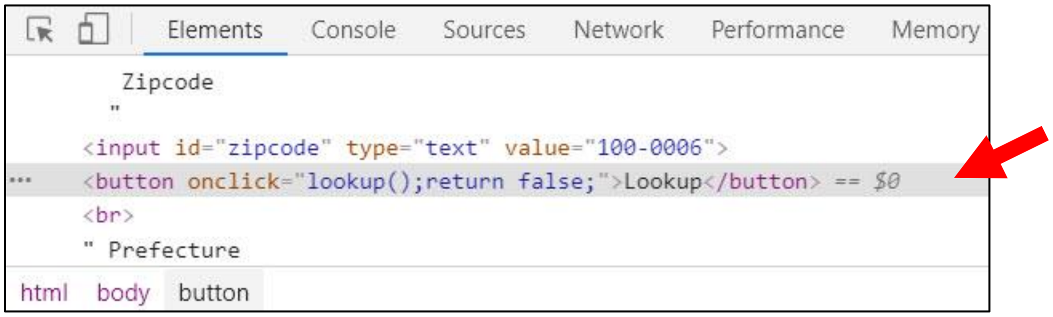
```
<html>
<head>
<title>Address Lookup</title>
</head>
<body>

Zipcode <input id="zipcode" type="text" value="100-0006"><br>
Prefecture <input id="pref" type="text" value=""><br>
Address <input id="address" type="text" value="">

<p>
  This example shows how to call a REST API and handle the response.<br>
  …
</p>

</body>
</html>
```

In the body section, there is an address form containing three fields (Zipcode, Prfecture and Address). Hence, it looks to be expected that user manually enters a value in each field. But, if you open the page, you will see an additional button “Lookup” right next to the Zipcode text field.



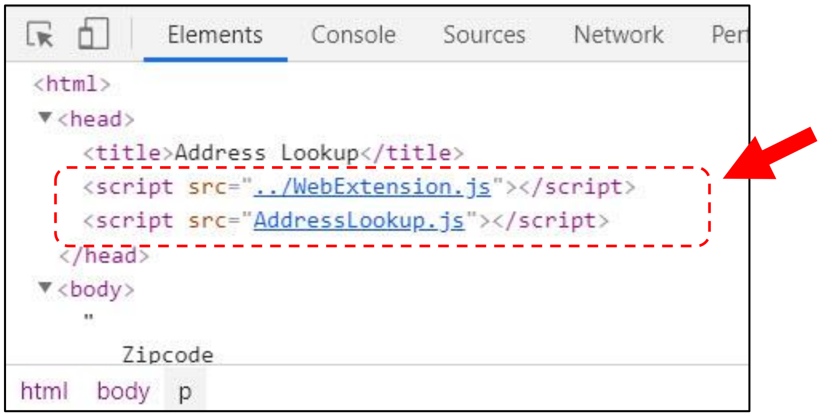
This example shows how to call a REST API and handle the response.
Before sending the response HTML back to browser, AddressLookupU

In fact, a <button> tag was added between Zipcode’s <input> and <br> tags.



```
    Zipcode
   "
  <input id="zipcode" type="text" value="100-0006">
… <button onclick="lookup();return false;">Lookup</button> == $0
  <br>
  " Prefecture
```

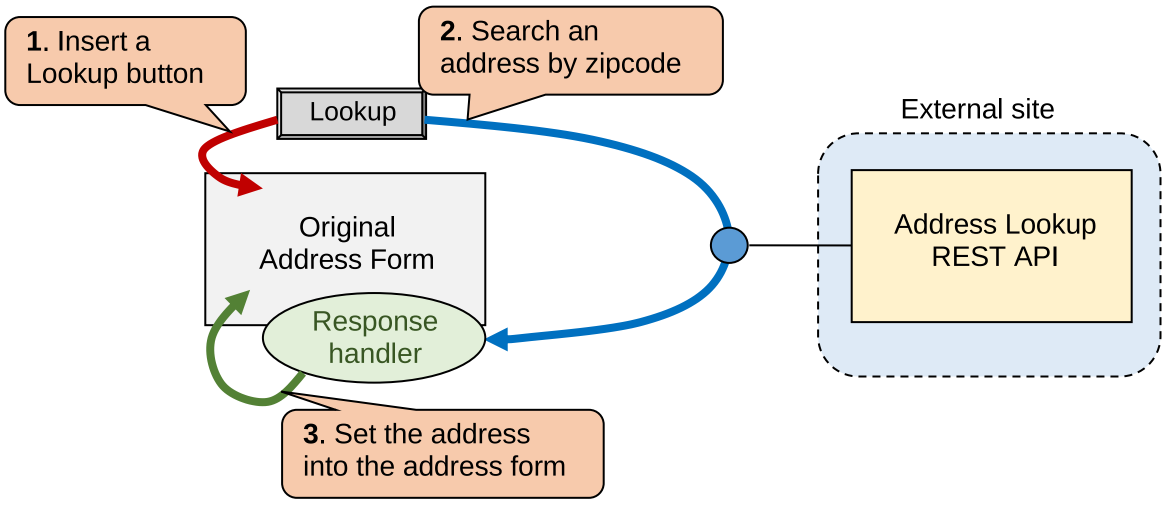
Also if you inspect the <head> tag, you will see additional two <script> tags are inserted.



```
<html>
▼<head>
    <title>Address Lookup</title>
    <script src="../WebExtension.js"></script>
    <script src="AddressLookup.js"></script>
  </head>
▼<body>
    "
       Zipcode
```

Those were inserted by Updater class of WebExtension to add the feature on this page, which searches an address by the zipcode and automatically populates values in the prefecture and address field. It works with the following steps:

Step 1: AddressLookupUpdater dynamically adds a Lookup button in the address form and <script> tags required for the feature.
Step 2: User enters a zipcode and clicks on the Lookup button. It executes a custom JavaScript function that makes a HTTP request against an external site that is AddressLookup REST API. The API returns address information matched with the zipcode in JSON format.
Step 3: When a response is returned, response hander function (a callback function) is fired. And the function populates the address returned from the 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

5.2 Address Lookup REST API

This sample uses Address Lookup REST API provided by 3rd party, zipcloud http://zipcloud.ibsnet.co.jp/doc/api.
The site provides a REST API to search an address from zipcode with the following URL:

```
https://zipcloud.ibsnet.co.jp/api/search?zipcode=<zipcode>
```

The REST API returns a code and an address data found in JSON format. If you pass a zipcode 100-0006, you will get the following JSON:

```
{
        "message": null,
        "results": [
                {
                        "address1": "東京都",
                        "address2": "千代田区",
                        "address3": "有楽町",
                        "kana1": "ﾄｳｷｮｳﾄ",
                        "kana2": "ﾁﾖﾀﾞｸ",
                        "kana3": "ﾕｳﾗｸﾁｮｳ",
                        "prefcode": "13",
                        "zipcode": "1000006"
                }
        ],
        "status": 200
}
```

5.3 Address Lookup Updater

AddressLookupUpdater inserts HTML elements onto the existing index.html page.

AddressLookupUpdater.java

```
package AddressLookup;

import java.util.Map;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import webextension.ResponseHtmlUpdater;
import webextension.annotation.WebExtensionFilter;

@WebExtensionFilter(contentType="text/html", servletPath="/AddressLookup/")
public class AddressLookupUpdater extends ResponseHtmlUpdater {

  public boolean update(Map<String, String> context, Document document) throws
Exception {
    Element head = document.selectFirst("head");
    if (head == null) return false;
    head.append("<script src='../WebExtension.js'></script>")
        .append("<script src='AddressLookup.js'></script>");

    Element input = document.selectFirst("input[id=zipcode]");
    if (input == null) return false;
    input.after("<button onClick='lookup();return false;'>Lookup</button>");

    return true;
  }
}
```

As explained n Hello World sample, Updater must be extended from one of Updater Base classes. This sample also updates HTML content, thus, we will extend from ResponseHtmlUpdaer class.

```
public class AddressLookupUpdater extends ResponseHtmlUpdater {
```

Next, we need to insert two <script> tags in <head> tag.

```
   Element head = document.selectFirst("head");
   if (head == null) return false;
   head.append("<script src='../WebExtension.js'></script>")
      .append("<script src='AddressLookup.js'></script>");
```

The selectFirst API returns the <p> tag object found first. If not found, the HTML content is not our target page, so we should return false not to update the page.
Once we found a <head>, add he the following two JavaScript files.

- **WebExtension.js** – WebExtension JavaScript framework that requires to make a HTTP request to call the AddressLookup REST API.

- **AddressLookup.js** – A custom JavaScript where our custom lookup function is implemented.

Those <script> tags are added by using append API that appends the given HTML elements to the end of the children of the <head>.

```
public Element append(String html)
```

For details on Jsoup API, see API Reference on the website: https://jsoup.org/

After adding the <script> tags, we need to add one more HTML element that is a <button> to invoke lookup JavaScript function. The <button> can be added like the way we did for the <script> tags.

```
   Element input = document.selectFirst("input[id=zipcode]");
   if (input == null) return false;
   input.after("<button onClick='lookup();return false;'>Lookup</button>");

   return true;
```

Get a <input> tag by using selectFirst API, and insert the <button> element after the <input> tag by using after API. The <button> has onClick event to fire lookup() function when clicked.

5.4 Custom JavaScript functions

Next, we need to implement the lookup JavaScript function. It is implemented in AddressLookup.js.

AddressLookup.js

```
function lookup() {
  const zipcode = document.querySelector('input[id=zipcode]').value;
  const url = 'https://zipcloud.ibsnet.co.jp/api/search?zipcode='+zipcode;

  // Make a HTTP request to call an AddressLookup REST API
  WebExtension.sendHttpRequest('get', url)
```

```
    .then(response => setAddress(response.results[0]));
}

function setAddress(address) {
  document.querySelector('input[id=pref]').value = address.address1;
  document.querySelector('input[id=address]').value = address.address2 +
address.address3;
}
```

The lookup function executes two tasks. It calls AddressLookup REST API with a zipcode, and sets the address returned from the API into the existing address form.
To make a HTTP request, sendHttpRequest function of WebExtension API is called.

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

The API makes a HTTP request against the given URL with using the method and returns promise with a response object returned from the server.
In this sample, the response is returned as a JSON object containing an address information. So, by calling setAddress function, the response is set into address from.

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

5.5 Web.xml settings

Finally, you need to register the AddressLookupUpdater to be called from ResponseFilter. If you have added settings for HelloWorld, required init-param should be already added. So, you just need to add AddressLookupUpdater file name in updater parameter.

webapps\samples\WEB-INF\web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">

  <filter>
    <filter-name>WebextensionResponseFilter</filter-name>
    <filter-class>webextension.ResponseFilter</filter-class>
    <init-param>
      <param-name>-s</param-name>
      <param-value>C:\apache-tomcat-9.0.0\webapps\samples</param-value>
    </init-param>
    <init-param>
      <param-name>-classpath</param-name>
      <param-value>
        C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib
      </param-value>
    </init-param>
    <init-param>
      <param-name>updater</param-name>
      <param-value>
        HelloWorld.HelloWorldUpdater.javax
```

```
          AddressLookup.AddressLookupUpdater.java
        </param-value>
      </init-param>
    </filter>
    <filter-mapping>
      <filter-name>WebextensionResponseFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

  </web-app>
```

6 Embedded IFrame

This chapter shows an example of Embedded Iframe pattern that extends existing application using an Iframe element. For example, existing application opens a frame and load another web application in it so that the external app features can be used as a part of the existing application.

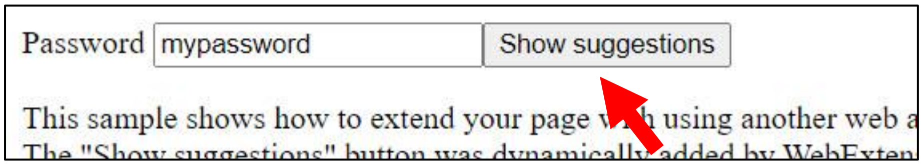
https://github.com/web-extension/master/tree/master/samples/PasswordGenerator

6.1 Original HTML Code

This sample has only a single HTML page that is **index.html**.

```
<html>
<head>
<title>Password Generator</title>
</head>
<body>

Password <input id="password" type="text" value="mypassword">

<p>
  This sample shows how to extend your page with using another web application through
IFrame element.<br>
  …
</p>

</body>
</html>
```

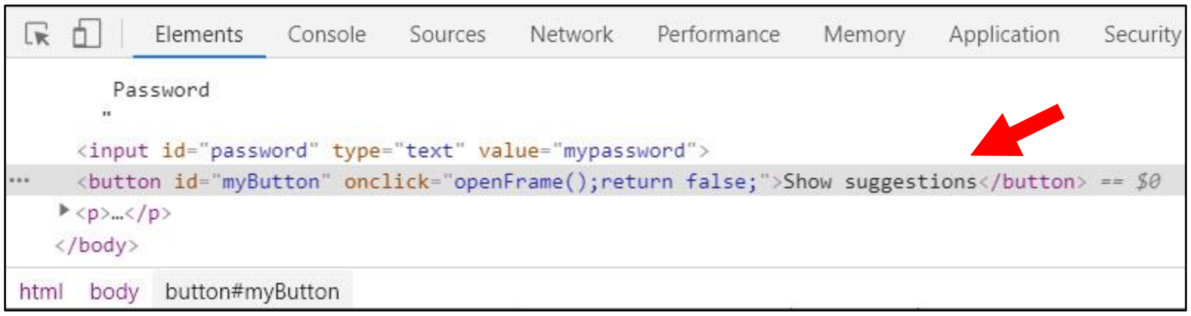
In the body section, there is an input field to enter a password. Hence, it looks to be expected that user manually enters a value in the field.
But, if you open the page, you will see an additional button "Show suggestions" right next to the Password text field.



Password mypassword Show suggestions

This sample shows how to extend your page with using another web a
The "Show suggestions" button was dynamically added by WebExten

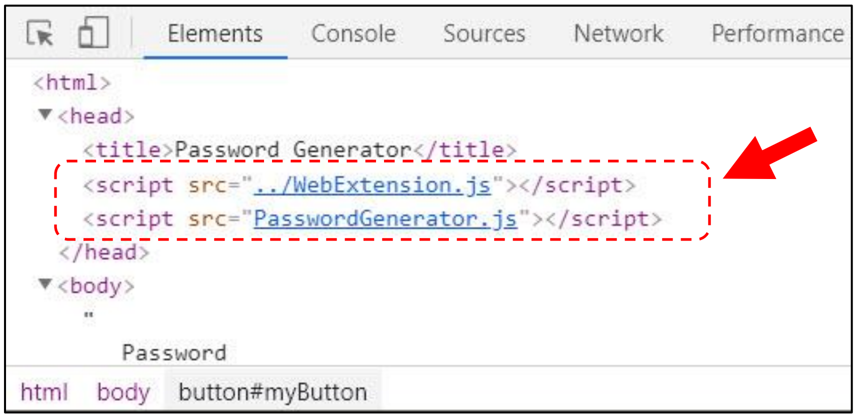
In fact, a <button> tag was added between Password's <input> and <p> tags.



```
    Password
    "
  <input id="password" type="text" value="mypassword">
  <button id="myButton" onclick="openFrame();return false;">Show suggestions</button> == $0
  ▸<p>…</p>
  </body>
```

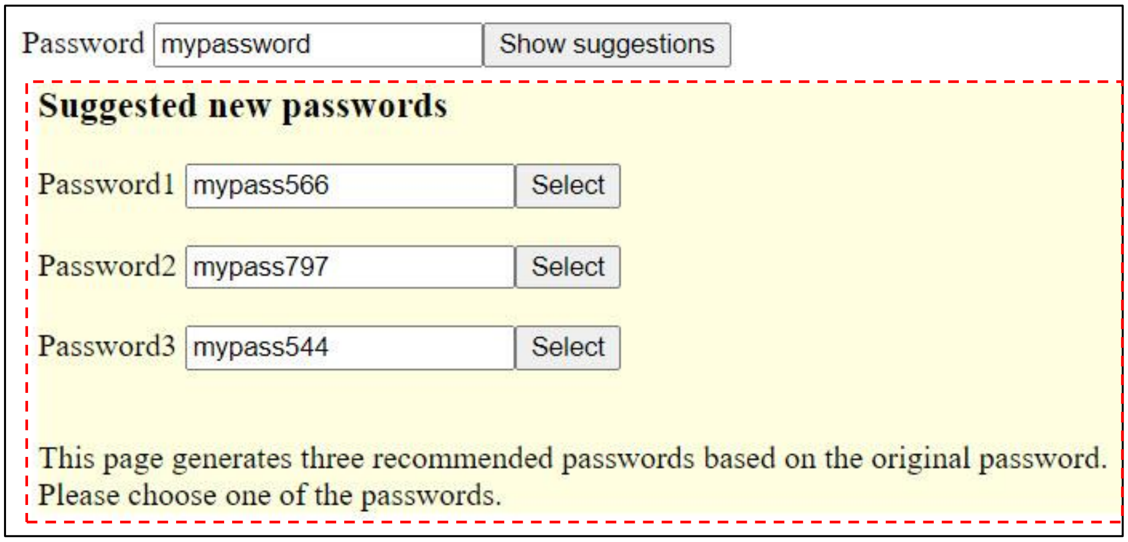
html body button#myButton

Also if you inspect the <head> tag, you will see additional two <script> tags are inserted.



```
<html>
▼<head>
    <title>Password Generator</title>
    <script src="../WebExtension.js"></script>
    <script src="PasswordGenerator.js"></script>
  </head>
▼<body>
    "
      Password
```

Those were inserted by Updater class of WebExtension to add a new feature on this page, which opens an Iframe and loads a web page to show three suggested passwords.
When user clicks on the “Show suggestions” button, it opens a new frame under the button and load another web page (generator.html) that shows three auto-generated passwords.



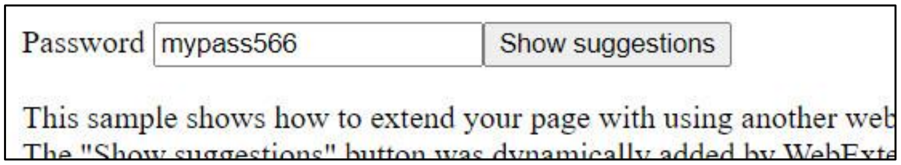
Password: mypassword [Show suggestions]

Suggested new passwords

Password1	mypass566	Select
Password2	mypass797	Select
Password3	mypass544	Select

This page generates three recommended passwords based on the original password.
Please choose one of the passwords.

User clicks on one of the “Select” button, for example, the first Select button, and then, the frame is closed and the selected password “mypass566” is populated into the Password field on the existing page.

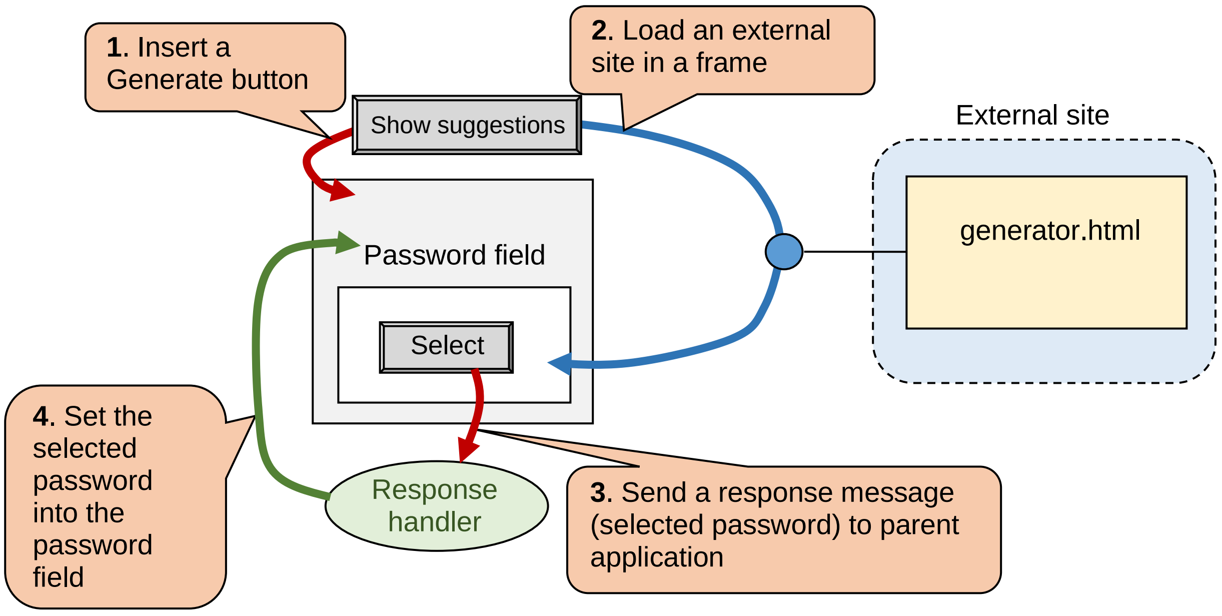


Password: mypass566 [Show suggestions]

This sample shows how to extend your page with using another web
The "Show suggestions" button was dynamically added by WebExte

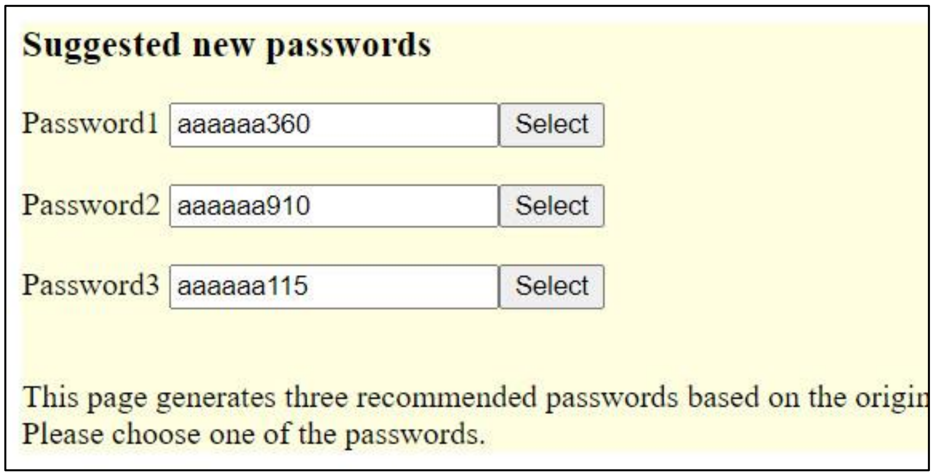
The above flow can be shows as the following steps:

Step 1: WebExtension dynamically adds a “Show suggestions” button in the form.
Step 2: When user clicks on the 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
Show suggestions
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

6.2 External Web Application

This is an independent web application and developed as another web app from the existing web application. This page aims to suggests passwords so that user can populate a password by choosing from suggested password list.



Suggested new passwords

Password1	aaaaaa360	Select
Password2	aaaaaa910	Select
Password3	aaaaaa115	Select

This page generates three recommended passwords based on the origin
Please choose one of the passwords.

Below is the source code of the above app:

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;
  WebExtensionClient.sendResponse('passwordGenerator', newPassword);
}
</script>
</head>
<body><div style="background-color:lightyellow;">

<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>

</div></body>
</html>
```

This page calls a WebExtensionClient API, so it loads WebExtensionClient.js file in its <head> tag.

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

When user clicks on one of Select button, sendResponse API is called to return the selected password to the existing application.

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

6.3 Password Generator Updater

PasswordGeneratorUpdater inserts HTML elements onto the existing index.html page.

PasswordGeneratorUpdater.java

```
package PasswordGenerator;

import java.util.Map;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import webextension.ResponseHtmlUpdater;
import webextension.annotation.WebExtensionFilter;

@WebExtensionFilter(contentType="text/html",
                    servletPath="/PasswordGenerator/index.html")
public class PasswordGeneratorUpdater extends ResponseHtmlUpdater {

  public boolean update(Map<String, String> context, Document document) throws
Exception {
    Element head = document.selectFirst("head");
    if (head == null) return false;
    head.append("<script src='../WebExtension.js'></script>")
        .append("<script src='PasswordGenerator.js'></script>");

    Element input = document.selectFirst("input[id=password]");
    if (input == null) return false;
    input.after("<button id='myButton' onClick='openFrame();return false;'>Show
suggestions</button>");

    return true;
  }
}
```

As explained n Hello World sample, Updater must be extended from one of Updater Base classes. This sample also updates HTML content, thus, we will extend from ResponseHtmlUpdaer class.

```
public class PasswordGeneratorUpdater extends ResponseHtmlUpdater {
```

Next, we need to insert two <script> tags in <head> tag.

```
    Element head = document.selectFirst("head");
    if (head == null) return false;
    head.append("<script src='../WebExtension.js'></script>")
        .append("<script src='PasswordGenerator.js'></script>");
```

After that, we need to add one more HTML element that is a <button> to invoke lookup JavaScript function. The <button> can be added like the way we did for the <script> tags.

```
    Element input = document.selectFirst("input[id=password]");
    if (input == null) return false;
    input.after("<button id='myButton' onClick='openFrame();return false;'>Show
suggestions</button>");

    return true;
```

6.4 Custom JavaScript functions

Next, we need to implement the openFrame JavaScript function. It is implemented in PasswordGenerator.js.

PasswordGenerator.js

```
function openFrame() {
  // Define a URL of an external HTML with a parameter
  const password = document.querySelector('input[id=password]').value;
  const url = 'generator.html?'+encodeURIComponent(password);

  // Open a frame and load the HTML under the button.
  const button = document.querySelector('button[id=myButton]');
  WebExtension.openFrame(url, button);
}

// Add a ResponseHandler function to handle the response from the external HTML.
WebExtension.addResponseHandler('passwordGenerator', response => {
  console.log('res handler response=('+response+')');
  document.querySelector('input[id=password]').value = response;
});
```

The openFrame function opens a new frame on the page by using openFrame API of WebExtension.

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

The second parameter requires a HTML element as an extension point. The frame will be opened under the button tag.

In this JavaScript file, there is another function call below:

```
WebExtension.addResponseHandler('passwordGenerator', response => {
  console.log('res handler response=('+response+')');
  document.querySelector('input[id=password]').value = response;
});
```

The addResponseHandler adds a ResponseHandler function that handles the response content from the external web app. In this sample, generator.html sends a selected password, hence the password is stored in the response variable and the function sets the response value into password field.

6.5 Web.xml settings

Finally, you need to register the PasswordGeneratorUpdater to be called from ResponseFilter. If you have added settings for HelloWorld, required init-param should be already added. So, you just need to add PasswordGeneratorUpdater file name in updater parameter.

webapps\samples\WEB-INF\web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">

  <filter>
    <filter-name>WebextensionResponseFilter</filter-name>
    <filter-class>webextension.ResponseFilter</filter-class>
    <init-param>
      <param-name>-s</param-name>
      <param-value>C:\apache-tomcat-9.0.0\webapps\samples</param-value>
    </init-param>
    <init-param>
      <param-name>-classpath</param-name>
      <param-value>
        C:\apache-tomcat-9.0.0\webapps\samples\WEB-INF\lib
      </param-value>
    </init-param>
    <init-param>
      <param-name>updater</param-name>
      <param-value>
        HelloWorld.HelloWorldUpdater.javax
        AddressLookup.AddressLookupUpdater.java
        PasswordGeneratorUpdater.PasswordGeneratorUpdater.java
      </param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WebextensionResponseFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

7 Two ways to add Extension

So far, extension triggers (e.g. a button or link to invoke a custom JavaScript function) are added in Updater class, but there is one more way. This chapter explains the other way to add an extension onto existing page using the sample site below:

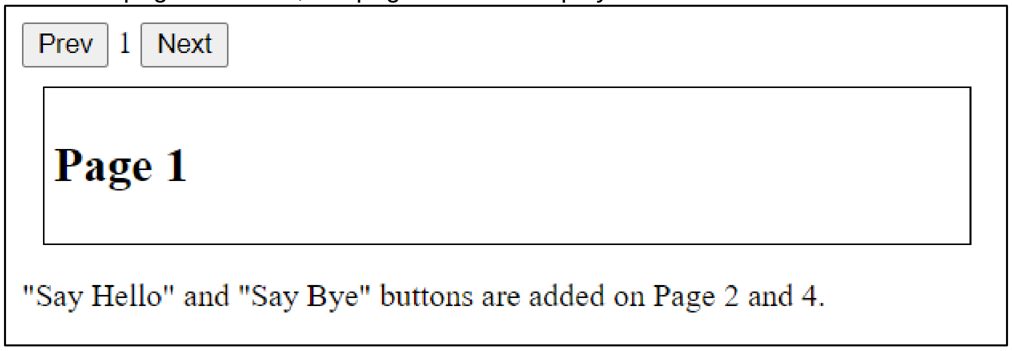
https://github.com/web-extension/master/tree/master/samples/SPA

7.1 Original HTML Code

This sample has only a single HTML page that is **index.html**.

```
<html>
<head>
<script src="../WebExtension.js"></script>
<script src="Page2.js"></script>
<script src="spa.js"></script>
<script>
window.onload = function() {
  render(1);
};
</script>
</head>
<body>

<button id='prev' onClick='prev();return false;'>Prev</button>
<span id='pageNo'>1</span>
<button id='next' onClick='next();return false;'>Next</button>
<br>
<div id='content' style='border: 1px solid black; padding: 5px; margin: 10px;'></div>

<p>
  "Say Hello" and "Say Bye" buttons are added on Page 2 and 4.<br>
</p>

</body>
</html>
```

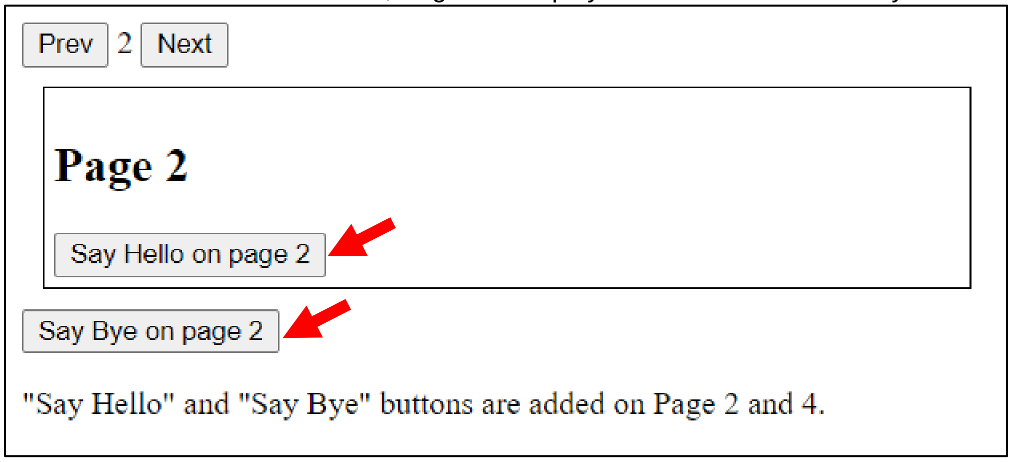
When the page is loaded, the page below is displayed:



Page 1

"Say Hello" and "Say Bye" buttons are added on Page 2 and 4.

When user clicks on Next button, Page 2 is displayed with two buttons newly added.



Page 2

"Say Hello" and "Say Bye" buttons are added on Page 2 and 4.

They are not added by using Updater class. They were dynamically added by JavaScript (Page2.js) when the page has been loaded.

Page2.js

```
WebExtension.addExtensionRegister(() => {
  const target = document.getElementById('page2');
  if (!target) {
    const button = document.getElementById('myButton2B');
    if (button) button.parentNode.removeChild(button);
    return;
  }

  const button = document.getElementById('myButton2A');
  if (!button) {
    target.insertAdjacentHTML('afterend', "<button id='myButton2A' onClick=\"alert('Hello,
World!');return false;\">Say Hello on page 2</button>");
    target.parentNode.insertAdjacentHTML('afterend', "<button id='myButton2B'
onClick=\"alert('Bye, World!');return false;\">Say Bye on page 2</button>");
  }
});
```

The addExtensionRegister function accepts a function to register extension trigger(s). The function is called on all pages. Even if user stays on a same page, it is called frequently. Hence, the function should not take longer time to complete the task. To do so, the lambda function checks to see if page2 element exists, and it returns immediately if it does not exist. Also, before inserting the myButton2A, the function checks if the button exists. That is because the function is called regularly no matter if the button exists or not. Inserting myButton2A and myButton2B must be performed only when they are not inserted yet.

8 WebExtensionFilter Annotation

Updater class updates response content of the target page. Whether the page is target or not is determined by contentType and servletPath specified by WebExtensinFilter annotation. The annotation accepts the following parameters:

Parameter Name	**Description**	**Sample Value**
contentType	Invokes update method if the response contentType contains the specified value.	text/html
contentTypePattern	Invokes update method if the response contentType matches the specified pattern in regular expression.	text/(html|xml)
servletPath	Invokes update method if the servletPath starts with the specified value.	/path/testServlet
servletPathPattern	Invokes update method if the servletPath matches the specified pattern in regular expression.	^/path/$

contentType and servletPath

```
@WebExtensionFilter(contentType="text/html")
```

The above annotation accepts any HTML responses. If you want to limit to accept a specific HTML page, specify the full path in servletPath parameter.

```
@WebExtensionFilter(contentType="text/html" ", servletPath="/HelloWorld/index.html")
```

Now the update method in the Updater class will be invoked for the response of /HelloWord/index.html page.
If you want to accept any pages under /HelloWorld/, specify the partial path instead.

```
@WebExtensionFilter(contentType="text/html", servletPath="/HelloWorld/")
```

Using Regular Expression

If you want to use regular expression to specify the contentType and/or servletPath, use contentTypePattern, servletPathPattern parameters.

```
@WebExtensionFilter(contentTypePattern="text/(html|xml)")
```

The above annotation accepts any HTML and XML responses.

```
@WebExtensionFilter(contentType="text/html",
                        servletPathPattern="^/HelloWorld/index.(html|jsp)$")
```

The above annotation accepts index pages with both html and jsp extensions.

9 Web.xml Settings

WebExtensions basically does not require to modify source code of the target web application but web.xml. You need to configure settings of ResponseFilter and HttpProxy in web.xml file.

9.1 RespnseFilter

ResponseFilter is a web filter to handle response contents and gives you a chance to update the content before sending it back to browser.
To enable your updater class, you must specify the class including package path as an initial parameter of the ResponseFilter.
If you have multiple Updaters, those must be separated by a newline code (an updater must be defined in each line).

```
<filter>
  <filter-name>WebextensionResponseFilter</filter-name>
  <filter-class>webextension.ResponseFilter</filter-class>
  <init-param>
  <param-name>updater</param-name>
    <param-value>
      mypackage.YourUpdater1
      mypackage.YourUpdater2
      mypackage.YourUpdater3
    </param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>WebextensionResponseFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

While you are in development phase and your updater would be modified often, source file name can be specified instead of class name. By doing so, the source is dynamically compiled and reloaded whenever you modify the source code. But it requires additional parameters. Below is an example of settings:

```
<filter>
  <filter-name>WebextensionResponseFilter</filter-name>
  <filter-class>webextension.ResponseFilter</filter-class>
  <init-param>
    <param-name>-s</param-name>
    <param-value>TOMCATROOT\webapps\samples</param-value>
  </init-param>
  <init-param>
    <param-name>-classpath</param-name>
    <param-value>
      TOMCATROOT\webapps\samples\WEB-INF\lib
    </param-value>
  </init-param>
  <init-param>
  <param-name>updater</param-name>
    <param-value>
      mypackage.YourUpdater1.javax
      mypackage.YourUpdater2.javax
      mypackage.YourUpdater3.javax
    </param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>WebextensionResponseFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The table below shows a list of init-param that ResponseFilter accepts:

Parameter Name	**Description**	**Required**
updater	Specifies an updater class name or java/javax source file name. Multiple values can be specified (Newline separated)	Yes
-s	Specifies the directory where to place source files.	Yes if updater contains java/javax
-classpath (or -cp)	Specifies where to find user class files. Multiple values can be specified (Newline separated)	Yes if updater contains java/javax
-d	Set the destination directory for class files. If -d is not specified, the class files are generated in the directory specified by -s parameter.	No
-encoding	Set the source file encoding name, such as UTF-8. If -encoding is not specified, the platform default converter is used.	No

9.2 HttpProxy

HttpProxy is a servlet to enable client to execute a cross-origin HTTP requests. The servlet class is defined as /WebExtensionHttpProxy by using WebServlet annotation.

```
@WebServlet("/WebExtensionHttpProxy")
```

So, you do not have to configure your web.xml to enable the proxy if **metadata-complete** in your web.xml is set to **false.** It is automatically detected as a servlet and mapped to the path "/WebExtensionHttpProxy".

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0"
  metadata-complete="false">
```

However, if the metadata-complete is set to **true**, the following entries must be added in your web.xml:

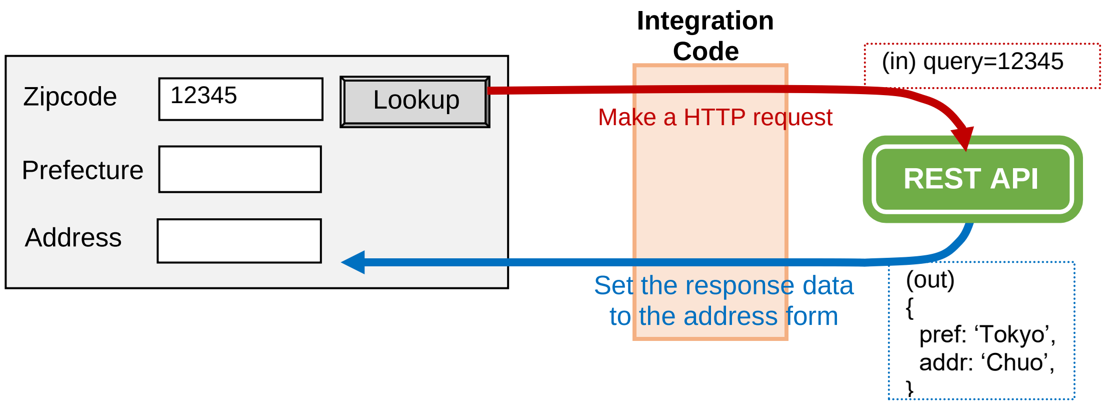
```
<servlet>
    <servlet-name>WebextensionHttpProxy</servlet-name>
    <servlet-class>webextension.HttpProxy</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>WebextensionHttpProxy</servlet-name>
    <url-pattern>/WebExtensionHttpProxy</url-pattern>
</servlet-mapping>
```

10 Extension Patterns

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

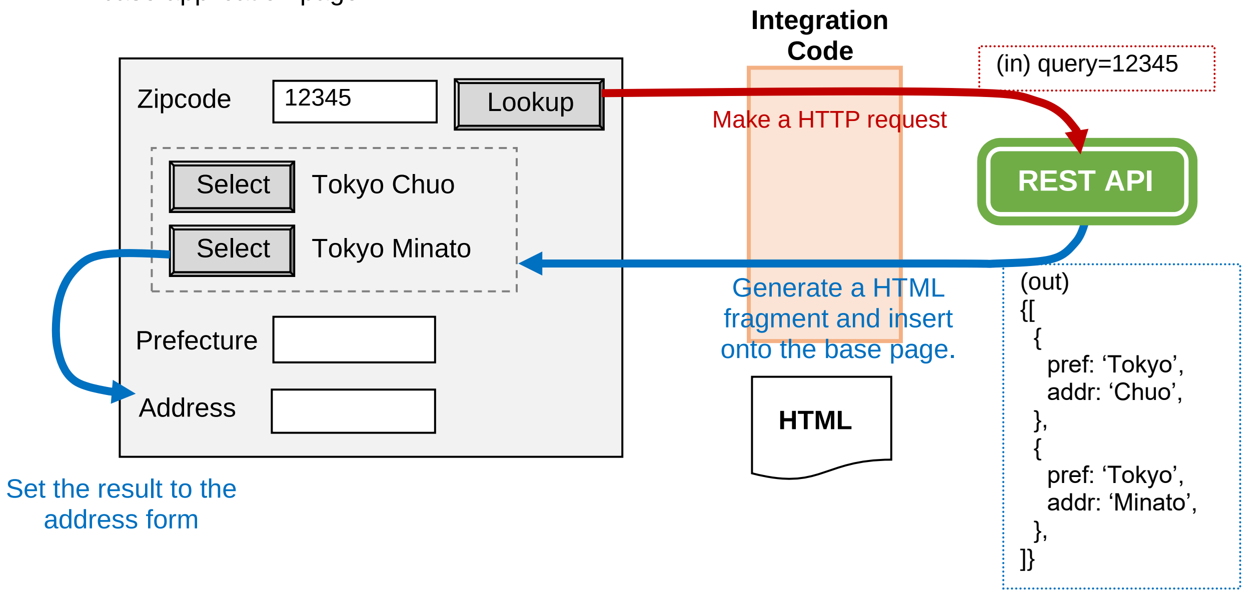
10.1 No UI

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.



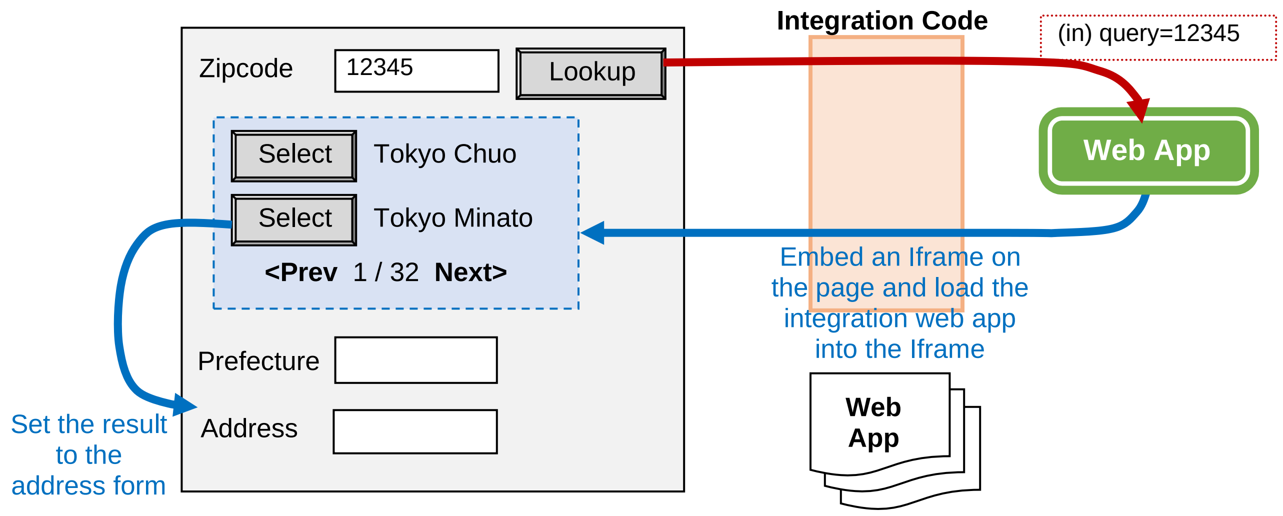
10.2 HTML Fragment

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.



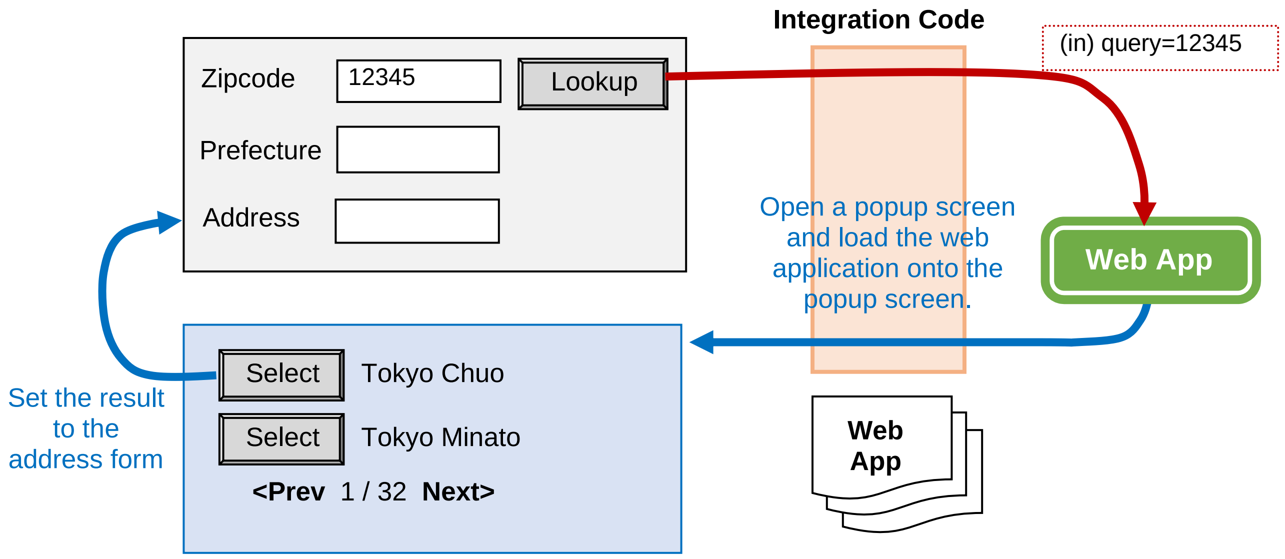
10.3 Embedded IFrame

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.



10.4 Popup IFrame

This pattern is technically same as Embedded 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.

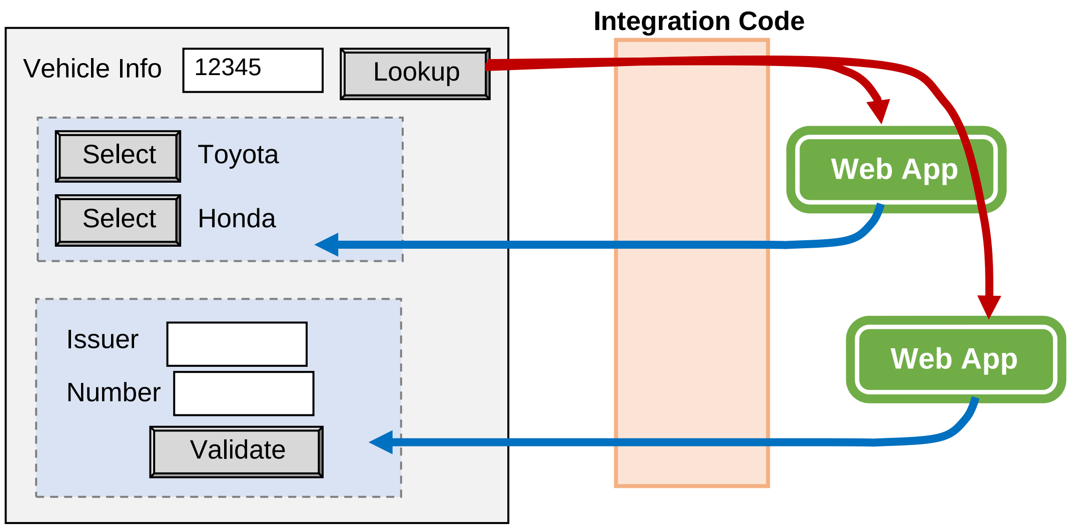


10.5 Combination Patterns

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 Embedded 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.

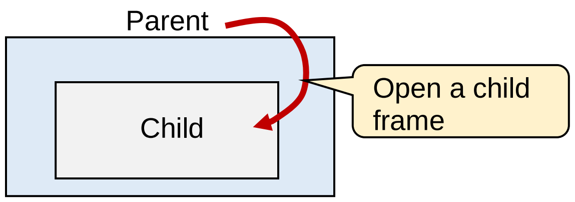


11 API Reference – WebExtension.js

This chapter lists WebExtension JavaScript APIs.

11.1 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 }
);
```

11.2 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.

11.3 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

11.4 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

11.5 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:

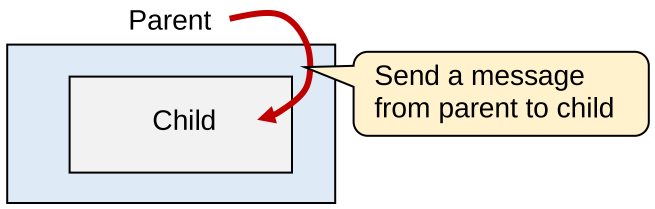
Name	Value Type	Description	Required
method	String	Specifies a HTTP method. **GET** or **POST**.	Yes
url	String	Specifies a URL to send the request to.	Yes
options	Object	Specifies option parameters. **body** Specifies body text or JSON object to be sent when POST method. **proxy** A URL of proxy servlet. 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), **json** or **xml**. **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
}
```

11.6 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);
    }
});
```

11.7 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);
}
```

11.8 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.

11.9 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>");
  }
});
```

11.10 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.

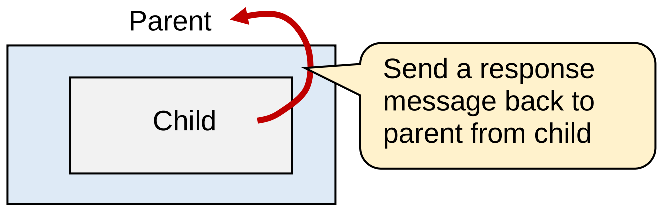
```
WebExtension.addResponseHandler('passwordGenerator', response => {
  document.getElementById('password').value = response;
});
```

12 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.

12.1 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
                })}
            >
```

12.2 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

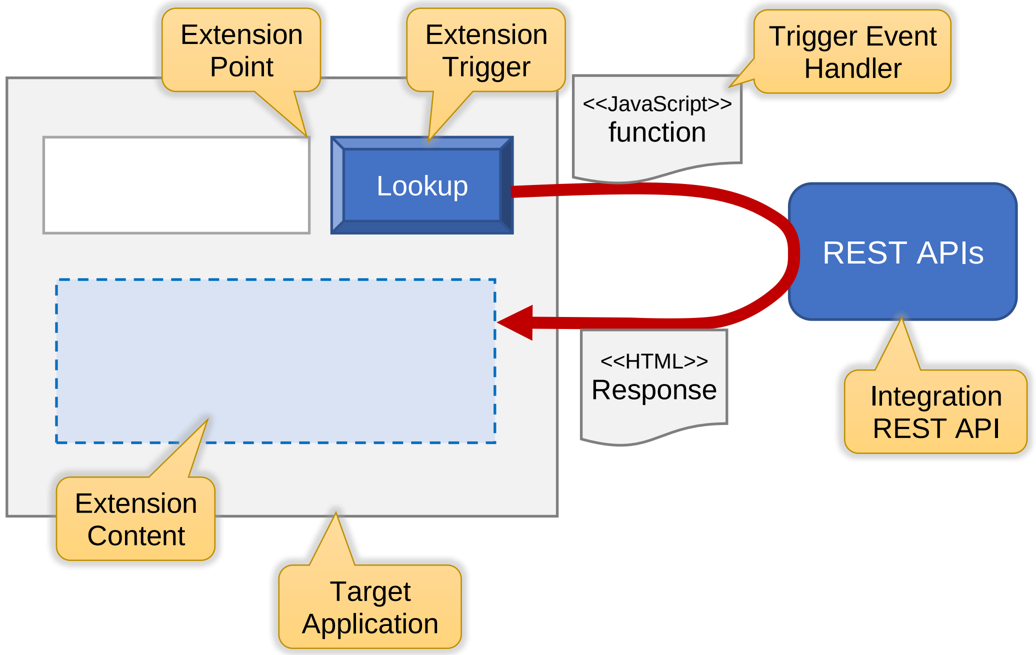
Example 1: Register a function that sets the list of addresses sent from the parent application.

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

13 Glossary of Technical Terms

The following are the technical terms used in this document. WebExtension is designed based on new concepts, so some of terms were newly created to explain the ideas. They are listed here to help the reader clarify the meaning.

Extension Content	A HTML document or fragment returned from integration web application.
Extension Point	An existing HTML element on the target page where an extension trigger will be inserted.
Extension Trigger	A HTML element to invoke a custom function that executes integration application or services.
Trigger Event Handler	A JavaScript function to handle an event fired by extension trigger. The function calls integration web application and/or REST APIs, and populates the response from them into the page of the target application if needed.
Response Updater	Java class to update response HTML from web server. There are three types of updaters depending on the object passed. String, Html and Xml.


Extension Point
Extension Trigger
Trigger Event Handler
<<JavaScript>> function
Lookup
REST APIs
<<HTML>> Response
Integration REST API
Extension Content
Target Application