Client Side AppSuite Application (CSASA)

This prototype demonstrates the feasibility of client, ie browser, installed applications to add or enhance AppSuite functionality.

The prototype may be useful in the following situations:

1. Customer learning lab. Fastest turnaround to see impact of examples.
2. Demonstrations: Instead of opening a debugger and pasting javascript that many may not understand, we can easily add and remove functionality by loading a Chrome extension.
3. UI prototypes: Ideas for new functionality can be easily created without impacting the mainline development sprints or requiring access to backend servers.
4. Customer prototypes: Rapid turnaround for some customer customizations
5. QA and Support debugging tools: Purpose built CSASA’s can be created by QA and support to help diagnose specific problems that may be impossible to reproduce. Additional console logging is one example that may be useful.
6. Proof that a CSASA infrastructure embedded directly in AppSuite as part of an AppStore offering can be effective.

To install a Chrome Extension that’s local, select Chrome/Preferences/Extensions or Tools/Extensions from the right menu icon. When you get the Extensions page, click the ‘Developer’ box at the top of the page. This will add an ‘Install Unpacked Extension’ link. Click and point to directory for the specific extension you want to install.

Packed extensions can be installed by submitting to Chrome App Store or by adding an ‘Add Extension’ to a web page somewhere.

After you install or refresh an extension, refresh the page that the extension is supposed to inject to and you should see the results if everything is working correctly.

The prototype attempts as much as possible to isolate the actual extension point script from the details of writing a Chrome extension. The Chrome extensions in the prototype use the basic extension mechanism of a JSON manifest file and supports HTML and JS files. If that is not sufficient, it is possible to create more complex extensions using native code.

A good article on extension is <http://www.jefvlamings.com/blog/chrome-extensions-for-dummies/>

Every extension will use have at least these three files

* Manifest.json: Describes all attributes of the extension
* ExtensionPoint.js: The actual AppSuite extension script
* Background.js: Optional background script to handle events. See manifest.json and MessageAdvertisement

An attempt to create an extension point namespace convention is implemented in order to prevent potential conflicts between extension point applications developed by different vendors. The namespace takes the form of

OxEp.DeveloperString.ExtensionName

For example;

OxEp.RAL.MessageAdvertisement

OxEp.Ox.SomeAppThatOxCreates

See MessageAdvertisement for an example template that can be reused.

Chrome extensions can run in multiple ways

1. As one or more ‘content scripts’ that look for page loads that match URL wild card string. When a page matches, one or more JS and/or CSS files are injected into the page. The injection can occur before or after the page is loaded. See general note on injection below.
2. As one or more ‘background scripts’ that look for events. When an event occurs, the script is injected. Background scripts can be ‘persistent’ and run constantly like a thread. Or than can be non-persistent and are only loaded when Chrome has an event the script has registered a listener for.
3. If the extension displays a page specific icon or a generic browser icon, clicking the icon can pop up HTML and/or inject JS

Some warnings on all forms of injected scripts

1. The Chrome extension management page is where you live when you are developing to load/reload your unpacked extension, pack your extension for distribution, and load the background script page for debugging. However this page does not accept ANY injections. I spent a long time trying to determine why my simple scripts, just logging to console, weren’t working before I learned this. Make sure you test your extensions on other pages!
2. You have to be very careful and aware of what context your extension scripts are running in. If in doubt, do a console.log() and see if the output is visible in the main page debugger. If not, look at the background page as indicated above and you should see it. This will confirm what context the script is running in.
3. All scripts have full access to the pages DOM, but no access to the page’s JS objects. This prevents name conflicts and protects each extension. My first experience with this was getting a ‘require is not defined’ error in my extension scripts. This was because the script had no visibility to the page. To work around this, I turn the script into a string and inject it directly into the actual page with document.write(). This puts the script in the correct execution context. I’ve tried to comment this, but it can be confusing.
4. Test your script with a BrowserAction icon and onClick handler first. This is a clean debugging environment and you know exactly when the script is being executed. Then convert to a content script or background script with more complex triggering rules.
5. Some Ox JS is loaded after the page is loaded. Make sure you use require([js you need], function() {What to do when js is loaded} )!

Updates

1. Overview of new object architecture and various ways to inject extensions (file, object, script)
2. Chrome UI (Selector)