Getting Started With Kendo UI And Chrome Packaged Apps

In addition to extensions, Chrome has **Installable Web Apps**. These fall into two categories; **Hosted Web Apps** and **Packaged Apps**. Hosted Web Apps are normal websites with a bit of extra metadata. They allow a site to be launched directly from the Chrome start screen and instead of prompting you for permission to use things like your location or the file system over and over again, Installable Web Apps only inform you once - on install. One of my personal fave's was/is TweetDeck.

Packaged Apps are a bit different. The packaged apps run completely outside of a browser tab, with no url bar or standard Chrome interface. Additionally you have access to native API's that are accessible off the Chrome object in JavaScript.

This means you are now able to create applications that can access hardware API's and take advantage of things like serial communication, native control over the file system, as well as built-in authentication to Google services.

Of course, all of this is done with HTML5, leaving you as the developer with the ability to write fully native applications that are cross platform and can be written entirely with web technologies.

Get Rolling With The Bootstrap

We have created a simple bootstrap project to show you how to get up and running with a new Chrome Package App built with Kendo UI.

You can download the bootstrap from GitHub. It contains everything that you need to get started building with Kendo UI. Make sure that you check you check your licensing for Kendo UI before you use it in production. The version included with the bootstrap is the CDN libraries are are not approved for production use without a license.

Lets take a look at the structure of the bootstrap folder.

packaged-app-bootstrap /

name	age	message	history
css	a minute ago	Add FileSystem access [burkeholland]	
img	a day ago	init [burkeholland]	
js	just now	Remove old files and references [burkeholland]	
README.md	a day ago	Initial commit [burkeholland]	
background.js	a minute ago	Add FileSystem access [burkeholland]	
index.html	a minute ago	Add FileSystem access [burkeholland]	
main.html	a minute ago	Add FileSystem access [burkeholland]	
manifest.json	a minute ago	Add FileSystem access [burkeholland]	

Here we have a simple project structure with **css, img,** and **js** folders. I bet you can guess what's in those! There is a **background.js** which will launch the **main.html** page. This is all defined in the manifest, which is the best place for us to start.

```
{
 "name": "<Application Name>",
 "description": "<Application Description>",
 "version": "1.0",
 "manifest_version": 2,
 "offline_enabled" : true,
 "app": {
  "background": {
    "scripts": [ "background.js" ]
  }
 },
 "permissions": [
   "fileSystem"
 ],
  "sandbox": {
    "pages": [ "index.html" ]
  }
}
```

The manifest tells Chrome how this application is configured.

name: Indicates the name of the application as it will appear everywhere. You should change this to be the name of your application.
description: The description that appears under the name.
version: The semantic version of the app. This is particularly important as you publish new versions to the Chrome Web Store.
manifest_version: Version 2 is the current manifest version for Chrome Apps and Extensions. In fact, if you load a **v1** manifest, it will tell you that v1 is about to be deprecated.
offline_enabled: Tells Chrome that this app can run without an active internet connection. Always have your app be offline capable.
app
background
scripts: These are the scripts that will be loaded into a background page that Chrome will create for you. In this case, we are loading the **background.js** file. The **background.js** file is actually what opens the app. Having an **app** element with a **background** element underneath it is what tells Chrome that this is a packaged application.
permissions: Any permissions that you will be needed are going to go here. I have the **fileSystem** permission in there as the bootstrap comes with an example of using the native FileSystem API.
sandbox: This is important and we're going to spend a bit of time on this subject:

Why The SandBox

Since you have access to native API's on the user's machine, there is a tightened CSP (Content Security Policy) that you cannot override. This means that things like evaluating strings as code in JavaScript will be blocked in the application. This is a problem in that many JavaScript UI toolkits and frameworks use either **eval** or **new Function** for templating. This is what makes Kendo UI templates so fast. Since there is no compilation step for Kendo UI or most other frameworks, the libraries are blocked by Chrome's CSP. In fact, if you don't specify the **index.html** file as sandboxed, you will see an error that looks something like this:

```
Uncaught Error: Invalid template:'<div class="k-group-indicator" data-#=data.ns#field="${data.field}" data-#=data.ns#title="${data.title || ""}" data-#=data.ns#dir="${data.dir || "asc"}"><a href="\#" class="k-link"><span class="k-icon k-arrow-${(data.dir || "asc") == "asc" ? "up" : "down"}-small">(sorted ${(data.dir || "asc") == "asc" ? "ascending": "descending"})</span>${data.title ? data.title: data.field}</a><a class="k-button k-button-icon k-button-bare"><span class="k-icon k-group-delete"></span></a></div>' Generated code:'var o,e=kendo.htmlEncode;o='<div class="k-group-indicator" data-'+(data.ns)+'field="'+(e(data.field))+'" data-'+(data.ns)+'title="'+(e(data.title || ""))+'" data-'+(data.ns)+'dir="'+(e(data.dir || "asc"))+'"><a href="#" class="k-link"><span class="k-icon k-arrow-'+(e((data.dir || "asc") == "asc" ? "up" : "down"))+'-small">(sorted '+(e((data.dir || "asc") == "asc" ? "ascending": "descending"))+')</span>'+(e(data.title ? data.title: data.field))+'</a><a class="k-button k-button-icon k-button-bare"><span class="k-icon k-group-delete"></span></a></div>';return o;' kendo.web.min.js:10
```

...which is a little misleading, but is the best information that Kendo UI can give you based on what's happened. If you were to actually step through the code, you would see that when Kendo UI tries to create a template (which it uses for pretty much every widget), Chrome blocks the use of **New Function** which is wrapped in a **try/catch** and the above error is the result.

Designating **index.html** as a sandboxed page allows full use of Kendo UI. But it comes with a caveat. The index.html page is completely blocked from using native API's.

What Now?

That seems like a bit of a deal breaker, but there is a simple work around. The sandbox can talk to the extension via PostMessage. Post messages in Chrome are fast. In fact, they are so fast that the Camera application we built uses them to send an entire video stream from the extension into the sandbox frame by frame.

All of this sounds like a bit of a headache so we have devised a very simple way for you to communicate between the app and the sandbox using a combination of PubSub and PostMessage and wrapped all of that for you in a simple jQuery plugin.

The Pkg Plugin

We created a jQuery plugin called the pkg plugin. Which is based heavily on Peter Higgins excellent PubSub Plugin. It's available off the jQuery object and it's how you are going to communicate in and out of the sandbox. The plugin does not have to be used with Kendo UI. It can be used with any UI framework and is completely open source.

Using it is easy. Just initialize the object and pass in the target of the object you want to communicate with. You will have an instance of it both inside the sandbox and in the extension. First initialize it and pass in the object you want to communicate with. In the bootstrap project, the sandbox is in an iFrame so this **window.top**. From the extension itself, it's **iframe.contentWindow**.

```
// from the extension
$.pkg.init(iframe.contentWindow);

// from the sandbox
$.pkg.init(window.top);
```

Now all you need to do to communicate between the sandbox and the extension is to send and listen for events. For instance, in the case of the bootstrap, the sandbox has a button which sends a message to the extension asking it for a file. The pkg **send** method takes two parameters:

1) The endpoint you want to send the message to
2) An array of parameters/object/functions you want to pass (optional)

```
// ask the extension for a file
$.pkg.send("/file/select");
```

Now in the extension you will need to be listening for this “/file/select” endpoint. Like this...

```
// listen at the /file/select endpoint
$.pkg.listen(“/file/select”, function() {
    // you can access native API’s here
};
```

This will allow you to fully communicate in and out of the application. When you want to pass parameters back and forth, you simply send them in as an array.

```
// in the extension
$.pkg.send(“/simple/message”, [“hello!”]);

// in the sandbox
$.pkg.listen(“/simple/message”, function(message) {
    // logs out “hello!”
    console.log(message);
}
```

Any parameters you send into the array will be parsed and passed into the listening function. If you want to stop listening for some reason, simply call the ignore command.

Putting It all Together

Now that we are able to talk to the application from inside the sandbox, anything is possible. You can call any API that the packaged app has access to, and then pass the result back down to the sandbox. Just remember that this process is asyncronous, so you will have to account for that in your code.

Let’s do a little demo and use some Kendo UI. We want to be able to have the user select a photo from their drive and then show that picture in a Kendo UI window. We’ll call this the picture browser.

To do this, we need to use the Chrome FileSystem API. We can invoke the file chooser window by calling **chooseFile.** We can then read that file as a DataURL and pass that into the sandbox.

In the bootstrap, the **background.js** file simply loads a **main.html** page when the packaged app is ready.

```
chrome.app.runtime.onLaunched.addListener(function() {

  // create a new window and position it with a fixed size
   var win = chrome.app.window.create('main.html', {
```

```
        width: 1024,
        height: 870,
        minWidth:900,
        minHeight:800,
        left:500,
        top:500
    });
});
```

The **main.html** file loads in jQuery, the pkg plugin and a **main.js** file. Inline script is also not allowed in packaged apps so we must put it all inside external files.

The **main.html** file also has the **index.html** file which is the sandbox as we defined it in the manifest above. It's in an iFrame so we can communicate directly with it using the **pkg** plugin. The iFrame is styled to take the whole screen all the time so you can't tell it's there.

The **index.html** file is sandboxed so we can do pretty much anything we want there, including using Kendo UI and inline JavaScript.

The **index.html** file contains the code to actually display an image. It is the UI.

```
$(function() {
    // initialize the pkg plugin
    $.pkg.init(window.top);

    // create a new kendo ui window and assign its reference to the win variable
    var win = $("<div></div>").kendoWindow({
        modal: true,
        visible: false
    }).data("kendoWindow");

    // assign a click event to the button
    $("button").on("click", function() {
      // send the message to the packaged app to open the native 'select file' window
      $.pkg.send("/select/file");
    });

    ...
```

The **main.js** simply responds, selects an image from the file system and sends it back.

```
(function($) {
    // get a reference to the iframe that holds the app
    iframe = document.getElementById("iframe");
```

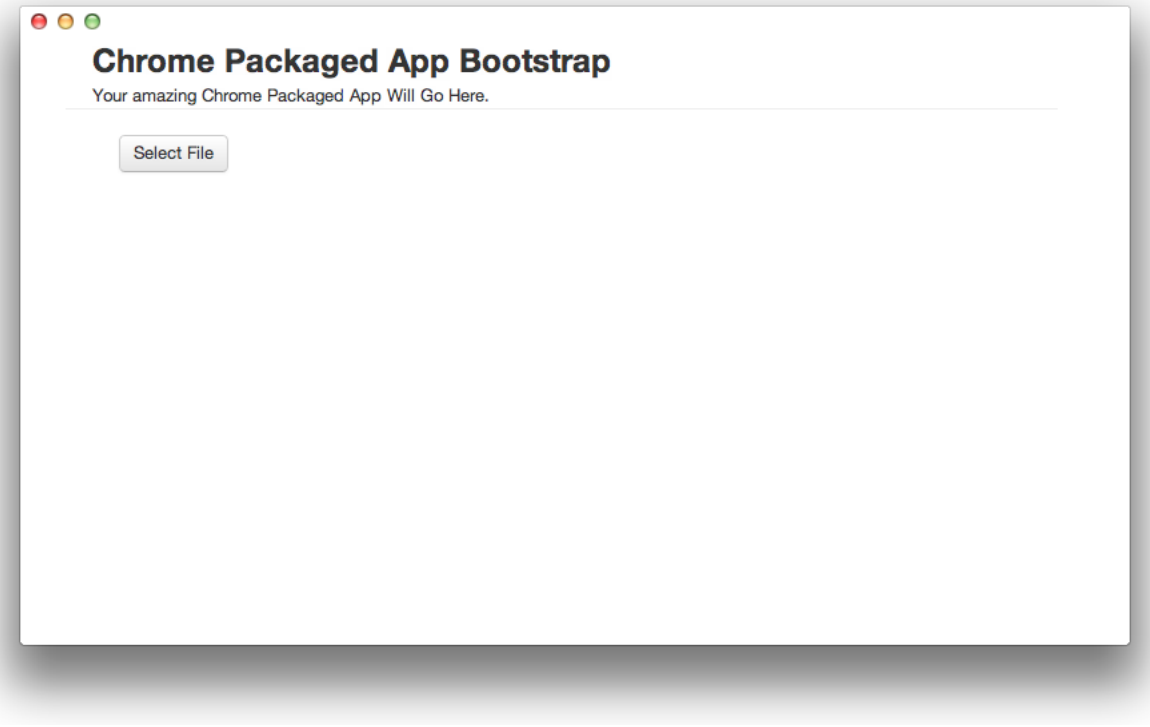
```
        // initialize the pkg plugin passing in the the iframe where the app lives.
        $.pkg.init(iframe.contentWindow);

        // subscribe to the /select/image message and return an image using
        // the native api file picker in chrome packaged apps
        $.pkg.listen("/select/file", function() {

            chrome.fileSystem.chooseFile({ type: "openFile"}, function(entry) {

                // this gives us a file entry. We just need to read it.
                entry.file(function(file) {

                // create a new file reader
                var reader = new FileReader();

                // create an event for when the file is done reading
                reader.onloadend = function(e) {
                // tell the postman to deliver this to the sandbox
                $.pkg.send("/file/loaded", [this.result, "loaded"])
              }

            // read the file as a data URL
            reader.readAsDataURL(file);

          });

        });
    });

})(jQuery);
```

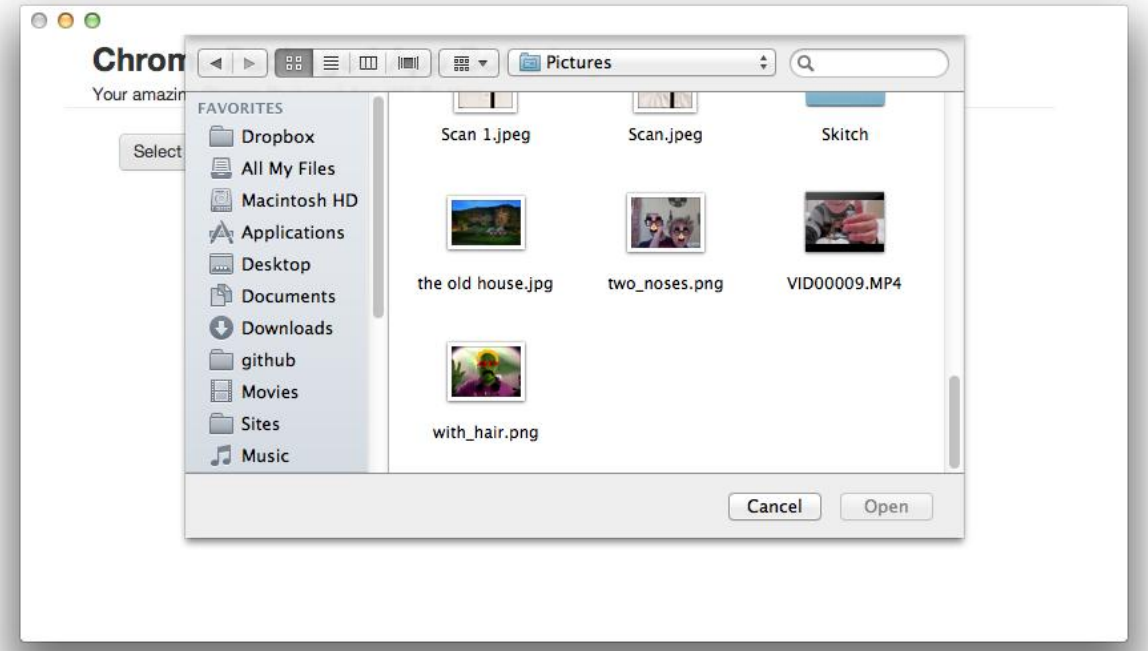
Inside the sandbox, we will listen to the **/file/loaded** endpoint and pop open the [Kendo UI Window] while setting it's source.

```
// listen on the image loaded endpoint
    $.pkg.listen("/file/loaded", function(src) {
        // create a new image
        var img = new Image();
        img.src = src;
        // when the image has loaded
        img.onload = function()  {
            // assign this image as the content of the kendo ui window and open it
            win.content(img).center().open()
```

```
    }
});
```

And here is what the bootstrap looks like. It's plain, simple and should get you off and running on building your app.




Chron
Your amazin
Select
Pictures
FAVORITES
Dropbox
All My Files
Macintosh HD
Applications
Desktop
Documents
Downloads
github
Movies
Sites
Music
Scan 1.jpeg
Scan.jpeg
Skitch
the old house.jpg
two_noses.png
VID00009.MP4
with_hair.png
Cancel
Open


Chrome Packaged App Bootstrap
Your amazing Chrome
Select File

Next Steps

Now that that we have the complete architecture setup for using Kendo UI inside of a Chrome Packaged Application, there is nothing you are limited too with Kendo UI or a Chrome Packaged App. Use any of the widgets or framework components exactly the same way that you would on the web.

You can download the Packaged Apps With Kendo UI Bootstrap here. If you want to check out a very large application done with Kendo UI, the entire code for Chrome Camera is on GitHub as well and you can get that here.

Power

Kendo UI and Chrome Packaged apps are a powerful combination. Think of being able to write cross-platform web applications that are completely native and all done in HTML5. Kendo UI is a complete package for building HTML5 Applications, and Chrome Packaged Apps take it off the web and bring it to the desktop.

Download Kendo UI and create beautiful native applications with Kendo UI and the power of HTML5 and Google Chrome.