Docs

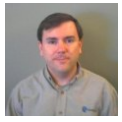
Tutorials and Labs

Themes

Script Portlet

Inter-Portlet Communication in IBM Script Portlet Using Public Render Parameters

Using Single Page Applications with Script Portlet



MICHAEL BURATI / APRIL 14, 2015 / 2 COMMENTS

One of the powerful new features of the Script Portlet for IBM WebSphere Portal is the ability to push or import a JavaScript, HTML, and CSS-based Single Page Application

Parameters

Business Process Manager Sample for IBM Script Portlet in WebSphere Portal

Using Node.js Build Tools with Script Portlet

Using an AngularJS Single Page Application as a Script Portlet

Using a Backbone.js Single Page Application as a Script Portlet

Accessing Server-Side Data in Script Portlet Using a Web Content Manager JSP Component

Getting Started with Script Portlet

(SPA) containing multiple artifacts, including images and JSON configuration, into a Script Portlet. JavaScript SPA developers can develop the application outside of the portal and portlet environment, by using their existing familiar tools. Then, they can push or import the result into a Script Portlet when ready. As with prior releases, multiple Script Portlet applications can be placed on a portal page, turning what was previously a single page web application into multi-portlet portal sites and applications.

The Script Portlet reference documentation provides a simple overview and getting started information, for those just playing with Script Portlet for the first time and wanting to push or import existing sample applications. When you are ready to start building more complex applications, it helps to know a little more about what happens during push and import. For example, where do the artifacts go and what does push and import support for URL references between the artifacts. This document provides that information, and more, to help you successfully develop Script Portlet based applications for WebSphere Portal.

Overview of the Steps Involved

The simple elevator pitch that describes developing and pushing or importing a script portlet is explained in the following steps:

1. Develop your application standalone with a main index.html page and other files such as Javascript, CSS and images:

Script Portlet and Source Code Management across Environments

Using Single Page Applications with Script Portlet

Script Portlet Development Best Practices

Using Command Line Push to Deploy Script Portlet Applications

Adding a Script Portlet Application to the Site Toolbar in WebSphere Portal 8.5

Script Portlet Support for External Editors

Responsive, Mobile, and Worklight

- Use your favorite editor to work with files on your local system
- Your application can include multiple Javascript, image and other files, for example JSON files.
- The main HTML page (typically index.html) should be at the root of the application folder tree and will be the primary page rendered in the script portlet

2. Test your application locally, as you would any script based single page web application.
3. Push your application to a WCM Site Area leveraging the command line utility described here: Using Command Line Push to Deploy Script Portlet Applications .
4. Add the script portlet to a portal page using the toolbar's "Content" palette of available "Script Portlet Applications".

Application Folder Layout

A properly laid out application tree (folder hierarchy on disk) should have the following structure:

- index.html (or alternative specified primary page for the application) should be at the root of the folder structure
- subfolders of the root directory for images, js, css, json, etc.

- Use relative subfolder URLs from the main HTML page down to the artifacts stored in subfolders; for example, <img src="images/background.png">
- Simple "../" relative links from a CSS style to images in a sibling folder are supported as of version 1.3. However, it is not a best practice to use "../" parent relative URL syntax, such as "../folder/filename" other than for typical CSS URL references.
- JavaScript references to artifacts with URL strings should be the entire relative path, such as "images/backgrounds/clouds.png". The Import/Push matches paths in the zip file and creates WCM elements with the same folder structure as children of the Content Item representing the application.
- JavaScript logic that creates URLs dynamically from multiple components that are stored in variables cannot be found by the Import/Push logic that constructs the WCM elements and replaces matching URL paths with WCM plug-in markup to generate the URLs to those new WCM artifacts. Therefore, avoid such dynamic URL constructions in application JS intended for use in a Script Portlet. When dynamic Ajax URLs cannot be avoided, refer to the workaround for runtime URLs section.

For example:

```
index.html
images/
        background.png
        logo.jpg
js/
```

```
        app.js
data/
        config.json
css/
        main.css
```

Best Practices, Tips, and Restrictions

Before you start your script portlet development effort, familiarize yourself with suggested Best Practices, Tips, and Restrictions related to developing applications that work best in a multi-portlet portal environment that uses Script Portlet.

URLs to application artifacts

The importer and push service attempt to find quoted (single, double quotation marks, and parentheses) instances of paths to local application artifacts (e.g., “images/faces/mike.jpg”) in the application when storing the artifacts from the uploaded application, to the associated WCM Content Item and its child elements.

Local URls are replaced with WCM plug-in markup that generates the corresponding WCM URL to the element at render time.

Simple “../images/background.png” style “..” relative image URLs for CSS styles are handled with Script Portlet 1.3, but it is recommended to avoid the use of “..” components in relative URLs to be sure that local relative URLs are fixed up with WCM URL generation markup on import/push.

For example, <img src=”images/skiing.png”> is modified during push or import. It replaces the local relative URL with the WCM plug-in markup that generates the URL to where the imported image is stored as a child element of the application’s web content item. The following example shows the updated URL:

```
<img src=”[Plugin:ScriptPortletElementURL element=”images/skiing.jpg”]”>
```

URLs built up by JavaScript frameworks at run time in the browser cannot necessarily be detected at “import or push” time. You cannot fix it with the WCM URL plug-in markup. For example, the following dynamically generated URLs would not be recognized as URLs by Import and Push and thus is not fixed:

```
var contacts = [ {“name”: “Jane”}, {“name”, “John”}, … ];
function getImageName(contactName) {
  return “images/” + contactName + “.jpg”;
}
```

Where the following URLs in the page HTML and inline JS in the main page should get fixed with WCM element markup referring to the appropriate image components:

```
<img src="images/banner.png"/>
<script type="text/javascript">
var contacts = [
{ "name":"Jane",   "pic": "images/faces/jane.jpg" },
{ "name":"John",   "pic": "images/faces/john.jpg" },
…
];
```

Workaround for run time URLs:

There might be some frameworks or existing applications where you just cannot avoid having such a relative URL to local application artifacts generated at run time.

Script Portlet 1.3 provides a new optional feature, where anelement directive attribute: data-scriptportlet-generate-url-map="true" indicates to the importer and application push service that your application is going to need to look up WCM URLs, that use application relative URLs at run time. This feature generates a Javascript-based map in the application page. It maps a relative URL to the associated WCM URL, along with a method on the namespaced spHelper API, to retrieve a WCM URL from the specified

relative URL; for example: var runtimeImageURL = __SPNS__spHelper.getElementURL("images/faces/mike.jpg"); // return the WCM URL storing the app artifact images/faces/mike.jpg

NOTE: It is not typical to store all of your contacts and employee images directly with your portlet. Instead, the lists of employees and contacts typically come form a back-end REST service and include absolute URLs to where those images are stored in a static artifact-serving environment. The previous example URL illustrates how to use this API at run time if your application has no choice but to generate and look up an application relative local URL while the application is running.

Unique Names and IDs

As described in the referenced Best Practices document, it is best to proactively avoid collisions across portlets by using unique names for JavaScript variables and HTML ID values. There are differing opinions out there as to how to best namespace your global names and IDs, and you should choose the method that works best for your projects. There might be times though that manually specified unique names might not be enough. For example, the atypical case where the same portlet is used multiple times on a single portal page. In these cases, you can use portlet namespaces to provide uniqueness where necessary.

- Leverage the Script Portlet 1.3 feature that provides ability to replace __SPNS__

token with Script Portlet Namespace plug-in when the application is imported or pushed to the server.

- The Script Portlet Namespace plug-in renders the portlet namespace when the portlet artifact renders to the browser

For example, the following syntax allows your application to run locally.

…
…
var myContainer__**SPNS**__ = document.getElementById("myContainer__**SPNS**__");

When pushed or imported to the server, the __SPNS__ token (SPNS surrounded by two underscore characters on each side), is replaced by the WCM plug-in markup for a plug-in that generates the unique portlet namespace at render time.

…
…
var myContainer**[Plugin:ScriptPortletNamespace]** =
document.getElementById("myContainer**[Plugin:ScriptPortletNamespace]**");

When the WCM markup is rendered on the page, each instance of the plug-in markup is replaced with that portlet's unique namespace ID. Therefore, further scoping the JavaScript variable names and HTML IDs to that portlet instance on the page. Again, if

your global JavaScript variable names and HTML IDs are unique enough to begin with, it might not be necessary. It is needed when you do not know what other portlets might be used on the page with yours. It is also needed when your script portlet application needs to support being used in more than one portlet on a single portal page.

If you use the “Export” feature to export a script portlet to a downloaded zip archive of the application, the script portlet namespace plugin markup is replaced by the __SPNS__ token, so that the application might again be run standalone, outside of portal, for testing purposes.

Advanced Information: What Does Import or Push really do with the zip contents/artifacts?

This section is for the advanced user and not necessary to understand for basic use of Script Portlets and the Import, Export or Push functionality, but it can help developers understand better what’s going on under the covers, and thus how to diagnose issues on their own should they encounter an issue with specific complex applications.

When you import or push an application folder or zip to a script portlet instance, here are the basic steps that happen behind the scenes and what artifacts are created (and where) as a result of that import.

- Assuming the ZIP makes it through the maximum size check (100mb by default) and

valid-zip check, it passes along to the following steps

- The main page of the application (primary page for the portlet) must be identified, either explicitly (via Push) or via confirmation page in Import if there is more than one html file at the root of the imported application.
- A list of valid files (files of known type – not .exe .bat etc) including the folder path structure, is generated from the incoming zip.
- Because WCM "elements" must be addressed by generated URL and not by simple folder path, the "content" of every artifact being imported (html, js, css, json) is searched for those exact incoming paths (eg, "images/background.png", "js/app.js", "css/main.css") and "replaced" with WCM Plugin markup that is used to generate the actual WCM URL to the WCM element at runtime. This is a key fact to note because if Import/Push cannot find the URLs (eg, they're buried in JS logic that constructs URLs dynamically at runtime) then those URLs cannot be fixed up to point to the actual WCM elements where the artifacts are stored.
- The main (primary) HTML page will then be stored in the element named "HTML" in the WCM Content Item associated with the Script Portlet instance that you're doing the import or push to.
- For each artifact of known type (image types, js, css, json) in the zip, create a WCM Element as a child of the WCM Content item for the given Script Portlet instance with the full subfolder path to that artifact as specified in the zip.
- NOTE: WCM doesn't allow a sub-folder hierarchy for child elements of content items,

so the folder path structure is simulated by leaving the forward slashes in the “name” of the WCM content item’s child element (eg, “images/background.png”). The path structure uses a forward-slash canonical form of the folder structure, regardless of which operating system the application was imported or pushed from.

Tip: Look at your imported HTML in the Script Portlet Editor, you can see the plugin markup being used to generate URLs to the WCM elements created for the additional artifacts for your application.

Note: As described in the Script Portlet Importing Application reference documentation, WCM Plugin markup that’s specific to portlets can only be used in the main HTML at this time (eg, not in JS artifacts) because only the main HTML page is being rendered by the WCM Rendering Portlet. The rest of the artifacts are being accessed directly via URL from the browser and not running via the rendering portlet, so plugin markup that requires a portlet context is not supported in those artifacts.

Back End Services May Be Called Through Portal’s AJAX Proxy

Some cross-domain AJAX request challenges may be resolved by leveraging Portal’s AJAX Proxy to access backend services from your portlet application(s).For backend services that require additional authentication, you may leverage the Ajax Proxy’s ability

to use Credential Vault support of basic authn, form based authn, SAML, certificate and OAuth.

2 Comments

Thai Hau

AUGUST 10, 2015 AT 10:14 AM

Any document show how to implement OAuth on REST API by using Ajax Proxy? Cant find OAuth implementation on http://www-01.ibm.com/support/knowledgecenter/SSHRKX_8.5.0/mp/dev-portlet/outbhttp_authntct.dita?lang=en

REPLY

Michael Burati

SEPTEMBER 9, 2015 AT 2:39 PM

I'm sorry, I don't know of an article showing integration of OAuth using AJAX Proxy, and a quick search did not turn up one that I could find. I checked with the portal ajax proxy experts that provided some of the above info and while the out of the box documented integration is as you found in the link above, there's also the ability to retrieve credentials (which could be a token) from credential vault and the ability to write a custom service extension as documented here: http://www-01.ibm.com/support/knowledgecenter/SSHRKX_8.5.0/mp/dev-

portlet/outbhttp_progr_xtns.dita?lang=en

I hope that info helps,

..Mike Burati

The postings on this site are my own and do not necessarily represent the positions, strategies, or opinions of IBM.

REPLY

Leave a comment

Name *

Email *

Comment

Submit Comment