Adobe CQ Help / Developing CQ components | Best practices

- Suryakand Shinde and Samartha Vashishtha

Target audience	CQ developers
Applicable CQ versions	5 and later

A component is a modular and reusable unit that provides specific functionality for presenting content on a website powered by Adobe CQ. Programmatically speaking, components are self-contained pieces of code residing in a repository folder.

CQ ships with several out-of-the-box components that provide comprehensive functionality for website authors. If necessary, CQ developers can write custom components to extend the default functionality.

Some planning questions

To the top

Before you code your custom component, answer these planning questions:

- What main functionality do you plan to deliver through the custom component?
- Is your custom component user interface-only or do you plan to associate business logic with it?
- If you plan to associate business logic with the custom component, does that business logic stay the same for all CQ websites across which you want to use the component?
- Is the custom component abstract? An abstract component doesn't provide any functionality by itself. However, other components inherit from the abstract component and add their own user interface or business logic.

These questions help define the purpose of your custom component.

Best practices for developing components

To the top

As you start coding your custom component in Java, JavaServer Pages (JSP), JavaScript (JS), and CSS; keep in mind the following best practices:

- Keep business logic independent of the user interface layer (JSP). Extract the business logic in a separate class, so that you can reuse it in other parts of the application. This distinction between the user interface logic and business logic simplifies application maintenance. Further, in cases where different business logic is required for different components, you can extend existing classes to quickly add the required functionality.
- Unify application-level initialization in a single place. Ensure that common variables—such as user account information and session variables—are not initialized repeatedly at a component level. These variables should be initialized only once per page request.
- Do not define CSS styling in components. This practice keeps components loosely coupled from a styling standpoint and lets you restyle them whenever necessary. Also, define a convention for naming HTML elements, so that you can modify them through external CSS files.
- Use the JSP Expression Language (EL) liberally to ensure code readability.
- Use JSP beans (`jsp:usebean`) to access class (business) logic.
- It's OK to keep JavaScript code in a JSP file. However, if some JavaScript code is common to all components, move it to a dedicated JavaScript file.
- If you want to display elements from a combination of CSS and JavaScript files on your CQ

page, use the `<cqIncludeClientLib>` tag. This tag is a convenience wrapper around the `com.day.cq.widget.HtmlLibraryManager` service interface.

Component development examples

To the top

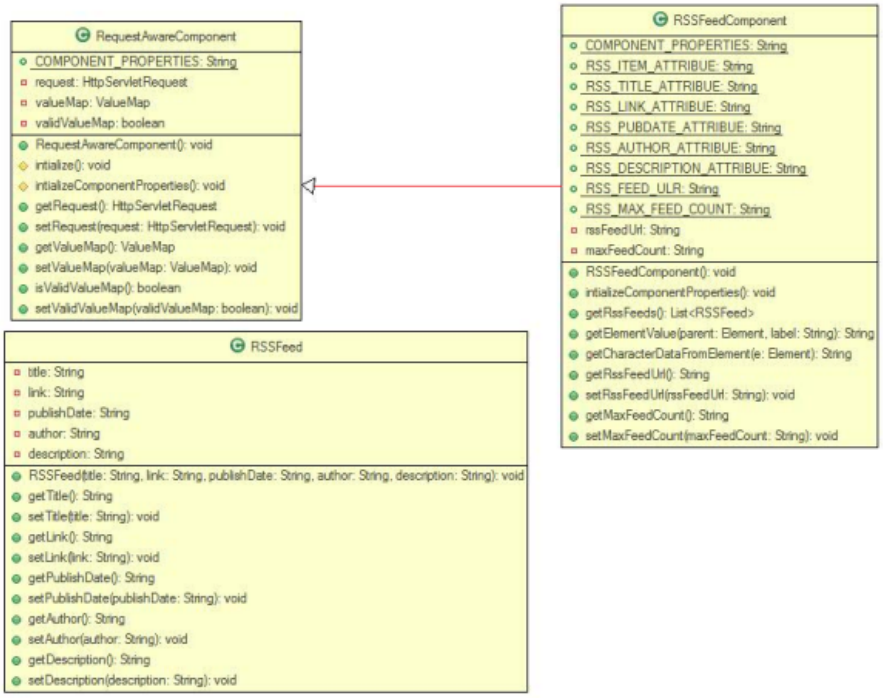
Build an RSS feed component

Suryakand Shinde from the Adobe CQ community illustrates component development through a detailed example on his blog. Shinde builds a reusable component that reads an RSS feed and displays it on a CQ website.

In line with the best practices described above, Shinde's custom component follows these guidelines:

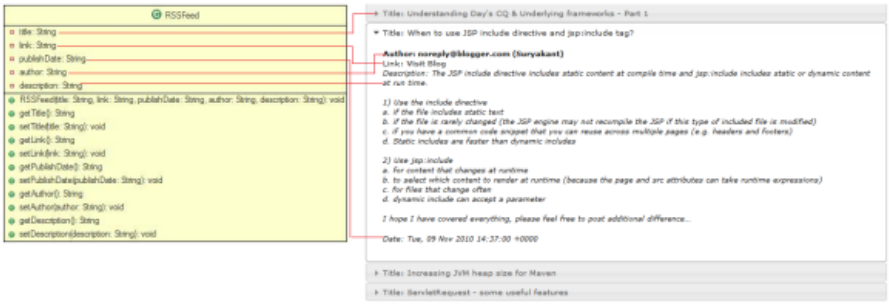
- The user must be able to configure the RSS feed URL as well as the number of RSS feed posts displayed at a time.
- Do not make the logic for parsing the RSS feed part of the component. The component should focus on displaying the content.
- To apply styles in the component, use a Cascading Style Sheet (CSS), which is configurable whenever necessary.

UML representation of classes in the example RSS feed component



Click here to view a larger version of the diagram.

Final component: Mapping between the RSSFeed class and an actual RSS feed entry



Click here to view a larger version of the illustration.

See Suryakand Shinde's blog for the details of these classes.

A tutorial on cq:includeClientLib using the jQuery user interface

Marcel Boucher, Senior Product Marketing Manager—Adobe, describes on his blog how you can leverage the `<cq:includeClientLib>` approach while developing CQ components.

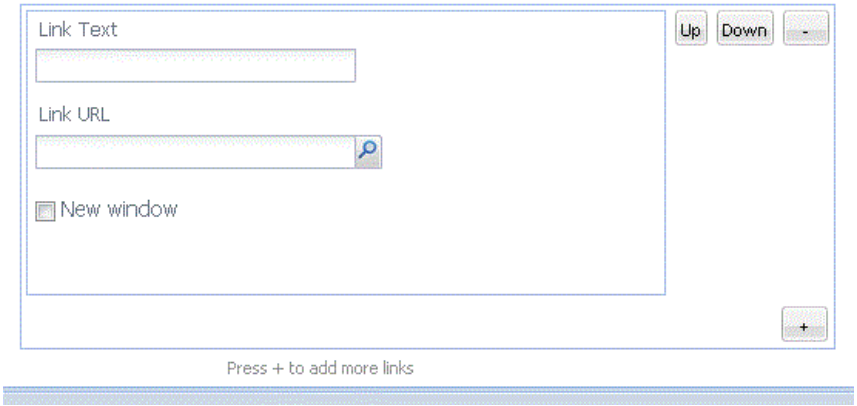
Build a component to find the most viewed pages on a website

Shishank Mathur from the Adobe CQ community describes on his blog how you can create a component to identify the most viewed/popular pages on a CQ website.

Build a multifield component of a custom xtype

Shishank Mathur has a detailed blog post discussing the sample implementation of a component that supports the addition of multiple term/value pairs using a + icon.

Sample component supporting the addition of multiple term/value pairs



Follow these steps to implement such a component:

1. Create a folder named `clientlib` in your application project.
2. Create a text file—`js.text`—and within that text file, specify the name of the JavaScript file that will hold the custom `xtype` definition. For example:

   ```
   #base=js
   CustomPathField.js
   ```

3. Define `CustomPathField.js`, such that it extends `CQ.form.CompositeField` and contains the attributes required for the custom `xtype`. For example:

```
1  /**
2  * @class MyClientLib.CustomPathFieldWidget
3  * @extends CQ.form.CompositeField
4  * This is a custom path field with a Link Text and a Link URL
5  * @param {Object} config the config object
6  */
7  /**
8  * @class Ejst.CustomWidget
9  * @extends CQ.form.CompositeField
10 * This is a custom widget based on {@link CQ.form.CompositeField}.
11 * @constructor
12 * Creates a new CustomWidget.
13 * @param {Object} config The config object
14 */
15 MyClientLib.CustomPathFieldWidget = CQ.Ext.extend(CQ.form.CompositeField, {
16
17 /**
18 * @private
19 * @type CQ.Ext.form.TextField
20 */
21 hiddenField: null,
22
23 /**
24 * @private
25 * @type CQ.Ext.form.TextField
26 */
27 linkText: null,
28
29 /**
30 * @private
31 * @type CQ.Ext.form.TextField
32 */
```

```
linkURL: null,

/**
* @private
* @type CQ.Ext.form.CheckBox
*/
openInNewWindow: null,

/**
* @private
* @type CQ.Ext.form.FormPanel
*/
formPanel: null,

constructor: function (config) {
config = config || {};
var defaults = {
“border”: true,
“labelWidth”: 75,
“layout”: “form”
//”columns”:6
};
config = CQ.Util.applyDefaults(config, defaults);
MyClientLib.CustomPathFieldWidget.superclass.constructor.call(this, config);
},

//overriding CQ.Ext.Component#initComponent
initComponent: function () {
MyClientLib.CustomPathFieldWidget.superclass.initComponent.call(this);

// Hidden field
this.hiddenField = new CQ.Ext.form.Hidden({
name: this.name
});
this.add(this.hiddenField);

// Link text
this.add(new CQ.Ext.form.Label({
cls: “customwidget-label”,
text: “Link Text”
}));
this.linkText = new CQ.Ext.form.TextField({
cls: “customwidget-1”,
fieldLabel: “Link Text: “,
maxLength: 80,
maxLengthText: “A maximum of 80 characters”,
allowBlank: true,
listeners: {
change: {
scope: this,
fn: this.updateHidden
}
}
});
this.add(this.linkText);

// Link URL
this.add(new CQ.Ext.form.Label({
cls: “customwidget-label”,
text: “Link URL”
}));
this.linkURL = new CQ.form.PathField({
cls: “customwidget-2”,
fieldLabel: “Link URL: “,
allowBlank: false,
width: 225,
listeners: {
change: {
scope: this,
fn: this.updateHidden
},
dialogclose: {
scope: this,
fn: this.updateHidden
}
}
});
this.add(this.linkURL);

// Link openInNewWindow
this.openInNewWindow = new CQ.Ext.form.Checkbox({
cls: “customwidget-3”,
boxLabel: “New window”,
listeners: {
```

```
change: {
scope: this,
fn: this.updateHidden
},
check: {
scope: this,
fn: this.updateHidden
}
}
});
this.add(this.openInNewWindow);

},

processInit: function (path, record) {
this.linkText.processInit(path, record);
this.linkURL.processInit(path, record);
this.openInNewWindow.processInit(path, record);
},

setValue: function (value) {
var link = JSON.parse(value);
this.linkText.setValue(link.text);
this.linkURL.setValue(link.url);
this.openInNewWindow.setValue(link.openInNewWindow);
this.hiddenField.setValue(value);
},

getValue: function () {
return this.getRawValue();
},

getRawValue: function () {
var link = {
“url”: this.linkURL.getValue(),
“text”: this.linkText.getValue(),
“openInNewWindow”: this.openInNewWindow.getValue()
};
return JSON.stringify(link);
},

updateHidden: function () {
this.hiddenField.setValue(this.getValue());
}
});

CQ.Ext.reg(‘mypathfield’, MyClientLib.CustomPathFieldWidget);
```

4. Use the custom `xtype` in your component. Add the following sample code to the `dialog.xml` file for the component:

```
< linkspanel
jcr:primaryType="cq:Panel"
border="false"
height=""
title="Links"
width="">
< items jcr:primaryType="cq:WidgetCollection">
< links
jcr:primaryType="cq:Widget"
fieldDescription="Press + to add more links"
fieldLabel="Links"
hideLabel="true"
name="./links"
width="1000"
xtype="multifield">
< fieldConfig
jcr:primaryType="cq:Widget"
xtype="mypathfield"/>
< listeners
jcr:primaryType="nt:unstructured" />
< /links>
< /items>
< /linkspanel>
```

See also

To the top

- CQ Help and Support Resources
- Other CQ best practice articles

- "Components", "Default components", "Developing components", and "<cq:includeClientLib>" in CQ documentation
- "Expression Language" in JSP documentation
