Next generation RPG documentation

Learn how to build browser applications using RPG and ExtJS

Aaron Bartell April 12, 2011

The RPG language has many aspects to it that have survived the test of time, like it's ease of database access and simple modularity structure. But other areas of RPG have lagged behind, like the user interfacing layer and a modern approach to community documentation. This article will describe how you can use RPG to build a modern interface that addresses the community supplied documentation need by combining new browser framework technology (for example, ExtJS) and existing back-end server software (such as, Apache, RPG and DB2 for i).

What's best for the RPG shop?

I am a big advocate for shops to make the most of their existing investment in technology. That means, if I consult with an RPG shop on how to modernize their application user interfaces, then I don't try to move them to Microsoft®, PHP or Java™ unless it makes solid business sense. Most times, it makes more sense to stay with RPG. A common issue I come across is lack of being able to hire new personnel. If you are short on RPG programmers, then you can follow the lead of Jorge Gutierrez in this IBM i PowerUp blog where he describes the process they go through to train Java or C++ programmers into RPG programmers in 2 months time. But lack of available RPG personnel is only one of the issues plaguing RPG.

A history

A few years back I started to take inventory about all the positives and negatives with RPG, and at the end of the day I came to the conclusion that what RPG developers lack is an easy way to talk to a modern user interface. We already have second-to-none integration with the solid DB2 database, excellent stability, and runtime with the operating system. RPG has easy syntax with simple modularity and wait-based technologies (like, data queues and library lists) to control environments and an easy to use command line with built-in help and prompting. Many of these things are huge positives and are unique to our platform. But again, we lack an adequate modern user interface.

I wasn't the first person to make the statement that we needed a modern user interface for RPG, but I also haven't seen a solution that really works well. I have used the likes of CGIDEV2 and other template based engines for many years only to find that they still required me to do a lot of work. I wanted the same ease of screen configuration that DDS and 5250 offered. Then in

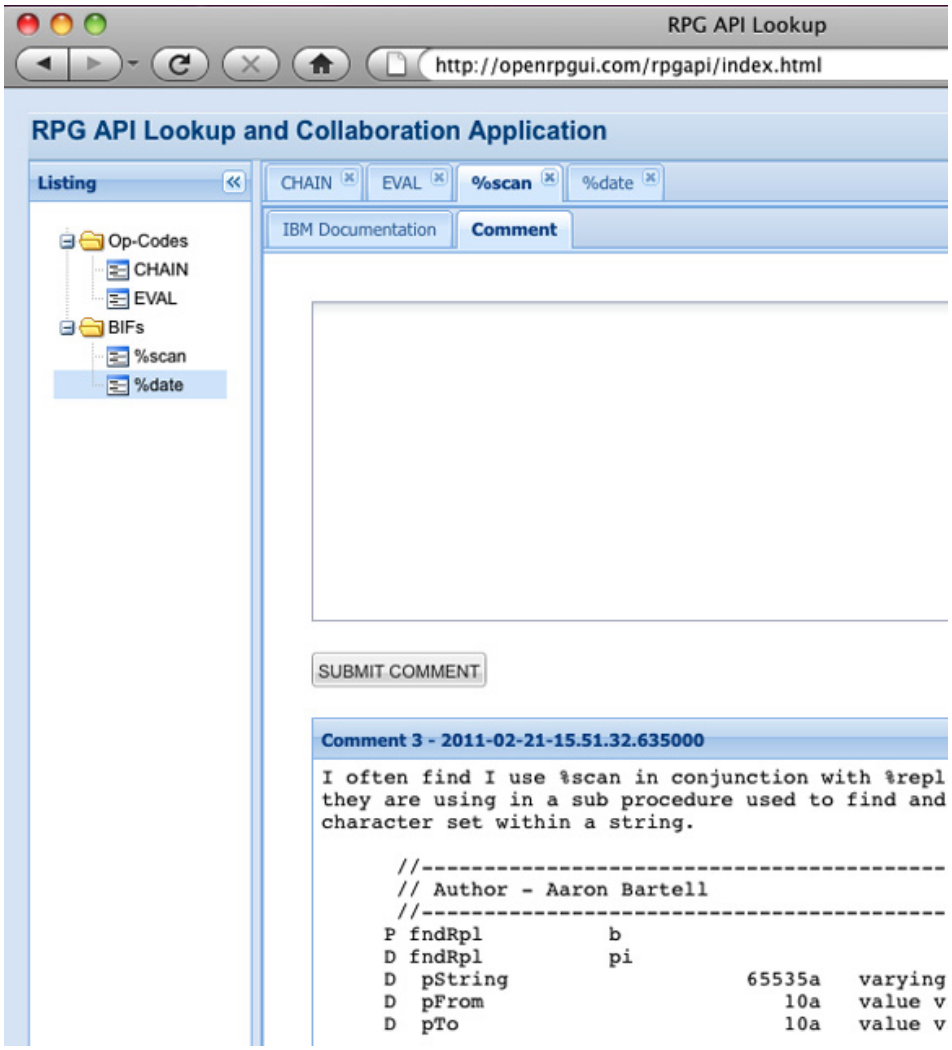
early 2009, I came across a match made in heaven—I found ExtJS, an open source JavaScript framework that has many pre-built UI components. I also found Michael Schmidt's open source JSON service program which then allowed me to process JSON strings directly in RPG with ease. From both of those technologies, the OpenRPGUI tool was born. Ever since then, I have been working to further the patterns and practices of using OpenRPGUI to interface, not only with the browser, but also with mobile platforms like Android. See the Resources section for more information about OpenRPGUI.

Setting the stage

Back when I was first learning how to program, it would always bother me when real-world examples were not used to convey how technology works. For example, while learning Java's OO concepts with a base class of "Animal" and extending it to get a "Dog" or "Cat" class, I questioned whether that example would ever be relevant in the business world. With that said, we are going to build a web application that will hopefully be used daily in the life of an RPG programmer. It is an RPG API Lookup and Collaboration application that is similar to what the www.PHP.net website has offered it's community of programmers for many years. Figure 1 below shows an overview of the entire application which can be accessed live at http://OpenRPGUI.com/rpgapi. In the far left navigation, we see a tree that contains two folders—Op-Codes and BIFs—and within there we see two corresponding entries. For the purposes of this example, I imported only four APIs because I want to know if this is something the community will use before I put more time into it. Please let me know if you have additional ideas for features, and I will do my best to add them!

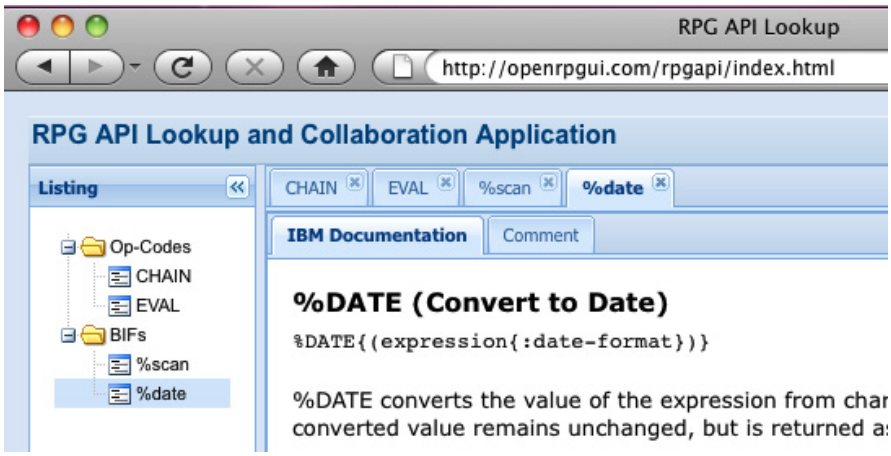
When you click on an Op-Code or BIF it opens a new named tab in the center of the screen (such as, '%scan') with another sub tab panel within it containing the formal IBM documentation and a place for the community to enter their own comments. You can have many Op-Codes or BIFs open at the same time, which makes it easy to navigate back and forth without having to traverse the tree again. If you choose to leave a comment, it shows up below the text area on the Comment tab. This is handy because sometimes there are a number of best practices that senior developers have adopted that would be useful for beginners to know.

Figure 1. Main application overview



In Figure 2 below, you can see the formal IBM documentation. This is stored in a static HTML file on the local web server. I am debating whether to have it out the IBM website and render their content into this tab in dynamic fashion. The danger in doing that is slow load times, and also if their website URLs ever changes, then it would break this application. For now, we will just go the route of local static HTML documents.

Figure 2. IBM documentation tab



Now that we have seen the visual representation of the application, it is time to dive into the code. There are two primary files you need to be concerned with and those are the index.html file and RPG program RPGAPI. The index.html file contains the screen definitions for the web application much like how a source member on the IBM i can contain DDS that describes a 5250 green screen. This is one of my favorite things about ExtJS—the fact that I can `define` and `configure` my screens versus mashing together a whole bunch of HTML tags and hoping for the best. The RPGAPI program's job is to act as the controller, do business logic, and access the database. The ExtJS screen definitions have mechanisms to communicate with RPG on the server by using HTTP and passing what are called JSON strings. The good thing about this model is it is in alignment with the statement made earlier that we already have an excellent platform, language and database—we just need a new UI.

Now, we will dig into some ExtJS definitions as found in the index.html file. At the very top of index.html, we see the inclusion of ext-all.css, ext-base.js and ext-all.js. These are the foundational text files that give us access to various pre-built UI components, many of which we saw in Figure 1. The utility.js file has commonly used functions that we reference later. The Ext.ux.AlertBox.js and Ext.ux.AlertBox.css relate to a user defined extension that I found on the Internet. We use it to notify the user when they submit a new comment for an API.

Loaded and ready to configure

Now that we have all of the right resources included in our index.html file, we can start defining the user interface. I describe the user interface by first starting at the outer most portions and then describing the child components. ExtJS has the capability to have many parent-child relationships. For example, I can have a panel within a tabbed panel, that is yet again within another tabbed panel. You could relate this to RPG DDS screen design where we have the ability to have multiple records and fields within those records—though a single tier is as deep as we can go. The shell of the entire web application is held and defined in what is called an Ext.Viewport, which is used at the bottom of index.html. The `viewport` variable is defined to have multiple regions, and in this case we have north, west, and center regions. The syntax used to define the `viewport` is standard JavaScript. You can read up on how JavaScript objects and arrays work because ExtJS uses them extensively. As you read that documentation, you'll discover the incredible power and flexibility of JavaScript. But, with power tools comes great danger. I am missing three fingers

because of JavaScript—metaphorically speaking of course. Configuring an ExtJS entity is all about specifying a variety of simple, or complex, name-value pairs. For example, the first config option for `viewport` is `layout` with a value of `'border'`. There are many different ways to configure a single UI component in ExtJS. The best way to see the available options is to head over to the ExtJS online documentation.

The `viewport` holds reference to two variables we are concerned with: `tpAPI`—an Ext.TreePanel definition and `tabsMain`—an Ext.TabPanel. The `tabsMain` definition is completely empty of an `items` array and initially holds no other components. That changes once the user clicks on one of the tree entries that are defined in `tpAPI`.

The `tpAPI` TreePanel is where all of the action happens. You notice there are many layers of embedding objects within objects and arrays within arrays. An alternate approach would be to define each entity separately in a declared variable using the JavaScript `var` keyword, similar to `viewport` where the `tpAPI` definition is referenced. For example, here's how you can do deep embedded declarations—pick the solution that is most legible and easy to maintain. At a high level, the `tpAPI` TreePanel has two areas of interest: the `root` config option—where we define the parent child relationships of the tree and the `listener` config option—where we define the user interactions we wish to listen for and act on. In the `root` config option, you can see I have more or less hard coded the Op-Codes and BIFs into the `children` array of the Ext.tree.AsyncTreeNode. I should note that before this article, I hadn't yet used the ExtJS tree features, nor did I have any idea how to implement it. With that said, I googled my need to produce a tree-like structure and that turned up an example in the ExtJS community forums. Never start from scratch with anything concerning ExtJS. At this point it has been around for a good many years, and there are many thousands of developers using it and posting questions and solutions on the Internet.

Listen and act

Now let's move into the fun stuff within the `tpAPI's listener` config option. When the user clicks on a node within the tree, it invokes our in-line JavaScript function and passes the node that was clicked along with an event object; we are only interested in the node parameter. First, we check to see if the node clicked was a folder or a "leaf", because we only want to process leaf events. Next, we invoke `tabsMain.items.find()` to see if this node is actively open in the `tabsMain` tab panel, because if it is we do not want to open it again. If the `tab` variable was not occupied, then we proceed to invoke the `tabsMain.add()` method. Notice we are making calls to things like `new Ext.Panel` and `new Ext.TabPanel` without assigning them to a variable like we did with `tpAPI`. This is because we don't have need to reference the variable later on and instead are going to let the definition float up to it's parent, `tabsMain` in this case. Actually, that is partly a lie because you will notice that we are specifying an `id` config option on many of the Ext.Panel definitions. We do this so we can subsequently reference the panel to either add content to it or obtain form variable values for submission back to the server.

The next pertinent panel configuration is the `autoload` option. The curly brackets after `autoload` tell us we are configuring an additional object with it's own config options that will be placed into `autoload`. In this case, we are supplying a URL to an HTML file on the server that is relative to this directory. The `node.id` variable holds the name of the API we are wanting to retrieve from the

IFS on the IBM i. For example, if the `node.id` contained a value of 'chain', then a request would be made to the server as follows:

```
    http://openrpgui.com/rpgapi/chain.html
```

The chain.html file occupies the 'IBM Documentation' tab, and then the 'Comment' tab will be defined in the next portion of code where we also see a config option of `handler` with a value of `btnHandler`. This is how we capture the event when the user clicks the SUBMIT COMMENT button. If you scroll up in the index.html file, you will see the JavaScript function definition for `btnHandler`. The general gist of this function is that the Ext.Ajax.request is used to make an HTTP request to the server and include the forms that are specified on the `button's submitForms` config option. In this case, we are using the word "form" loosely because what we are actually doing is passing in the name of an Ext.Panel. The `getFormVars()` function will recursively iterate through that panel until all fields are accumulated into a name-value-pair string (i.e. TXT=some comment&TYP=chain). The `button.getId()` is used to relay to the server's RPG program what action has been taken on the client. After accumulating the request information, the Ext.Ajax.request invokes the program specified on the `url` config option, and if the HTTP communication is successful, it invokes the code that exists in the `success` config option. When the server responds successfully, we access the `rsp.responseText`, which currently holds the JSON response, and pass it to JavaScript function `uiactn` (defined in the utility.js file mentioned at the beginning of this article).

Going back down to the `listeners` config option within `tpAPI`, we see `jsCmt` being configured as an Ext.data.JsonStore. This JsonStore serves two purposes. First, it retrieves from the server a list of comments that have been entered for this particular API and returns them to the browser. Then, the `load` config is invoked, and based on the ExtJS documentation JsonStore, we can see that one of the parameters for this function is the current JsonStore record set. This means we can now use Ext.iterate to go through each JsonStore record and add a new panel for each comment that needs to be displayed back to the user. What's interesting about this bit of code is that normally I would have used Ext.GridPanel to display a list of data. However, I could not find any examples of how to accomplish that task, so I created what you see here. The last thing we do is invoke `tabsMain.setActiveTab()` to make sure the tab we just created is now in focus on the browser page.

Server side RPG code

Now that we have the client user interface components configured, we can move on to the server side that is supplying both the static HTML and the comment capabilities. On the server side, we are using Apache to facilitate the HTTP requests, and then Apache hands off the dynamic processing to RPG program RPGAPI.RPGLE. The RPGAPI program is making use of the open source and free OpenRPGUI framework to receive in requests and compose JSON string responses. I am going to make an assumption that you are versed in the RPG language and syntax, so I won't be covering those types of details. Instead, let's look at the general concepts of what is happening starting with the mainline of the program where sub routine `initpgm` is being invoked. Inside `initpgm`, we see that we are invoking `json_create()` and `jsona_create()`. The variable `gJRsp` is used to store the root of the response that is sent back

to the client that this program will be incrementally adding to by using other `json_*` APIs. Next we see `http_inStr()` being invoked to retrieve the request string from the browser (for example, TXT=some comment&TYP=chain&action=btnPstCmt). We then use `http_getCgiVal` to extract the `action` which is then evaluated back in the mainline to determine which local sub procedure should be invoked, in this case it is btnPstCmt.

In the btnPstCmt sub procedure, we can see that it is obtaining the next available unique key for a column in DB2 table COMMENT and then subsequently occupying the record with information from the HTTP request and finally writing the record. We then make a call to a locally defined sub procedure named `uiAlert.` Here is where I am introducing something new into the mix of how OpenRPGUI interacts with ExtJS. Up till now, I have been sending down JSON strings that state things like `{ show:'panelName'}`, and then on the client side in utility.js, I would iterate through the JSON string and use the Ext.getCmp() API to obtain dynamically the UI component and subsequently invoke it's `show()` function. That was all well and good, but I found I was constantly changing both the server and client side code to add new UI actions. It was then I remembered the JavaScript eval function. The `eval` function allows you to pass JavaScript statements to it that are invoked within the browser in dynamic fashion. This opens up many possibilities, because now we can simply create easy to call RPG sub procedures, like uiAlert, that dynamically string together a JavaScript statement and subsequently send it down to the client to be invoked. One possibility that I have been considering is storing the panel definitions in a DB2 table and having a single statement in the client code that simply invokes a URL (like an RPG program) and when that program is called the first time (for example, no parms sent) it responds with the first screen definition that should be displayed. Storing screen definitions in a database, where have I seen that done before? Oh yeah, that is how we do it with DDS. Maybe I will show how to do that in my next developerWorks article!

The other portion of RPGAPI that needs to be described is the `loadCmtList` sub procedure which is used to compose a list of comments to send back to the client. As you can see, we are first obtaining the `TYP` value from the HTTP request and using that to do an SQL query against the COMMENT table. The COMMENT table definition can be found at the very bottom of the RPGAPI.RPGLE.pdf document. The important part of this sub procedure that I need to call out is how JSON can be composed using the `json_*` APIs. In the `dou` loop, the `json_create()` API is first called to create a JSON object. Then we can add sub values to the object by calling `json_putInt` and `json_putString`. Once the object contains the values we want, it then needs to be added to the `ja` object which is a JSON array created earlier using `jsona_create()`. You can see the resulting JSON this produces in code sample below. Pay special attention to the `list` and `sql` JSON names. The `list` was created in the `dou` loop and the `sql` is purely for debugging and example purposes.

Listing 1: JSON code sample

```
{ "list" : [{ "CRTTS" : "2011-02-21-15.00.27.50400",
"TXT" : "/free <br/> <br/> chain (key1 : key2) table;<br/> i
"TYP" : "chain",
"UID" : 2
}],
"msg" : "",
"sql" : "SELECT UID,TYP,CRTTS,TXT FROM RPGAPI/COMMENT WHERE TYP='cha
"uiactn" : [ ]
}
```

The last processing that occurs with each request to the RPGAPI program is for sub routine `endpgm` to be invoked. In `endpgm`, we are finalizing the JSON to be placed in `gJRsp` and also sending the response back to the browser using `http_outStr` and `http_outPtr`. Note that `Content-Type` followed by two line feeds is necessary so we are compliant with the HTTP specification. The `Content-Type` is considered the HTTP header, and the two line feeds tell the browser where the content of the response starts. That's it for the server side!

Conclusion

The potential for RPG shops to succeed on the web is better than ever before because processing is quickly moving back up to the server versus being on the client. IBM is taking advantage of this by putting forth many cloud initiatives.