Jeff's Excruciatingly Simple Extension Tutorial for Fantasy Grounds (Part 1)

Jeffery W. Redding

8/22/2015

version 1.1

Table of Contents

Introduction....3
The (Very) Basics....3
1Anatomy of an Extension....3
2Directory Structure....3
3XML at the 50,000 ft. Level....4
Minimal Viable Product....4
1Creating Our Extension File....4
2TutorialWindow.xml....6
3Let's run it!....6
4Finding a better base....7
5Adding the close button....8
6Adding a title graphic....8
7Resize this Sucker!....9
Getting a little Lua going....10
1Lua are you?....10
2Adding some Lua script....10
3Adding a simple script in a windowclass....11
4Deeper into our script....12
4.1Debug.console("self", self)....12
4.2Debug.console("getClass", self.getClass())....13
4.3Debug.console("getDatabaseNode", self.getDatabaseNode())....13
4.4Debug.console("getFrame", self.getFrame())....13
5Time for the Meat!....13
5.1Finding our methods....14
5.2Finishing touches....15
That's a Wrap!....17
Appendix A: The Final Result....18

Introduction

Fantasy Grounds is a very powerful and full featured platform for building and running desktop RPG games. It also provides a lot of room for developing custom rulesets, extensions and so forth, but at a high price of complexity and lack of really good introductory material.

Given that, I decided to jump in and write an extension. Silly me.

Trying to build that extension took a lot of fits and starts, so I decided I would also create this tutorial in hopes that others would find this an easy way to get their feet wet in this realm. I'm going to start with the VERY minimum, and work up to something better (I hope!). Keep in mind as you read this that this is only one way to go about things. Once you have a basic understanding of how the FG engine works, and what some of the terminology means, then branching out and digging through the HUGE volume of the existing rulesets gets a lot easier.

Starting with an extension is probably the easiest way to go.

The (Very) Basics

1 Anatomy of an Extension

An extension adds or replaces functionality on an existing ruleset.

Adding is like building an addition to your house. You need to figure out where the addition attaches, how it's going to look, and what's going in it. You have to understand enough of your existing structure (i.e. the ruleset) to figure out how to shoehorn this new, alien thing into what's already there.

Replacing is more like converting your extra bedroom to a bathroom. You have to really understand how the house is built, how the room is built, what functions the room already has, what functions you're going to replace, etc. It can be quite daunting given that if you override something improperly, you might loose something you need!

For this demo, we're going to add functionality. In particular, we're going to add a new window, and put some stuff inside it.

In order to get started, FG only needs a few things from us. First, an extension.xml file, second a windowclass that will be the basis of our extension, and, of course, a place to hold our stuff

2 Directory Structure

For our basic extension, we're going to start with the MVP (minimal viable product). So let's start by doing the following:

- Create a new directory under your fantasy grounds extensions directory that will hold your new extension. In my case, my FG directory is “D:\Users\FantasyGrounds\extensions”, and I'm adding a folder called “ext-tutorial”.

While we're creating folders, we might as well create some additional folders under our new folder. These folders are not required. The main reason we want to create these folders is to make management of our project easier as we get more complex. Really complex extensions can have many folders, with many sub-folders. How you organize is up to you.

For our example, since I don't anticipate our tutorial will be monstrously large, I'm going to create

folders to hold XML files, Lua files, and graphic files. In effect, under the ext-tutorial folder, I'm going to create the following folders:

- xml
- lua
- graphics

3 XML at the 50,000 ft. Level

XML stands for eXtensible Markup Language. It's designed to allow people to create files with “tags” in them that are relatively easy for a person to read, and also easy for a machine to parse and use.

Tags are all the things you'll see that start with “<” followed by some identifier (like “properties”) followed by a closing “>”. XML Tags have both start and stop tags, with the stop tags denoted by “</” instead of just “<”. For example, “<properties>” as a start tag, with “</properties>” as the end tag. You can also have tags that have the start and end together in one tag like “<properties />”. XML is “well ordered” in that every start tag MUST have a close associated with it. In addition, tags are hierarchical which means that a tag that started inside a tag, must end inside the tag it started in.

We'll get plenty of examples, but if the paragraph above is too confusing, then you probably should take a little time to find an online XML tutorial and go through that. FG makes heavy use of XML, and understanding the basics can make working with the files we create much easier.

Minimal Viable Product

The intent here is to do the minimum needed to create and display a window in Fantasy Grounds. In future chapters, we'll add some features and data to grow our MVP to something useful.

The first thing we'll create is our extension xml file.

1 Creating Our Extension File

Our first XML file, and also the one that FG will see and use first is called “extension.xml”. We create it in our extensions folder (not in one of the sub-folders), and it must be called “extension.xml”.

You can use whatever text editor you like, though I recommend you use one that understands XML since it will help you find missing close tags and syntax errors. Lots of folks use Notepad++, though I personally prefer something more technical like Eclipse or IntelliJ, or the like. Another good alternative to Notepad++ is the Atom editor.

Here is what goes in our most basic extension.xml file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- This is our MVP for the Extension Tutorial -->
<root release="3.0" version="3">
     <properties>
          <name>Extension Tutorial - Jeffery W. Redding</name>
          <version>0.0.1</version>
          <author>Jeffery W. Redding</author>
          <description>
               A really simple  extension file for
               creating a really simple extension
          </description>
     </properties>
     <base>
          <includefile source="xml/TutorialWindow.xml" />
     </base>
</root>
```

Now let's digest this a bit.

The first line is a special line. This line actually breaks the XML convention about having a close tag. Instead, it starts with a "<?" and closes with a "?>". This tag is used by the XML parser to understand what character set and version this file is. Once you cut and paste this line, you'll probably never change it. You'll also see this line in every XML file you create.

The next line is special, too. It's called a comment. Using the start tag "<!--" and the close tag "-->" allows us to add comments within the XML file that the machine reading the file will ignore. This is really important for us humans looking through these files and trying to understand how they are used. It will be the single most frustrating part of looking through the core files later. They are nearly comment free, so understanding them is MUCH harder.

On the other hand, my stuff probably has too many comments. It's the only way I can remember why I did what I did. Moving on.

Now we start to see normal tags. As you look through this, you'll see each one is properly closed, either with a "</" close, or with a "/>" at the end of the tag. One quick note: Within FG and a lot of the documentation, you will hear each "tag" referred to as a "node". Most of the XML spec refers to "tags", while other specs refer to "nodes". No big deal, just something to be aware of.

Lets walk through each tag and it's purpose:

- **root**: This tag is the base, or outermost, tag for every XML file used by FG. The name/value pairs inside the tag are called "attributes". It's REALLY important to get the right release and version in this tag! This is NOT the release and version of your extension, but rather the release and version of FG that you've built your extension for.
- **properties**: This tag contains some high-level descriptive information about our extension.
  - **name**: This is the name of our extension. You'll notice the text for our extension name is in between the start and end tags for the name, rather than in attributes within the tag itself (like in root). This is because the title is the actual value for the name tag, whereas attributes are modifications to the tag. This isn't always clear, and it's not always consistent, since it is dependent on the person who wrote the definition for the xml file.

- **version**: This holds the version that we want associated with our extension. Like name, it has it's close tag on the same line as the open tag, with only a short text snippet in its contents.
- **author**: That's me! (Or you, if you're writing your own extension)
- **description**: That is a text description of our extension. You'll notice that the text is over multiple lines. It could have been one (like name), but the key is the the XML parser doesn't care. It will remove the line breaks and tabs, and push it into one line anyway when it all gets parsed. That's important to note if you want text to appear a particular way, it won't without special steps.

- **base**: Finally! Something more interesting. The base tag is where we start to do something that FG will really pay attention to. This is the tag that tells FG where scripts, windows and graphics start to come into play.
  - **includefile**: This does exactly what it says. It allows use to include a file into our 'base' that will be parsed by FG for our extension. We'll add more of these includefile tags as we go along, but for now, we're only including the XML file for our MVP tutorial window. You'll notice, that when I specified the value for the source attribute, I made sure to tell FG that it was in the subfolder called 'xml'. You'll also notice that this tag makes use of the alternate close syntax "/>" since it doesn't have any real contents.

2 TutorialWindow.xml

No we start to get to some harder stuff. First, let's create a file called TutorialWindow.xml in our XML sub-folder. Here is the basic XML that we'll start with:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<root>
	<windowclass name="TutorialWindow">
		<frame>utilitybox</frame>
	</windowclass>
</root>
```

Believe it or not, this is all that's needed in order to create a brand-spankin' new window in FG. You can see the standard xml and root tags, and then we have the following:

- windowclass: This is the name of our newly created window
- frame: This is the name of the frame for our new window.

For this tutorial, I dug around in the coreRPG files and found a frame definition for "utilitybox". A frame definition provides the "frame" in which we add window classes in order to build up our window. There are 40+ framedefs within the core packages, but most of them had names that didn't sound like something we wanted to use, I took a guess at the frame we want, and we'll be able to see if it was a good guess quite soon.

3 Let's run it!

Fire up FG, and either load an existing campaign or create a new one. You should see our extension tutorial listed in "Campaign Detail". Make sure and click that so FG will load it. Then hit "Start"

At this point, nothing should be different. You see, we've created the window, but we haven't attached it any where so that you can click to see it. That's an issue, but minor for now.

To see our new window, go to the chat window and type '/openwindow TutorialWindow' and hit enter. You should now see our glorious new window! You should also notice some immediate problems:

- No way to close the window
- No way to resize the window
- No label telling us what this window is

You'll also note that the window has borders, background, titlebar etc. that we didn't specify. That's because it all came from the frame we chose; utilitybox.

4 Finding a better base

After our inspection, I'm not really sure that's what I want for my starting window. I really like the "Notes" frame better. At this point, I'm not 100% sure what frame it uses. As a simple test, I can just change the frame name in our xml to some other framedef that I found. That can be a little too random, so I'm going to do a little research.

First, within the RPGCore/graphics/frames folder I can see the various graphics that make up existing windows. By looking at these, I find that "storybox" looks like the background used for Notes, and looks like a good fit for my purposes. Next, I use my editor to search for "storybox" in all the XML files. This leads me to a framedef "storybox", and I find the campaign_notes template as well.

Let's change our "utilitybox" to "storybox". Save the file. Then, in FG go to the chat window and type "/reload ruleset". This will reload all the rulesets without having to exit FG and come back in. It's not as quick as I'd like, but it's better than going all the way out.

A quick note as it's reloading. Rather than continually typing "/openwindow TutorialWindow" and "/reload ruleset", type them into the chat window BUT DON'T HIT ENTER. You can then drag and drop the command onto an open slot in your command bar. Then just click that button to run the command.

So, now we've reloaded the ruleset and opened the window we can see it's closer to the Notes box, but still not the same. In fact, it's not the main note box, but the box you get when you want to add a note. Not quite what we're looking for. As part of our 'storybox' search, I noticed a file called campaign_notes.xml. That sounds pretty promising, so I brought that up in my editor and took a look through the contents.

This file has several windowclasses in it: note, note_header, notesmall and notelist. The last one looks the most promising, and looking into it we can see that it uses "campaignlistwithtabs" as it's frame, so we're going to change our frame to use that:

```
<frame>campaignlistwithtabs</frame>
```

Save, reload and re-open. You'll do that a lot.

Success! We haven't fixed our earlier problems, but now we have a window that is a good base for what I want.

5 Adding the close button

Looking at the "notelist" windowclass, we can see it has some stuff we don't need (at least yet). Some

of this is pretty obvious, some of it is more obtuse. We can start by looking at “windowclass” in the FG reference docs (do a Google search for “Fantasygrounds windowclass” to find the URL). Unfortunately, none of that seems to be what we need to be able to close our window.

Looking closer at notelist, we can see within the sheetdata tag a couple interesting tags: resize_campaignlistwithtabs and close_campaignlist. We can search for those with our editor to see what they do.

Looking into close_campaignlist, we find it has a “close” tag, with an anchored tag. The anchored tag tells us where the “close” is anchored on the frame or window that its in. Even more digging reveals that close is also a template, and it has what we're looking for! It has a close button, and a lua script function that closes the window. Awesome! Since the work has been done for us up in the close_campaignlist, that's what we're going to add to our windowclass.

Our windowclass should now look like:

```
<windowclass name="TutorialWindow">
	<frame>campaignlistwithtabs</frame>
	<sheetdata>
		<close_campaignlist />
	</sheetdata>
</windowclass>
```

You'll notice we had to add sheetdata to our windowclass as well. That's because we want the delete button and script to be part of our window. By placing it in the sheetdata section, we're telling FG that this is part of what we're displaying and acting on. You can try it without being in sheetdata and you'll see it has no effect. FG doesn't attach it the way we want so we can't use it. Sheetdata must be in place.

Now, if you do the usual SRO (Save file, reload rulset, open window), you'll see we have a nice window that we can close! Go ahead and test the close by clicking on the 'X', and then opening the window again.

6 Adding a title graphic

Next let's go ahead and tackle our title problem. If we look at campaign_notes again, we can see that the 'banner_campaign' tag has an icon called 'title_notes'. If we search for title_notes, we'll find that it is a png file that is specified in the graphics_icons.xml file.

We don't need to go clear out to the graphics_icons file. Instead, we're going to add the icon line inside our extension 'root', and then include the icon inside the banner_campaign inside the sheetdata. When done, it should look something like this:

```
<root>
	<icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
	<windowclass name="TutorialWindow">
		<frame>campaignlistwithtabs</frame>
		<sheetdata>
			<banner_campaign>
				<icon>TutorialTitle</icon>
			</banner_campaign>
			<close_campaignlist />
		</sheetdata>
	</windowclass>
</root>
```

You'll notice that I've specified a file in the graphics sub-folder of our extension. I created that myself using Gimp. This isn't a graphics tutorial, so you're on your own for that part. :-) Here is what I used:

One thing you'll see is that this graphic is oriented vertically, which makes sense because that's what the CoreRPG ruleset is expecting. In CoreRPG and other rulesets, the graphic for the title is vertical, and along the left edge of the window. If we try to use our new extension in 5E, our title won't work since the title location is horizontal and along the top! We'll cover some code later to detect which is which, and switch the graphic as needed. (Chapter 3)

When you SRO again, we now have a nice window with a title and a close button.

7 Resize this Sucker!

For resizing, we repeat what we did for close, and find a tag called “resize_campaignlistwithtabs”. We add that to our windowclass just like we did for the close and retest:

```
<root>
	<icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
	<windowclass name="TutorialWindow">
		<frame>campaignlistwithtabs</frame>
		<sheetdata>
			<banner_campaign>
				<icon>TutorialTitle</icon>
			</banner_campaign>
			<close_campaignlist />
				<resize_campaignlistwithtabs />
		</sheetdata>
	</windowclass>
</root>
```

Voila! The resize icon is now visible in the lower right corner. Which is nice, but it still doesn't resize.

After digging into resize_campaignlistwithtabs, one thing we can see is that unlike close_campaignlist, there isn't any lua script that makes the window resize. This is because resize is controlled by a special tag that we need to add in our window class. Let's add the needed tags now.

```
<root>
	<icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
	<windowclass name="TutorialWindow">
		<frame>campaignlistwithtabs</frame>
		<sizelimits>
			<dynamic />
		</sizelimits>
		<sheetdata>
			<banner_campaign>
				<icon>TutorialTitle</icon>
			</banner_campaign>
			<close_campaignlist />
				<resize_campaignlistwithtabs />
		</sheetdata>
	</windowclass>
</root>
```

As you can see, we needed to add the “sizelimits” tag so that we could add the “dynamic” tag underneath it. This tells FG that we want the windowclass we've just created to be resizable. When we SRO this, we now see the pointer change to a resize pointer when we hover over the sides, or the resize icon.

Getting a little Lua going

Earlier, I told you about a problem with our extension when we go from CoreRPG rulest to the 5E ruleset – Our title isn't oriented effectively for the changed ruleset. We're going to fix that by diving into a little bit of Lua script.

1 Lua are you?

Lua is a programming language similar to many, many others. If you're not a programmer, or don't play one on TV, then this is going to get mighty tough. If you start feeling lost, go find a beginners tutorial for lua and learn the basic programming concepts. Then come back and carry on.

2 Adding some Lua script

The first thing we're going to do is add a little lua script. The first thing we have to think about is WHERE we want to add the script. Based on the docs, we can:

- Add a script within a windowclass
- Add a script within a control
- We can create a standalone lua script file that we can include in place
- We can create a standalone lua script that we use as a package.

Awesome. So what does that mean, and why is it important to us?

It means that you can add lua scripting functions in very specific ways, gaining access to pieces of your window to do some complicated things like mathy calculations, database manipulation, window

trickery and other assorted tomfoolery.

It also means you need to understand some coding principles like scope, objects, functions, references, etc. I'll try to explain as we go along, without getting too bogged down in things like accuracy...

3 Adding a simple script in a windowclass

Let's start here, because it's easy, and lets us get our feet wet. Here is our familiar windowclass with a bit of lua script added:

```
<root>
    <icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
    <windowclass name="TutorialWindow">
        <frame>campaignlistwithtabs</frame>
        <sizelimits>
            <dynamic />
        </sizelimits>
        <script>
            function onInit()
                --[[ Self points to a windowinstance, in this case. ]]
                Debug.console("self", self)
                Debug.console("getclass", self.getClass())
                Debug.console("getDatabaseNode", self.getDatabaseNode())
                Debug.console("getFrame", self.getFrame())
            end
        </script>
        <sheetdata>
             <banner_campaign>
                <icon>TutorialTitle</icon>
            </banner_campaign>
            <close_campaignlist />
            <resize_campaignlistwithtabs />
        </sheetdata>
    </windowclass>
</root>
```

- ***Script Tag -*** The first thing you'll notice is that we've used XML tags to identify that we're including some lua script. Later, we'll see different ways to use the script tag.
- ***OnInit() -*** Inside the script tag, we encounter our first bit of Lua. In this case, we're creating a new function called "onInit". The keyword "function" identifies to the lua processor that it's a function, and the "()" tells it that it doesn't take any arguments. Functions are just convenient ways to group logical blocks of code together to make everything cleaner, and also to allow us to reuse the same function. If your confused here, go find an introduction to lua tutorial!

  This new function is a special function. If we have an onInit function defined, then this function is called by FG when the window instance is created, after the onInit functions of child controls have been called, but before the window is first displayed. Let's not worry about child controls just yet. The important thing to know is that this function will get called when we try to open our window.

- ***Comments -*** The next line is a Lua comment. If you remember from earlier, XML comments start with "<!--" and end with "-->". Lua comments come in two forms:
  - Wherever there is a "--" then lua assumes comment to the end of line
  - Wherever there is a "--[[" lua assumes everything is a comment until "]]"

  Easy enough, but there is a catch! When doing Lua inline (like in our example), the "--" comment doesn't work. This is because of the way the XML gets processed it ignores end-of-line. You must use the block form in order to put comments in your Lua code.

  In addition, once you're inside the script tag, you can't use XML comments anymore! You've left that world and know you're in Lua land!
- ***Console Logging -*** The remaining lines output some information to the FG console. You'll find the console to be your friend when you're tearing your hair out trying to figure out what is supposed to do what. To open the console, type "/console" in the chat window. This is another good string to drag and drop onto your command short cuts. You'll use it a ton.

  Note that the console function (just like onInit, console is a function) has that "Debug." prefix. That's because the console function is part of the "Debug" package. In order for our bit of code to call the console function, we have to tell FG where to find the function. That's why we have that prefix.

For now, go ahead and SRO. Then, once the extension is reloaded, open the console and then open our window. You should see several lines displayed corresponding to our statements. Next, we'll look at what those statements mean.

4 Deeper into our script

First, you'll see that unlike our onInit function, there is stuff inside the parenthesis. These things, separated by commas are called "arguments". They provide extra information to the function for whatever the function is trying to accomplish. In most cases, you have to make sure to give the right number of arguments, with the right type of arguments, and in the right order. In the case of console(), it can take variable arguments, in various orders, and can figure out the type of the argument which makes our lives easier.

If you compare the log to our script, it's pretty easy to see that the first argument we passed to console() was a string. Console displays the string exactly as it appears within the quotes.

4.1 Debug.console("self", self)

The first argument here, as in all the rest is simply a string that we want to display on the console. As you progress, you'll find that putting a sensible string in is REALLY important for debugging – especially when there are hundreds of messages popping up in the console! One thing that I suggest when your logging gets large, is to include the file name, class and the function in the log message. In other words, we could have used:

- Debug.console("ExtTutorial.xml:TutorialWindow:onInit:self(windowinstance)", self)

Which brings us to the next argument, self.

Self is a special argument. It points to the thing (object) that owns this function. In essence, in this case self refers to the instance of the windowclass that we put the script in. The console tells us we're dealing with a windowinstance object, and it also lists some attributes of that windowinstance:

- **class**: The name of the class. Not surprisingly, it's TutorialWindow because that's the name we provided when we defined the windowclass in our XML.
- **node**: This is the database node associated with this class. Since we haven't gotten so fancy, it shows Lua's 'nil' value which means none, empty, or null.
- **x,y,w,h**: Yes, Virginia, this means exactly what you think. This is the (x,y) coordinates of the window (i.e. the position in the FG desktop), and the width and height of the window.

4.2 Debug.console("getClass", self.getClass())

If you pull up the FG reference documentation, you'll find windowinstance on the left hand side. When you click on that, it pulls up the documentation for windowinstance, including a list of methods (functions) that a windowinstance has. GetClass() is one of the functions listed. Since we know we have a windowinstance, we know we can call it's getClass() function.

Looking at our output, you'll see exactly what we expected: “TutorialWindow” – the same thing it already told us in the output from printing self.

4.3 Debug.console("getDatabaseNode", self.getDatabaseNode())

No big surprises or revelations. This attempts to get the database node assoicated with this class. As we already know, this should be 'nil', and much to our relief, it is. According to the documents, this means our window is unbound. The only thing that means to us, is that we can only have one instance of our window up since there's no database node to make it unique.

4.4 Debug.console("getFrame", self.getFrame())

This function returns the name of the frame that our windowclass is using. Again, no big whoop. We knew what it was because we're the ones who set it. That's not always the case, so sometimes in can be good to be able to have FG give us the 411 on stuff like that.

5 Time for the Meat!

If you remember from the beginning of this chapter (Yes, it was a really, really long time ago), our intent is to fix the way our title shows up in 5E vs. other rulesets. Based on what we know so far, that means we need to change the banner_campaign, since it contains the icon we want to change. Let's start out by adding a script in the banner, with an onInit() function.

```
...
        <banner_campaign>
            <script>
                function onInit()
                    Debug.console(“self”, self)
                end
            </script>
            <icon>TutorialTitle</icon>
        </banner_campaign>
...
```

This time when we SRO, you'll notice we get this console line printed before our windowclass console lines. This is because the onInit script for the child nodes runs before the onInit for the parent node. That allows the children to complete their initiation tasks before the parents have to do the same. That way, if the parent relies on the child to be ready, everything works out.

Of particular interest is that 'self' in this context refers to a windowcontrol, rather than a windowclass. This makes sense since the contents of sheetdata is intended to be controls. Self doesn't have as many attributes, either. Just x,y,w,h.

5.1 Finding our methods

Looking through the reference docs, we can see that windowcontrols have quite a number of methods, but nothing that looks like it will help us set the icon, or even figure out what ruleset is being used. Time to dig deeper.

If we use our editor to search the RPGCore ruleset for banner_campaign (just like we searched for some other stuff earlier), we can find the template definition in template_campaign.xml. Looking at the template, we can see that banner_campaign is really a genericcontrol, which is a more specific kind of windowcontrol. Switching to the FG reference docs and looking up genericcontrol, we find a couple promising items! The functions hasIcon() and setIcon().

We're still missing anything to find the ruleset.

Ruleset seems like a pretty big deal. If you think about it, it probably doesn't make sense to find the ruleset as a method off of some window or object. This is more likely to be something we can find off one of the global packages. Heading back to the reference page, we can see the packages listed on the top left of the page. Some of those we can eliminate by name, some we can't. I found what I was looking for by sifting through a bunch of function calls, but I'll save you the trouble and tell you to pull up the User package, and look down the list. There is our quarry – getRulesetName()!

Let's put it together in some Lua code:

```
...
<banner_campaign>
    <script>
        function onInit()
            Debug.console("self", self);
            if(self.hasIcon() == true) then
                local rset = User.getRulesetName();
                Debug.console("rulesetName", rset);
                if(rset == "5E") then
                    self.setIcon("TutorialTitle5E", true);
                else
                    self.setIcon("TutorialTitle", true);
                end
            end
        end
    </script>
    <icon>TutorialTitle</icon>
</banner_campaign>
...
```

Welcome to the big time! Now we have some real programmin' shiznit goin' on in the hiz'ouse! Here, you can see we made use of our newly discovered methods; hasIcon(), setIcon() and getRulesetName(). In addition, we've added some 'conditional' statements. Conditional statements are particular kinds of coding statements that make our programs able to make decisions. As you can see here, the first statement, if(self.hasIcon() == true), is looking to see if self has an Icon. If it does, it will execute the code underneath it (before it's matching 'end' statement). If it doesn't then it will skip to its matching end and continue with the script from there. The next conditional statement is checking to see what the ruleset name is, and then uses the setIcon method to set to one of those two icons.

If you SRO at this point, it isn't going to work right. We need to make a change to add the new icon, "TutorialTitle5E". First, of course you have to make your graphic, or find one (not a photoshop course, remember?), and here is what I used:

Next, we could add another icon tag to our TutorialWindow.xml file, but it's already starting to get bigger. So let's move the icon tag from TutorialWindow.xml and put it in extension.xml and add the new one like so:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- This is our MVP for the Extension Tutorial -->
<root release="3.0" version="3">
     <properties>
          <name>Extension Tutorial - Jeffery W. Redding</name>
          <version>0.0.1</version>
          <author>Jeffery W. Redding</author>
          <description>
               A really simple  extension file for
               creating a really simple extension
          </description>
     </properties>
     <base>
        <icon file="graphics/TutorialTitle5E.png" name="TutorialTitle5E" />
        <icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
         <includefile source="xml/TutorialWindow.xml" />
     </base>
</root>
```

Now, if you SRO everything should look good. If you exit back to the launcher, and create a 5E campaign with this extension, the title will now open in accordance with that ruleset!

5.2 Finishing touches

At this point, we've done what we set out to do, but there's one more thing I want to do as kind of a housekeeping/cleanup step and that's to pull some of the Lua code out of our XML, and into our extension package.

So first, we're going to create a file called ExtensionTutorial.lua in our extensions' lua directory. For the contents, we're going to make a new function called “setIconByRuleset” and take our code out of our XML file and put it in this function.

```
-- ---------------------------------------------------------------------
-- setIconByRuleset - This function takes a windowcontrol for an
--                    argument and, if that windowcontrol has an icon
--                    it sets that icon to be one of two icons, based
--                    on the ruleset.
-- ---------------------------------------------------------------------
function setIconByRuleset(wc)
    if(wc.hasIcon() == false) then
        Debug.console("setIcon - no icon", wc)
        return
    end

    local rset = User.getRulesetName()
    Debug.console("setIconbyRuleset rulsetName", rset)
    if(rset == "5E") then
        wc.setIcon("TutorialTitle5E", true)
    else
        wc.setIcon("TutorialTitle", true)
    end
end
```

There are a couple of things to note. First, notice that since this is a lua script, and not and XML file, I can use the lua '–' comments that I couldn't use before.

Next, note that the function takes an argument, wc. This is because by taking the code out of the XML, it no longer has 'self' in scope. We'll have to pass the window control when we call this, which we will do in just a minute.

I also changed the first conditional to look for cases where the windowcontrol doesn't have an icon. This gets rid of some of the indention, and allows us to 'return' from this function immediately if there is no icon.

You may have noticed the use of the word local, both in the script in the XML, and in our new function. This is used to tell the Lua interpreter that we're creating a variable, in this case rset, but we don't want it to be visible or even persist outside of the scope we've declared it in. This keeps our code cleaner and smaller, and eliminates the risk of colliding with other variables with the same name.

Now that that's done, let's include this lua file in our extension.xml file:

```
<root release="3.0" version="3">
    ...
    <base>
        <icon file="graphics/TutorialTitle5E.png" name="TutorialTitle5E" />
        <icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
        <script name="ExtTutorial" file="lua/ExtensionTutorial.lua" />
        <includefile source="xml/TutorialWindow.xml" />
    </base>
</root>
```

And lastly, modify our onInit function to call our new function:

```
<banner_campaign>
    <script>
        function onInit()
            Debug.console("self", self)
            ExtTutorial.setIconByRuleset(self)
        end
    </script>
    <icon>TutorialTitle</icon>
</banner_campaign>
```

That's a Wrap!

Granted, it doesn't do much, but if you've never worked in Fantasy Grounds before, we covered a good bit of ground fairly quickly. In part 2 of this tutorial, we're going to take our MVP, add some contents, attach it to a database, and make it do something.

In particular, we're going to make it into a simple extension for creating and saving common chat phrases, with their appropriate emotes. In other word, if you're fond of typing “/ooc scratches his head and wonders aloud when dinner is”, then rather than take up a command slot for that, you'll be able to type it in once, select “ooc”, and then save it. From that point forward, a single click will send your message to the chat console.

Hmmm. That sounds pretty hard right now. Looks like I've got my work cut out for me...

Appendix A: The Final Result

If you managed to make it through all this, then you should have something fairly equivalent to this:

Text 1: extension.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--
    © Copyright Jeffery W Redding 2015+ except where explicitly stated otherwise.
    Fantasy Grounds is Copyright © 2004-2013 SmiteWorks USA LLC.

    Any use of material copyrighted by others is unintentional.  Let me know, and I'll make sure credit is
    given appropriately.

    This material is provided freely, without warranty of any kind.  This material is owned by me, so any use,
    modification, or publication, whether electronic, printed, verbal or anything else must provide appropriate
    attribution.
-->

<root release="3.0" version="3">
    <properties>
        <name>Extension Tutorial - Jeffery W. Redding</name>
        <version>0.0.1</version>
        <author>Jeffery W. Redding</author>
        <description>
            A really simple  extension file for
            creating a really simple extension
        </description>
    </properties>
    <base>
        <icon name="ExtTutorialLogo" file="ExtTutorialLogo.png" />
        <icon file="graphics/TutorialTitle5E.png" name="TutorialTitle5E" />
        <icon file="graphics/TutorialTitle.png" name="TutorialTitle" />
        <script name="ExtTutorial" file="lua/ExtensionTutorial.lua" />
        <includefile source="xml/TutorialWindow.xml" />
    </base>
</root>
```

Text 2: ExtensionTutorial.lua

```
-- ---------------------------------------------------------------------------------------------------------------------
-- This is the main file for the Lua Scripts for ExtensionTutorial.
-- Copyright Jeffery W. Redding 2015+ except where explicitly stated otherwise.
-- Fantasy Grounds is Copyright © 2004-2015 SmiteWorks USA LLC.

-- Any use of material copyrighted by others is unintentional.  Let me know, and I'll make
-- sure credit is given appropriately.

-- This material is provided freely, without warranty of any kind.  This material is owned by me, so any use,
-- modification, or publication, whether electronic, printed, verbal or anything else must provide appropriate
-- attribution.
-- ---------------------------------------------------------------------------------------------------------------------

-- ---------------------------------------------------------------------------------------------------------------------
-- setIconByRuleset - This function takes a windowcontrol for an argument and, if that windowcontrol has an icon
--                    it sets that icon to be one of two icons, based on the ruleset.
-- ---------------------------------------------------------------------------------------------------------------------
function setIconByRuleset(wc)
    if(wc.hasIcon() == false) then
        Debug.console("setIcon - no icon", wc)
        return
    end

    local rset = User.getRulesetName()
    Debug.console("setIconbyRuleset rulsetName", rset)
    if(rset == "5E") then
        wc.setIcon("TutorialTitle5E", true)
    else
        wc.setIcon("TutorialTitle", true)
    end
end
```

Text 3: TutorialWindow.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--
    © Copyright Jeffery W Redding 2015+ except where explicitly stated otherwise.
    Fantasy Grounds is Copyright © 2004-2013 SmiteWorks USA LLC.

    Any use of material copyrighted by others is unintentional.  Let me know, and I'll make sure credit is
    given appropriately.

    This material is provided freely, without warranty of any kind.  This material is owned by me, so any use,
    modification, or publication, whether electronic, printed, verbal or anything else must provide appropriate
    attribution.
-->

<root>
    <!-- This is the main window for the tutorial.  I'm using the framedef from RPGCore -->
    <windowclass name="TutorialWindow">
        <frame>campaignlistwithtabs</frame>
        <script>
            function onInit()
                --[[ Self points to a windowinstance, in this case. ]]
                Debug.console("self", self)
                Debug.console("windowclass", self.getClass())
                Debug.console("node", self.getDatabaseNode())
                Debug.console("frame", self.getFrame())
            end
        </script>
        <sizelimits>
            <dynamic />
        </sizelimits>
        <sheetdata>
            <banner_campaign>
                <script>
                    function onInit()
                        --[[ Self points to a windowcontrol, in this case. ]]
                        Debug.console("self", self);
                        ExtTutorial.setIconByRuleset(self);
                    end;
                </script>
                <icon>TutorialTitle</icon>
            </banner_campaign>
            <resize_campaignlistwithtabs />
            <close_campaignlist />
        </sheetdata>
    </windowclass>
</root>
```