Mozilla XUL: Making Interfaces

Building Blocks

In the first installment of our XUL programming tutorial last month, we explored some of the different widgets and facilities that XUL offers when constructing graphical interfaces. Although nice to look at, our interface was the most fickle you can imagine – all good looks, but no brains. BY JONO BACON

This month we are going to take our newfound knowledge of building interface elements and extend them with some real and useful functionality that will be used to create the building blocks for future applications using XUL.

Building blocks of interaction

Any form of graphical toolkit, and XUL is included here, is useless without the facilities to deal with interaction. When a user clicks on a button, scrolls a scrollbar, selects a tab, selects a menu item or clicks on a toolbar button, we should expect something to happen. As an example, if you click on the File menu and then click on Exit, you would expect the application to shut down as planned. This kind of interaction involves two types of process; an Event Handler and a Function.

Some readers may be familiar with this kind of functionality from other graphical toolkits; Qt has its Signal/Slots system for example. The basic premise is that each type of graphical control (such as a button, scrollbar, menu, toolbar, textbox etc) has a number of different ways in which the user can interact with it.

As an example, with a button drawn on the screen you can click it, with a list of items in a box you can select an entry and with a scroll bar you can scroll it. Each of these different types of interaction is called an Event. Some widgets (graphical controls are commonly referred to as widgets) also have a number of different events for different types of interaction.

Each of these events is useless if we cannot respond to that event with something useful, and to do this we need to use a function. A function is simply a special chunk of code that we can write that does something useful. This could change the text of some parts of the interface, add some information to a widget, process information or anything else that is useful.

The benefit of using functions is that they can be reused over and over again by different events. As an example, if you write a function that creates a new document, you would want to connect it to the *File->New* menu item as well as the toolbar button for a new document. This way everything connects together as one.

Dominating your code

With all of this theory discussed, you are probably wondering how all of this fits together in the bigger picture. How do we create an event handler, how do we create a function and how do we tie everything together to work seamlessly. The answer to all of these questions is Javascript.

Before we go any further, a few words should be mentioned about Javascript. Many people sneer at the thought of Javascript, and cite it as only useful for creating those annoying scrolling status bar messages and for creating menus that can make a website feel clunky and that only partially work in some browsers.

Admittedly, Javascript can be used for all of these things, and this is possibly the most public facing side of the language. In reality though, Javascript is a wonderfully compact and useful little language that can be considered the glue of a web browser; it is very useful for sticking together different types of functionality.

Within the land of XUL we use a special feature called the Document Object Model (DOM). The idea behind the DOM is that elements on a web page should all be accessible in code so that we can view and change the content of items as and when we need then. As an example, it would be useful to change the headline on a website, change the date, and edit the labels of buttons when you interact with the website.

A common example of this kind of functionality is with modern forum software. When you use the formatting buttons when composing a message to the forum, the button will add formatting tags to the main message textbox. How does the button know how to communicate with the message text box when you click on it? It simply uses the DOM.

The way the DOM works is to create a tree of all of the elements on your web page. Take the following code as an example:

```
<html>
<head>
  <title>My Website</title>
</head>
<body>
  <div>
    <h1>Welcome to my ➥
website</h1>
  </div>
  <div>
    Main content.
  </div>
</body>
</html>
```

Each of the tags in the code nests inside one another, apart from the <html> tag which is the parent tag. The parent tag is like the root of a tree, and each of the other tags is like a branch coming out of the tree. For example, if you look at the <body> tag, we have two <div> tags that are inside it. These two <div> tags are like two branches. Below we can see how our tags form into a tree, with the <html> tag as its root:

```
<html>
  <head>
    <title>My Website</title>
  <body>
    <div>
       <h1>Welcome to my ➥
website</h1>
    <div>
       Main content.
```

With our tags in a tree like this, we can use the DOM to find certain tags and display them or get the information required from them. This way we can find a particular tag and modify it in some way. As a result, the DOM not only provides a method of finding the right tags, but also a method of updating them.

DOM and XUL

To get us off to a flying start, we will create a simple example of creating a button with an Event/Function system. First of all, create a new file called *first.xul* and add the following code (see Listing 1 on the previous page):

When you load this file into a Mozilla browser, you will get a simple button as

XUL events

Event	Description
onclick	Called when the mouse is pressed and released on an element. You should only use the onclick event when you have a reason to only respond to mouse clicks. For buttons, menu items and the like, you would use oncommand to respond, because it also catches users who use the keyboard or other devices.
onmousedown	Called when a mouse button is pressed down on an element. The event handler will be called as soon as a mouse button is pressed, even if it hasn't been released yet.
onmouseup	Called when a mouse button is released on an element
onmouseover	Called when the mouse pointer is moved onto an element. You could use this to highlight the element, however, CSS provides a way to do this automatically – so you shouldn't do it with an event. You might, however, want to display some help text on a status bar.
onmousemove	Called when the mouse pointer is moved while over an element. The event will be called many times if the user moves the mouse – so you should try to avoid using this handler if you can.
onmouseout	Called when the mouse pointer is moved off of an element. You might then unhighlight the element or remove status text.
oncommand	This event is called when a button or menu item is selected. For menus, add this handler to the menuitem element. You should use this handler rather than handling the mouse yourself as the user might select the button or menu item with the mouse or by pressing the access key or keyboard shortcut.
onkeypress	Called when a key is pressed and released when an element has the focus. You might use this to add extra shortcut key handling or to check for allowed characters in a field. We'll see how to create keyboard shortcuts in a later section.
onkeydown	Called when a key is pressed down while an element has the focus. Note that the event will be called as soon as the key is pressed, even if it hasn't been released. You probably won't use this event very often as the other key events are more suitable.
onkeyup	Called when a key is released while an element has the focus.
onfocus	Called when an element receives the focus either by clicking with the mouse or by using the TAB key. You might use this event to highlight the element or display some help text.
onblur	Called when an element loses the focus either by a user clicking on another element or by pressing TAB. You might use this to verify information or close popups. It is better to verify fields when the OK button is clicked however.
onload	Called on a window when it first opens. You would usually add this event handler to a window tag in order to initialize a window. This would allow fields to be set to default values based on conditions contained in a script.
onunload	Called when the window is closing. You would usually add this to the window tag to record information before the window closes.

Listing 1: first.xul

```
<?xml version="1.0"?>
<?xml-stylesheet
   href="chrome://global/skin/"
   type="text/css"?>

<window
  id="test-window"
  title="Test Program"

   xmlns="http://www.mozilla.org/
   keymaster/gatekeeper/there.is.
   only.xul">

    <button id="name"
   label="My button"/>

</window>
```

expected. Our first step is to now add an event handler to the button to specify what kind of interaction we want to respond to. Each event is triggered by a special handler that specifies the type of interaction that we are dealing with. The XUL events table is a list of the currently supported events that are available in XUL (this list is taken from *XULPlanet.com*).

Button Power

We will begin by exploring one of the most common event handlers; 'onclick'. To add this handler, you need to add 'onclick' to the line of code that creates the button, and then specify what should happen. Change the button code to this line:

```
<button id="name" label="My but➥
ton" onclick="alert('hello');"/>
```

In this code we are adding the 'onclick' handler, and inside the quotation marks we say what we want the handler to do. In this simple example we are using the Javascript 'alert' function to pop up a dialog box with the word hello inside it. If you now save the code, reload the page and click on the button, you will see the resulting box pops up when you click on the button with your mouse.

Getting all functional

As you can see from our first example, connecting functionality to a widget is quite simple. Despite this simplicity, we face the problem that our code is going to get very long and confusing if we put all of our functionality inside the 'onclick' quotation marks. What do we do if we need to perform some clever maths, expansive processing or other such programming magic that would require a lot of code to be written? The answer is we make our own function.

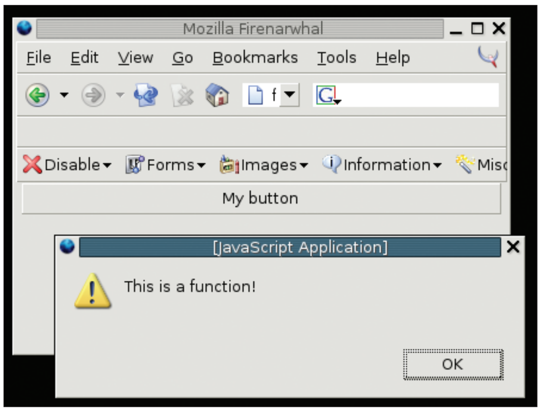
A function is a block of code that can be called into action by another piece of code. You can think of a function as a book inside a library. You may be reading another book and not understand a particular subject, so to understand the subject you will pick up another book to explore it further. A function basically allows you to package together in code a complex chunk of programming code and access it with a single line. Although functions can be quite complex and handle much functionality, they can also be simple. We will explore our first use of functions in Javascript with a very simple example.

To use functions, it is recommended that you put all of your functions in a separate file and access the file from your XUL script. To begin, we need to create a file called functions.js and add the following code:

```
function func_showdialog()
{
  alert("This is a function!");
}
```

This code contains a number of key elements. The first line says that we are creating a function ('function'), and that we want to call it func_showdialog. The empty brackets are used to indicate that our function does not process anything. A common feature of functions is that you can pass them information in the brackets, and they will process the information and give you a result. With our function here, we are simply running the code inside it, and not processing anything, hence the empty brackets.

The curly brackets indicate the extent of the function. The code within the function begins after the { symbol and finishes before the closing } symbol. In our current function we have one line that gives us a familiar alert dialog box. An important point to note is the semicolon at the end of each statement inside the programming function; this semicolon is used to indicate the end of a line in Javascript.


Mozilla Firenarwhal
File Edit View Go Bookmarks Tools Help
Disable Forms Images Information Misc
My button
[JavaScript Application]
OK

This is a function!

Figure 1: Our first Javascript enabled script.

Linking together

With our function complete, we are now ready to connect it to our main XUL code. We first need to load the functions.js file into our XUL file as a source of scripting code. To do this, we use the <script> tag inside the <window> tag to specify the source of the file. In addition to this we will also need to change the content of the button's onclick attribute to our function name. To clarify these additions, the full code is shown in Listing 2:

Listing 2: Full code

```
<?xml version="1.0"?>
<?xml-stylesheet
href="chrome://global/skin/"
type="text/css"?>

<window
   id="test-window"
  title="Test Program"

xmlns="http://www.mozilla.org/
keymaster/gatekeeper/there.is.
only.xul">

    <script
src="functions.js"/>

    <button id="name"
label="My button"
onclick="func_showdialog();"/>

</window>
```

Other languages and XUL

One of the biggest questions that people ask when looking into XUL is if there are any other languages available for writing XUL scripts. At the current time, the only language that is supported is Javascript, but other languages are being discussed for future releases of Mozilla. These languages could include Python, C#, and others. Many of these discussions are based around the creation of the next level of Mozilla application functionality; the Mozilla Virtual Machine. For more information on the progress of the Mozilla project, we recommend you visit the Mozillazine (*http://www.mozillazine.org/*) and Planet Mozilla (*http://planet.mozilla.org/*) websites.

The main reason why I reproduced the entire code here was to demonstrate the correct placement of the new code. If you were to put the <script> tag below the <?xml> tags (a common mistake for those who think that the script applies to the entire file) you would get an error and your efforts would go to waste. You need to ensure that the script is within the <window> tag.

Exploring the DOM

We will now step forward and explore the DOM a little more. To get started we will create some new code. Call this file second.xul, and add the following code as seen in Listing 3 below:

In this code we have two main graphical elements; a text box and a button. It is important to note how each widget has an 'id' attribute that gives it a unique name. You should ensure that this attribute is both unique and easy to remember as we will be referring to widgets by this name. In our code we have also changed the name of the function to func_getinfo() as well as the name of our functions file in line 9 to second.js.

The next file to edit is second.js. Add the code shown in Listing 4 at the bottom of this page. This is the code where the real action happens. The aim of our script is that the user can type something into the textbox, and the contents of the textbox will be displayed in an alert dialog box. To do this we need to communicate with the textbox, get the information, and put it into an alert box.

In the first line of code in the function, we are creating a new variable called 'info'. To do this we use the 'var' keyword to indicate that 'info' is a variable. For those unfamiliar with variables, a variable allows you to store information in the computer's memory and refer to it with a name. You can think of a variable as a cardboard box with a name written on the side. If you put something in the box, you can then refer to it by the name on the side of the box. In this case, the name written on the side of the box is 'info'.

On the same line as our 'var info' code we also set the contents of the variable with the code on the right-hand side of the = sign; this is the code that actually gets the information from the textbox. Within the code we use the getElementById feature of Javascript to access the widget with the id of 'textbox', which is our textbox.

You will see that the getElementById function is written right next to the 'document' word. The way this works is that 'document' refers to our main XUL document and the widgets that are included on it.

Period

The period sign (.) indicates that the code on the right of the sign should be applied to the code on the left of the sign. In this case we look for a widget with the id 'textbox' within our main document. With this line complete, we now have the contents of out textbox stored in the 'info' variable.

This concept of using periods to indicate where we are applying a function is a common feature in many programming languages. We use this concept again in our second line when we use the 'value' function to get the value from our 'info' variable. We nest this code within an alert function to display it in a dialog box. Note how we do not use double quotes in this alert box code; you only use double quotes when you are printing out text (called a string) to the alert box. In our alert box, this time, we are printing the contents of a variable.

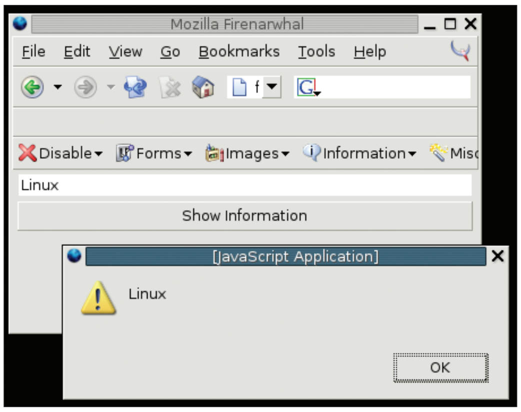

Mozilla Firenarwhal
File Edit View Go Bookmarks Tools Help
Disable Forms Images Information Misc
Linux
Show Information
[JavaScript Application]
Linux
OK

Figure 2: Using the DOM to connect our widgets together.

Conclusion

In this issue we have made some important first steps towards making a functional XUL interface. We still have some way to go, but these foundations will make our future scripts easier to understand and write. It is always important to have a clear understanding of the core fundamentals before we continue along the path.

In the next issue we are going to push forward and write some more elaborate functions and features to tie together different parts of our XUL interfaces. Until then, have a good look through the code again and try to understand as many of the core concepts discussed today as you can. If some things are confusing, don't worry; we will be exploring many of these concepts in better detail in future issues. Good luck! ■

Listing 3: second.xul

```
01 <?xml version="1.0"?>
02 <?xml-stylesheet
   href="chrome://global/skin/"
   type="text/css"?>
03
04 <window
05   id="test-window"
06   title="Test Program"
07
   xmlns="http://www.mozilla.org/
   keymaster/gatekeeper/there.is.
   only.xul">
08
09     <script src="second.js"/>
10
11     <textbox id="textbox"/>
12     <button id="showinfo"
   label="Show Information"
   onclick="func_getinfo();"/>
13
14 </window>
```

Listing 4: second.js

```
01 function func_getinfo()
02 {
03   var
   info=document.getElementById('
   textbox');
04   alert(info.value);
05 }
```

THE AUTHOR

Jono Bacon is a writer/journalist, consultant and developer based in England. Jono has been actively involved with Linux since 1998 and has worked on a number of different projects including KDE, KDE::Enteprise, KDE Usability Study, Kafka and Linux UK. You can find his website at http://www.jonobacon.org.