XUL and Mozilla development

Browser Building

Of all the inventions of the last 50 years or so, few nuggets of conceived genius have risen to a point where we all use these products regularly in our daily lives. Even so, we do not have to remain stuck with its initial design flaws.

BY JONO BACON

Although such gems as the Sinclair C5, the Chopper, Tab Clear and Microsoft FoxPro have fallen by the wayside, there are some products and technologies that have stuck. One such technology is the Web.

There is little doubt that the Web has made a huge impact in the way we communicate, shop and do other things. Despite the politics of the browser wars, the incompetently implemented standards and the censorship attempts by certain American politicians, the Web has proved to be a compelling medium where access is now pretty much assumed. Sometimes I can hear the words "what, you haven't got the Internet?" echoed across the land.

Despite the adoption of the Web, the limitations have become apparent. The most visible limitation is the fact that the entire interface needs to be re-invented with each developed website. Also, the interface needs to be reloaded with each (even marginal) change to a page. Not only is this inefficient in the way that redundant HTML needs to be blasted back and forth between the server and the browser, but it creates an environment where more dynamic changes to the page are more difficult to perform, giving the web a distinctly "clunky" feel.

Enter XUL

The dependence on HTML and its expected browser functionality is really the crux in the problems that we have just outlined. Although subsequent technologies such as Dynamic HTML (DHTML) and the Document Object Model (DOM) have emerged to attack the problem, these technologies need a degree of coaxing to get them working.

The developers behind the popular Mozilla browser had a different idea. With some extensive discussion and design, the hackers have worked together to create the XML User Interface Language (XUL) to partially solve this problem. Pronounced 'zool' (and inspired by Ghostbusters), the XUL language essentially seeks to re-create the user interface features typically associated with normal graphical toolkits such as Qt and GTK. Features such as buttons, scrollbars, tabs, menus and more are all available within the XUL toolkit. Many of these features are simply not available with normal HTML forms. Each of the features within XUL is used by writing (rather unsurprisingly) XML files. For those of you who feel a sharp pain in the head when you read the letter XML, I will give a quick overview of what XML is.

XML is a set of rules and conventions that allows you to create languages that look similar to HTML. These languages include tags, attributes, content and other concepts that constitute HTML. The XML technology allows you to create your own tags and language that can be used for your own purpose. For example, if we wanted to store addresses in XML, we could create:

```
<address>
 <forename>Bob</forename>
 <surname>Smith</surname>
 <address>25, The Grove, London
, W2 4DF</address>
 <phone> 020 344 5443</phone>
</address>
```

Here we are using specifics tags to denote specific chunks of information, and we can then write software to read these tags in and use the content in the way we want to.

XUL follows the same concept, but the tags are used to create specific interface elements. Where the magic occurs though, is when Mozilla reads the tags in and actually creates the interface elements for you. The XML file is a simple means of specifying what you want in your interface and where.

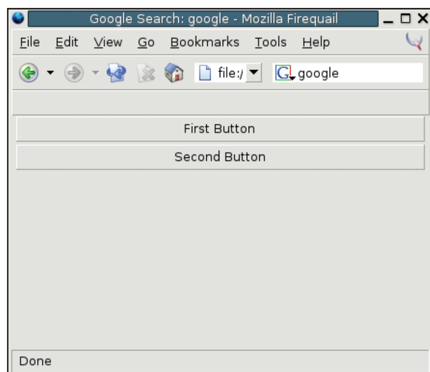

Figure 1: Not exactly the cleverest of XUL scripts, but it is a start!

Getting started

In this first part of our series, I will be showing you how to get a XUL based interface mocked up. This interface will comprise of some of the different XUL components available, and although we will not be covering how to actually make these widgets do something until the next issue, we will have a good grounding in actually creating our visual interfaces. To get started, we will create a simple XUL file that simply contains two buttons. To do this, create a new file called xul1.xul and add the following code to it. See Listing 1.

When you have added the code, use File | Open File to locate the file and load it into Mozilla. You should see something similar to Figure 1.

Any XML file, irrespective of what it is doing, should really have some lines at the top of the file that indicate the version of the XML, and a stylesheet if applicable. In our example, we have the following two lines:

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

As you can see, we have a version line that specifies this XML version as 1.0. The second line indicates that our stylesheet is in the chrome path. The chrome path contains some internal Mozilla facilities that typically manage the user interface of Mozilla.

Our next lines contain our first tag:

```
<window
 id="firstwindow"
 title="First XUL Window"
 xmlns="http://www.mozilla.org/↩
keymaster/gatekeeper/there.is.↩
only.xul">
```

Although we have four lines of code here, this is actually a single tag; I have just split this across a number of lines to make it easier to read. Every XUL page that you create needs a <window> tag that can be used to contain the widgets that form your interface. Inside this tag we have also used three attributes. The first (id) is a unique reference that points to this tag in the XML. The id attribute is essential for being able to communicate with tags and update them with changes and information. This will become clearer later when we use the DOM to actually update and reference tags. The second tag (title) contains a human readable string that is displayed in the title bar when you launch a XUL file in a dedicated window. If you load the file into the browser as we have done, this text is ignored. The final attribute is the xmlns part. This value specifies the namespace on which this <window> tag, and all tags within, are based. A namespace is like a special group that you can specify to determine where a tag comes from. This helps in situations when you may have a <window> tag from another XML language and a <window> tag from the XUL language – a namespace tells them apart.

We are now ready to put something in our window. In our example, we are creating two buttons with the code:

```
<button id="button1" label=↩
"First Button"/>
<button id="button2" label=↩
"Second Button"/>
```

Each line of code is very similar, with only the id and label attributes containing different content. The id attribute behaves in exactly the same way as the <window> equivalent – it is used to reference the tag later. The label attribute contains the actual text that will appear on the button.

Some of you may be wondering what the backslash is doing in the tag. Unlike some forms of HTML, where you can leave off tags here and there, XML is very strict about correct markup. With the case of our <button> tags, we should really include a closing </button> tag to keep with the rules of the XML. The backslash at the end of our <button> tag is shorthand for including the </button> tag. You will see this kind of shorthand used commonly in XML. To finish off our file, we include the closing </window> tag.

Layout management

Something you will have noticed from our first example is the way that the second button is positioned below the first. This is the default behavior for widgets that have no kind of layout specified. Although fine for simple pages, this stock method of laying out widgets is not flexible enough. This is where we need to use a Layout Manager.

Layout management is something that is common in most GUI widget sets such as Qt and GTK. The simple premise is that you place widgets inside another invisible widget that lays your visible widgets out in a particular method. In most toolkits, this comprises a Horizontal and Vertical layout manager. This standard method of handling layouts has been transferred over to XUL, and we consequently have the <hbox> and <vbox> tags. Here is an example of <hbox> management:

```
<hbox>
<button id="button1" label=↩
"First Button"/>
<button id="button2" label=↩
"Second button"/>
</hbox>
```

Listing 1: Xul1.xul

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

<window
    id="firstwindow"
    title="First XUL Window"

xmlns="http://www.mozilla.org/key
master/gatekeeper/there.is.only.x
ul">
        <button id="button1"
label="First Button"/>
        <button id="button2"
label="Second Button"/>
</window>
```

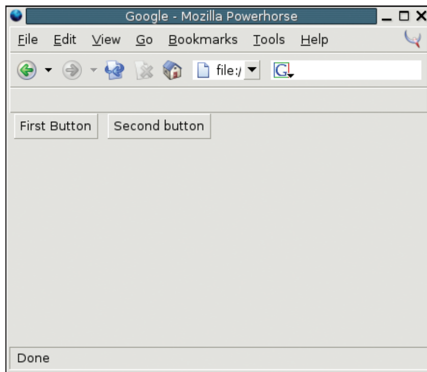

Figure 2: Horizontal layout management.

The way the <hbox> tag works, is to horizontally place widgets between the <hbox> and </hbox> tags next to each other. The result of this code is Figure 2.

The other kind of layout management is the <vbox> tag. This tag behaves in exactly the same way as the <hbox> tag, but displays its child widgets (widgets inside the <vbox> and </vbox> tags) vertically. If we place a similar block below our <hbox> block, you can see how it works. I have also put (H) on the horizontally managed buttons and (V) on the vertically managed buttons:

```
<hbox>
<button id="button1" label=↩
„First Button (H)"/>
<button id="button2" label=↩
„Second button (H)"/>
</hbox>
<vbox>
<button id="button1" label=↩
„First Button (V)"/>
<button id="button2" label=↩
„Second button (V)"/>
</vbox>
```

Where things get really interesting is when you combine one type of layout manager inside another. Take the following code for example:

```
<hbox>
<button id="button1" label=↩
„First Button (H)"/>
<button id="button2" label=↩
„Second button (H)"/>
<vbox>
<button id="button1" label=↩
„First Button (V)"/>
<button id="button2" label=↩
„Second button (V)"/>
</vbox>
</hbox>
```

Here we have put the vertically managed within the horizontally managed buttons. Note how we have placed the vertical block of buttons after widgets within the horizontal block. Due to the placement of our widgets, you should see something such as that in Figure 3.

One thing to be aware of when we are placing our widgets, is how the layout managers handle space. In our last example, the horizontal buttons were stretched wider to accommodate the room for the vertical buttons. This is because the vertical layout manager was nested inside the horizontal manager and affected the horizontal widgets.

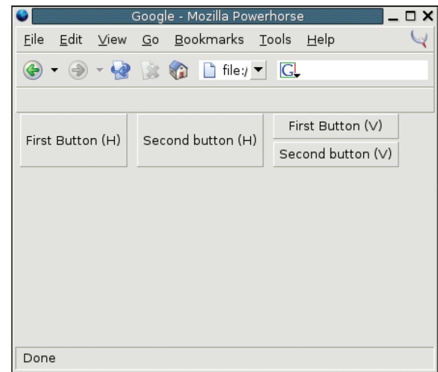

Figure 3: Combining horizontal and vertical layout management.

More exotic widgets

The true power of XUL lies in the way it can surpass HTML as a means of getting information from the user. This power lies in the way we can use normal GUI application widgets, but within the concept of the web. In the next example we are going to create two tabs, one with a multi-line text editing widget and one with a selectable list box. In this example, the tabs and the list box are not typically used in a web environment. We will go through this example step by step and write the code as we progress.

HTML type widgets

In our exploration of the XUL landscape we have currently only made use of layout managers and push buttons. There are a great many other widgets that we can make use of, and we will first look at the common HTML style widgets that can be used. We will learn these tags by running some code:

```
<vbox>
<hbox>
  <label value="Checkboxes"/>
  <vbox>
  <checkbox id="check1" label=↩
„First"/>
  <checkbox id="check2" label=↩
„Second"/>
  </vbox>
</hbox>
<hbox>
  <label value="Radio buttons"/>
  <vbox>
  <radio id="radio1" label=↩
„First"/>
  <radio id="radio2" label=↩
„Second"/>
  </vbox>
</hbox>
<hbox>
  <label value="Text box"/>
  <vbox>
  <textbox id="textbox"/>
  </vbox>
</hbox>
<hbox>
  <label value="Multiline ↩
Text box"/>
  <textbox id="multitextbox" ↩
multiline="true"/>
</hbox>
</vbox>
```

With this code we are laying out a number of widget descriptions and their corresponding widgets. We are using the <label> tag to specify text in our XUL interface. We use the attribute of of this tag to contain the text that we want to display in the label.

Our first type of widget is a checkbox. We create this with the <checkbox> tag, and we use the label attribute to specify the text next to the checkbox. The second widget we use is a radio button, and we use the <radio> tag in the same way to create this widget. Next, we create a single line text edit box. There is no label associated with the box, so we simply set the id attribute within the <textbox> tag. Finally, we create a multiline text box. To do this, we simply add multiline="true" to a normal text box.

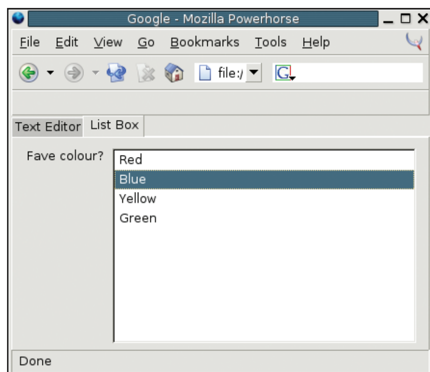

Figure 4: Using tabs and a list box.

First, create a new file with the XML version, stylesheet and <window> tags, and then add the following lines:

```
<tabbox>
  <tabs>
    <tab label="Text Editor"/>
    <tab label="List Box"/>
  </tabs>
```

Here we begin by creating a new widget that contains tabs (<tabbox>). A tab box will contain a number of tabs that in turn can contain other widgets. We then open up the <tabs> tag to specify the names of the tabs in our interface. For each tab we use the <tab> tag to specify what label should be defined . Your tabs will be added from left to right in the order in which you specify them in the XUL file. In this case, the Text Editor tab will be the tab on the left, and the List Box tab will be on the right. We now need to create our actual tab panels. We do this by first creating a general tab panels tag:

```
<tabpanels>
```

Now we create each panel in turn. First we will create our textbox panel. We use the <tabpanel> tag to create each panel and then fill it with other widgets:

```
<tabpanel id="text">
  <label value=➥
"Type in some text:"/>
  <textbox id="textbox" ➥
multiline="true" flex="1"/>
</tabpanel>
```

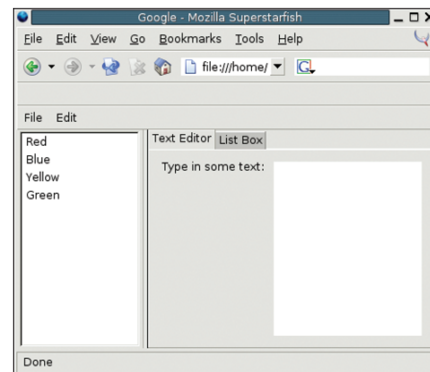

Figure 5: A complete XUL interface.

You may have spotted the flex attribute that sneaked its way into the code. When you set this attribute to 1, the widget will be stretched to take up all available space.

For our second panel we will create our list box inside the panel. We use the <listbox> tag to create the main box, and then we use the <listitem> tag to add each item to the box.

```
<tabpanel id="listbox">
  <label value="Fave colour?"/>
```

Want to know what's up next?

Subscribe to Linux Magazine Preview, our free monthly email newsletter!

WWW.LINUX-MAGAZINE.COM/NEWSLETTER

```
  <listbox flex="1">
      <listitem label="Red"/>
      <listitem label="Blue"/>
      <listitem label="Yellow"/>
    <listitem label="Green"/>
  </listbox>
</tabpanel>
```

Finally, we close the tab panels and main tab box:

```
</tabpanels>
</tabbox>
```

You can see our completed interface in Figure 4.

A complete interfaces

To complete this first installment on XUL programming, we will run through a complete example of a XUL interface. This interface will include some of the code we have already covered as well as some menus and a splitter (resizeable divider). We will step through every line of code to cement the understanding of everything that we have covered.

First, we will begin with the XML definition tags and the creation of a main window. See Listing 2.

The first widgets that I will add are some menus. Creating them follows the same principle that we have been using previously by nesting tags inside each other to build the different elements. We begin by first creating a menu bar (the bar that the menus sit in) with the <menubar> tag.

```
<menubar id="menubar">
```

Next, we will add a complete menu. In this case, the File menu:

Listing 2: Main Window

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

<window
    id="complete"
    title="Complete Example"

xmlns="http://www.mozilla.org/key
master/gatekeeper/there.is.only.x
ul">
```

```
<menu id="filemenu" ↩
label="File">
  <menupopup id="file-popup">
    <menuitem label="New"/>
    <menuitem label="Open"/>
    <menuitem label="Save"/>
    <menuseparator/>
    <menuitem label="Exit"/>
  </menupopup>
</menu>
```

To create the menu, we first use the <menu> tag to create the actual menu entry, and then we use the <menupopup> to create the drop down menu popup area. Finally, we add a number of menu items with the <menuitem> tags. We will now use the same concept to create the Edit menu:

```
<menu id="editmenu" ↩
label="Edit">
   <menupopup id="editpopup">
    <menuitem label="Undo"/>
 <menuitem label="Redo"/>
</menupopup>
  </menu>
</menubar>
```

With our menus added, we are ready to create our main interface area. If you look at Figure 5 you can see the result of our interface and how it is constructed.

In our interface we have a list box on the left side of the screen and the tabs on the right side. To manage this layout we first need to open up a <hbox> tag and then create the listbox:

```
<hbox>
  <listbox flex="1">
    <listitem label="Red"/>
    <listitem label="Blue"/>
    <listitem label="Yellow"/>
    <listitem label="Green"/>
  </listbox>
```

Next we are going to use a special widget called a splitter to add a resizeable bar that allows the user to adjust the size of the widget on the left and the right of the divider. We use the <splitter/> tag to create this widget:

```
<splitter/>
```

The next chunk of code is our familiar tab box that contains text editing and list box tabs. See Listing 3. This code is no different from the code in our previous example. Finally we close the horizontal layout manager and the window:

```
</hbox>
</window>
```

Conclusion

In this first part of our series we have covered a lot of ground. Not only have we thrown ourselves kicking and screaming into the world of XML and XUL programs, but we have covered HTML style widgets, special widgets, layout management, tab boxes, menus and more. With the knowledge we have developed so far, we have the ability to to create fairly expansive XUL interfaces. There are of course many more widgets available, and we will cover some of these in a later issue.

Next month we are going to take our knowledge from this issue and make it fully functional. We are going to take the JavaScript capabilities of Mozilla and merge them with XUL to make our interfaces interact with the user. ■

Listing 3: Tab box

```
  <tabbox>
    <tabs>
      <tab label="Text Editor"/>
      <tab label="List Box"/>
    </tabs>
    <tabpanels>
      <tabpanel id="text">
        <label value="Type in
some text:"/>
        <textbox id="textbox"
multiline="true" flex="1"/>
      </tabpanel>
      <tabpanel id="listbox">
        <label value="Fave
colour?"/>
        <listbox flex="1">
          <listitem label="Red"/>
          <listitem
label="Blue"/>
          <listitem
label="Yellow"/>
          <listitem
label="Green"/>
        </listbox>
      </tabpanel>
    </tabpanels>
  </tabbox>
```