Mozilla: Inside the Lizard's Lair

By Jerry Peek

After a development cycle that seemed like an eternity — during which Internet Explorer supposedly "won" the "browser war" — Mozilla, the open source application suite that's also the basis for Netscape, has become a stable and capable package with a new codebase. With Mozilla, you can browse the Web, read email and newsgroups, chat, and more. If you use multiple platforms, Mozilla is an ideal client: you can have the same, familiar Mozilla features and user interface on every machine you use (including Macintosh, Windows, Linux, Unix, just to name a few).

However, Mozilla is much more than just a browser and email agent. Targeted at developers, the *Mozilla toolkit* is full of features that programmers can fall in love with: a JavaScript debugger and DOM Inspector; XUL, a simple XML-based language for building user interfaces; X Window System remote control options; and more. All of these features can be leveraged to write other browsers (such as Netscape, Chimera, and Galeon) and a variety of other kinds of applications, including development tools, games, and browser enhancements. The Web site http://www.mozdev.org is full of Mozilla-based applications.

Mozilla is released daily for a dozen platforms and can be compiled for many others. Based on Unicode, it's due to be available in 38 languages. Mozilla also supports a long list of other standards, both established and emerging. For example, DTD files define strings for a given locale, and RDF files (XML files that describe data) identify components in an application package. Mozilla's wide use of standards lets developers create applications that run on many platforms in a host of countries, without change.

This month, let's take an in-depth look at Mozilla. We'll start with an overview of some nice user interface features, then see how to write a simple GUI application with the toolkit. Although the basic concepts are fairly simple, there's a lot of ground to cover in just a few pages. We're going to move quickly!

Mozilla Tidbits

Mozilla version 1.0, released in June 2002, is the stable Mozilla "base." Newer releases, versions numbered 1.*x*, started with version 1.1 in August, and version 1.2 was in Beta at the time we went to press. The final 1.2 release may be out by the time you read this.

Mozilla comes from http://www.mozilla.org. Though the website is mostly for developers, it has an FAQ and a User's Guide. If you'd like to follow along, go to the Web site now and download and install a copy. (Installation note: don't install Mozilla into a directory that contains a previous version. Additionally, if you have Netscape, Mozilla will copy Netscape's profile settings. If you want to prevent that, rename your *~/.netscape* or *~/.mozilla* directory before you start the Mozilla installation.)

The default Mozilla application suite has lots of features! Here are just a few:

Mozilla is an ideal client: it works on Linux, Mac, Windows, and more

- Browser *tabs* let you open multiple Web pages in the same browser window, a feature also found in the Opera browser (http://www.opera.com). Tabs are handy if you need to open and browse multiple, related pages. To open a new tab, simply right-click on any link and choose "Open in New Tab" from the menu (the middle mouse button can also be configured to do this). The CTRL-PGUP and CTRL-PGDN keyboard shortcuts cycle through the tabs. You can also bookmark a window full of tabs.

- Mozilla's security and convenience features control Web page scripts (for example, you can block pop-ups), manage cookies on a site-by-site basis, and remember passwords and entries in forms.

- *Themes* change the appearance of Mozilla. Themes are a nice entry point for Mozilla's internals, which we'll look at momentarily.

- Mozilla's "Mail & News" application supports multiple accounts and servers, message labeling ("Important", "Personal", "Do Later", etc.), automatic address completion, message filters, and an integrated address book that can also collect addresses automatically.

Mozilla also provides a set of "hooks" into its internal features. You can think of these hooks as a kind of "remote control," where shell scripts, window managers, and other applications can launch and control Mozilla programmatically.

For example, here's a *bash* alias that opens a new composition window to send email to *myteam@mycorp.com*:

```
alias mailteam='mozilla -remote  "
  mailto(myteam@mycorp.com)"'
```

Here, the argument after `-remote` is quoted to protect it from the shell. As another example, if you'd like to create a launcher button or menu entry that opens the Linux Magazine home page in a new browser tab, use the command:

```
mozilla -remote 'openURL \
  (http://www.linux-mag.com, new-tab)'
```

Hooks are convenient, and they're quick and efficient because they don't launch a new Mozilla process for each task. Instead, hooks use the X protocol to send messages to an existing process (Mozilla raises itself over other windows when it receives a hook message). For more information on the remote control feature, see http://www.mozilla.org/unix/remote.html.

The Mozilla development team has also produced tools that are valuable for developers of *any* kind of code — not just Mozilla. *Bugzilla* is a flexible bug-tracking system. The *Gecko* engine, which parses and renders standards-compliant HTML, can be embedded into other applications. *LXR* (http://lxr.mozilla.org/) displays the Mozilla source code and makes cross-referencing easy.

Morphing Mozilla Madness

As mentioned above, Mozilla is more than a browser. It's also a cross-platform toolkit for developing applications. Components supplied with Mozilla — and the browser is just one of the components — can be configured and customized extensively. If you've configured older versions of Netscape, you probably know about the *prefs.js* file that stores user preference settings. Mozilla has *prefs.js*, too, but even more of the underlying architecture is exposed and ready for hacking. One good example of this — which is also a gentle introduction into Mozilla's architecture — is Mozilla's implementation of themes or skins.

From the "Edit" menu, choose "Preferences," and then "Themes." Mozilla comes with two default themes: Classic and Modern. You'll also see a "Get New Themes" link on the Themes dialog. Click that link and one of the subsequent links to see some of the many themes available on the Net.

Let's look at three themes, shown in *Figure One*. Each

FIGURE ONE: The Orbit theme, The Star Trek Console theme, and The Internet Explorer theme

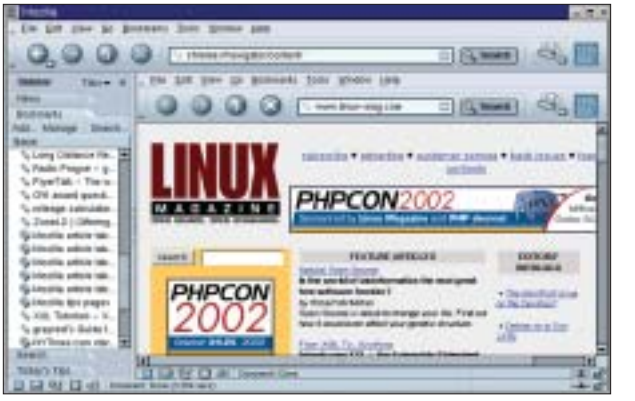

FIGURE TWO: The Mozilla browser running *inside* the Mozilla browser

image in the figure shows the same top-left corner of the browser. The top image shows "Orbit," by Chris Neale and Kent Thuresson (http://themes.mozdev.org/skins/orbit.html). The middle image shows "LCARStrek," a Star Trek console theme by Robert Kaiser http://themes.mozdev.org/skins/lcarstrek.html). The bottom image shows just how far you can take this theme business — you can even make Mozilla look like Internet Explorer, thanks to Bamm Gabriana (http://mozillako.hypermart. net/ieskin).

On any of the Web pages that preview themes, clicking the "Install Theme" link associated with a particular theme should open a dialog that downloads and installs a theme automatically. After you install a theme, notice how it not only changes the buttons and images in the browser, but also changes the look and feel of the application's dialog boxes, such as "Preferences."

How does it work? Here's a tip: use `cp -r` to make a copy of your *~/.mozilla* directory before installing the theme, then compare the two directories with `diff -r`.

Interface elements for a theme come from a downloaded JAR file. A typical JAR has a manifest in an RDF file and an image file with a preview of the interface. JAR subdirectories have lots of style sheets, such as *editor/textEditor.css*, and images, such as *editor/icons/btn1.gif*. A few other changes are made to *~/.mozilla* — especially to the registry file *default/…/chrome/chrome.rdf*, which is important for Mozilla customization.

Standalone Applications

While themes can make Mozilla look and act cool, themes don't add new features to Mozilla. Beyond themes, you can design and build independent applications based on the Mozilla toolkit. In fact, entire applications can run in a Mozilla browser window. For (a clever though not especially practical) example, type the URL *chrome://navigator/content/* into the location bar. Notice that Navigator (the browser application) can run

inside itself, as shown in *Figure Two*.

Similarly, the URL *chrome://editor/content/* brings up Composer, the Mozilla HTML editor, inside a browser window.

Application programming with Mozilla uses standard technologies including HTML, XML, JavaScript (ECMAScript), and Cascading Style Sheets (CSS). With some "glue" provided by the Mozilla development team, you can put these pieces together into a cross-platform application that runs anywhere Mozilla runs. Let's go ahead and build one.

Packaging an Application

Packaging means organizing application files into a structure that Mozilla supports. Although it's possible to drop all the files into one directory — or even to wedge almost everything into a single interface description file (described momentarily)— most packages have a top-level directory and multiple subdirectories with multiple files.

For instance, a subdirectory named *content* might hold scripts and interface description files for the application's structure and features, a *locale* directory might have DTDs and other resources for supported locales, and a *skin* subdirectory might hold image and CSS files for realizing the application's layout. Most important, you have to package properly to directly access other Mozilla resources, via *chrome://* URLs. (The simplest packages, like themes, don't need this access.) Let's look at a basic package.

XUL (pronounced *zool*, which rhymes with *cool*, and stands for XML-based User-interface Language) is a user interface description language designed for Mozilla. Along with other technologies created by the Mozilla team, as well as other standard technologies, XUL does more than create the look and feel of an application. Mozilla becomes a layer between the application and the operating system. Because Mozilla runs on so many platforms, learning to code with XUL (and familiar tools like JavaScript) makes it straightforward to create cross-platform applications.

FIGURE THREE: (Left) Before clicking either button; (Right) after clicking both buttons

(Because XUL and Mozilla internals haven't been completely standardized, development can sometimes be a bit of a challenge. For instance, the original examples for this column, which were partly based on code in apps from *http://www.mozdev.org*, failed in ways we couldn't completely debug before deadline. So, we "punted" and tried something else.) We recommend developing for the stable 1.0.x version, then testing against the bleeding-edge releases. If you have problems, check the newsgroups and other development plans and then — in the tradition of open-source development projects — report bugs to *http://bugzilla.mozilla.org*.)

Mozilla applications are a lot like Web pages that use a markup language (HTML), a scripting language (JavaScript), and stylesheets. Add DTDs (Document Type Definitions) for localization and you have most or all of what you need for an application. Unfortunately, there isn't room to show more than the most basic example here.

LISTING ONE: The user interface for the sample application (chrome/moztest/content/moztest.xul)

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="chrome://global/skin/"
     type="text/css"?>
3  <?xml-stylesheet href="chrome://moztest/skin/"
     type="text/css"?>
4  <!DOCTYPE window SYSTEM
     "chrome://moztest/locale/moztest.dtd" >
5
6  <window id="example-window" title="Mozilla Test"
7  xmlns:html="http://www.w3.org/1999/xhtml"
8  xmlns="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul">
9
10 <script type="application/x-javascript"
11 src="chrome://moztest/content/moztest.js" />
12
13 <!— Example HTML. Note that corresponding XUL tag
     is "image", not "img" —>
14 <html:img
15 src="http://www.linuxmagazine.com/downloads/2003-01/
     power/opener.gif"
16 width="170" height="107" />
17
18 <label id="label1"/>
19 <button id="button1"
20 label="&button1.val;"
21 oncommand="setLabel('label1');" />
22
23 <label id="label2"/>
24 <button id="button2"
25 label="&button2.val;"
26 oncommand="setLabel('label2');" />
27
28 <button id="button3"
29 label="&button3.val;"
30 oncommand="window.close();" />
31
32 </window>
```

Our simple application, *moztest*, has an image, three buttons, and two labels that appear when you click the corresponding button. *Figure Three* shows the application window as it starts and after both buttons have been clicked.

LISTING TWO: *chrome/moztest/content/moztest.js*

```
1  function setLabel(label_id)
2  {
3    var elem=document.getElementById(label_id);
4    elem.setAttribute("value","Thank you very much for clicking " + label_id);
5    window.sizeToContent();
7  }
```

Some Mozilla applications come from an XPI (Cross-Platform Installer) file instead of a simple JAR file. The XPI file has a script named *install.js* that performs the install. The XPI file may also contain JAR files. Download of the XPI file is triggered by clicking a web page link which, with a bit of code, triggers an install dialog.

For simplicity, let's do by hand the steps that an XPI script and archive would perform for a user. We'll start with the puzzle pieces, the files that make up this simple package.

Listing One shows the XUL file for the *moztest* application. You can see that it's a text file in XML syntax; its name ends with *.xul*. When Mozilla sees an *.xul* extension, it expects that file to draw UI widgets. Our *moztest.xul* file makes the user interface window, in order (by default) from top to bottom. When you compare the file to the image on the right in *Figure Three*, you should see what code makes each button and label.

The `<window>` element, starting at line 6, is the root of most XUL documents. Line 7 declares the HTML namespace, which lets us use HTML code in lines 14-16. In this way, you can mix HTML (actually, XHTML) in with XUL. Notice the XML conventions, like the closing `/>` on the `img` element in line 16. As the comment in line 13 says, we could have used XUL here instead.

This simple layout has some problems. For instance, as the first label get its content (after one of the buttons is clicked), the graphic is stretched horizontally. XUL has plenty of layout power to avoid problems like these, but we're using the simplest layout we can for this simple example.

LISTING THREE: *chrome/moztest/locale/moztest.dtd*

```
<!ENTITY button1.val "This is button 1">
<!ENTITY button2.val "This is button 2">
<!ENTITY button3.val "quit">
```

LISTING FOUR: *chrome/moztest/skin/moztest.css*

```
window { background-color: yellow; }
```

Lines 10-11 of *Listing One* refer to the JavaScript code shown in *Listing Two*. This little function writes the labels shown in *Figure Three* when a user clicks a button. See Lines 21 and 26 of *Listing One*.

How does the XUL code find its corresponding JavaScript code? We'll see that soon in *Listings Five* through *Eight*.

Listing Three is a DTD file that defines three entities used as the text for the three buttons. This locale information can be customized by the user — to, for instance, make the labels in French — with RDF information like that in *Listing Six*. These entities are used at lines 20, 25, and 29 of *Listing One*. In a real-life example, you'd define entities for any text that would change from locale to locale.

Listing Four is a CSS file that defines the simple "skin" for this application. This can also be customized by the user with RDF information like that in *Listing Seven*. We also could have added style information for other elements than the root `<window>`, but we'll keep this example simple.

Before looking at *Listing Five* through *Listing Eight*, let's see where each listing's

LISTING FIVE: *chrome/moztest/content/contents.rdf*

```
1  <?xml version="1.0"?>
2
3  <RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4  xmlns:chrome="http://www.mozilla.org/rdf/chrome#">
5
6  <RDF:Seq about="urn:mozilla:package:root">
7  <RDF:li resource="urn:mozilla:package:moztest"/>
8  </RDF:Seq>
9
10 <RDF:Description about="urn:mozilla:package:moztest"
11 chrome:displayName="moztest"
12 chrome:author="your-name"
13 chrome:name="moztest">
14 </RDF:Description>
15
16 </RDF:RDF>
```

file is stored. When you install Mozilla on your system, its root directory — like */usr/local/mozilla* — has a subdirectory named *chrome*, like */usr/local/mozilla/ chrome*. This directory has information about registered applications and packages; it's the root of the special *chrome://* URLs that you've already seen.

When you add an application to Mozilla, you typically add a new subdirectory under *chrome* (though you can actually put the subdirectory anywhere you want). Here, for example, we've made a *moztest* subdirectory, */usr/local/mozilla/chrome/moztest*.

This new application subdirectory typically has three subdirectories of its own: *content*, *locale*, and *skin*. Those subdirectories contain the application's XUL, JavaScript, CSS, and DTD files (see the pathnames for *Listings One* through *Four*). There can be more files than just four, of course — but this is a simple example!

The *content*, *locale*, and *skin* subdirectories also hold manifest files, one per directory, each named *contents.rdf*. These are shown in *Listings Five* through *Seven*. A lot of this is standard boilerplate which doesn't change from manifest to manifest. What does change is:

- Line 7 of each manifest, which identifies the package name. To register more than one package, we could have added more `li` entries after Line 7.
- The package name — here, *moztest*.
- Information to help users find you, the application author. See line 12 in *Listings Five* and *Six*.
- The *skin* manifest shows that these skin resources only apply to Mozilla's *modern/1.0* theme. This implementation is trivial; a fancier application window could have much more.

To make Mozilla register these packages when it starts up, we add the three lines in Listing Eight to the end of the *installed-chrome.txt* file.

We're done! In this short column, there's not room to explain any of these puzzle pieces in detail. The XUL tutorial from XUL Planet (which covers more than XUL, by the way) is excellent.

We've added these files to the top-level *chrome* directory manually — which is handiest to do while you're developing a Mozilla application.

A finished application would typically have an XPI file that installs these files and registers them in *installed-chrome.txt*. If you do the same with your Mozilla, you can launch this application with a command like:

LISTING SIX: *chrome/moztest/locale/contents.rdf*

```
 1  <?xml version="1.0"?>
 2
 3  <RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 4  xmlns:chrome="http://www.mozilla.org/rdf/chrome#">
 5
 6  <RDF:Seq about="urn:mozilla:locale:root">
 7  <RDF:li resource="urn:mozilla:locale:en-US"/>
 8  </RDF:Seq>
 9
10  <RDF:Description about="urn:mozilla:locale:en-US"
11  chrome:displayName="English(US)"
12  chrome:author="your-name"
13  chrome:name="en-US"
14 chrome:previewURL="http://www.mozilla.org/locales/en-US.gif">
15  <chrome:packages>
16  <RDF:Seq about="urn:mozilla:locale:en-US:packages">
17  <RDF:li resource="urn:mozilla:locale:en-US:moztest"/>
18 </RDF:Seq>
19  </chrome:packages>
20  </RDF:Description>
21
22  </RDF:RDF>
```

LISTING SEVEN: *chrome/moztest/skin/contents.rdf*

```
 1  <?xml version="1.0"?>
 2
 3  <RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 4  xmlns:chrome="http://www.mozilla.org/rdf/chrome#">
 5
 6  <RDF:Seq about="urn:mozilla:skin:root">
 7  <RDF:li resource="urn:mozilla:skin:modern/1.0" />
 8  </RDF:Seq>
 9
10 <RDF:Description about="urn:mozilla:skin:modern/1.0">
11  <chrome:packages>
12  <RDF:Seq about="urn:mozilla:skin:modern/1.0:packages">
13  <RDF:li resource="urn:mozilla:skin:modern/1.0:moztest"/>
14  </RDF:Seq>
15  </chrome:packages>
16  </RDF:Description>
17
18  </RDF:RDF>
```

LISTING EIGHT: Lines appended to chrome/installed-chrome.txt

```
content,install,url,resource:/chrome/moztest/content/
skin,install,url,resource:/chrome/moztest/skin/
locale,install,url,resource:/chrome/moztest/locale/
```

```
mozilla -chrome chrome://moztest/content/
```

Compare that URL to the earliest examples in this article and you'll see how Mozilla launches its standard applications. For instance, you can launch the HTML Composer directly from a command line with the command `mozilla -chrome chrome://editor/content/`. (Note that, if you have more than one instance of Mozilla running at any one time, each instance needs to use a separate Mozilla profile. You can define profiles by running the Profile Manager: `mozilla -profilemanager`.)

More Goodies

Mozilla also includes a JavaScript Debugger and a DOM Inspector — along with the familiar Java and JavaScript consoles (where error messages and warnings are shown, among other things). The DOM (Document Object Model) Inspector can show the internal tree structure of documents that Mozilla has parsed, making it easy to see and debug overall structure.

This, plus the many other goodies for developers at *www.mozilla.org* and *www.mozdev.org*, make Mozilla a cross-platform developer's dream.

Whew! We've seen that Mozilla is much more than the basis for a bunch of browsers. It blends standard technologies like JavaScript and DTDs, along with the XUL interface language and more, to form a framework for cross-platform applications.

Although Mozilla is still evolving rapidly, the 100 applications registered at *www.mozdev.org* — along with the browser, mail/news and chat clients, and more — already show its power for building open-source apps that can run almost anywhere.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com. You can download the source code used in this month's column from http://www.linuxmagazine.com/downloads/2003-01/power.