Gecko Embedding Basics

Given the ever-growing importance of the Web as a source of information, entertainment, and personal connectedness, the ability to access and view data stored in HTML format is becoming more and more important for a wide variety of otherwise highly divergent software applications. Whether it's a matter of a simple HTML page viewer or of a full-fledged web browser, the ability to parse and render HTML-based documents is an increasingly significant function in many, many situations. For the application developer, the problem becomes how to implement this crucial functionality in a way that minimizes development time yet results in an agile and robust product. Embedding Gecko, the rendering engine at the heart of the Netscape and Mozilla browsers, is an outstanding solution to this problem.

Why Gecko

Gecko is the smart embedding choice. It is quick, robust, and highly standards compliant. In its Mozilla and Netscape incarnations, it has been widely distributed and very well reviewed.

It is Open Source. Unlike other embedding choices, all of Gecko's source code is freely available and fully customizable. You can tinker and tweak as much as you need. Yet, depending on the license chosen, it is quite possible to use Gecko as a component in what is otherwise a fully proprietary commercial product.

And because Gecko is associated with the Mozilla project, there are many resources available to assist the embedding effort. The Mozilla web site, mozilla.org, has an embedding project area at mozilla.org/projects/embedding/. There is a newsgroup, netscape.public.mozilla.embedding, focussed on exchanging information among embedders, as well as a number of other related newsgroups. A complete cross-reference for the codebase is available at lxr.mozilla.org/seamonkey/. And filing, following the progress of, and helping to fix any bugs is made simple through the Bugzilla bug database, bugzilla.mozilla.org/.

Gecko is also architected from the ground up to be cross-platform. Directly from mozilla.org, it runs on Wintel, Mac OS 9.0 and OS X, and Linux, and there are third-party ports to a number of other platforms.

Finally, licensing Gecko is royalty-free, even if the final application is an otherwise proprietary commercial product. *Very* generally, any modifications of the original

Mozilla-supplied source code (but not the code in which it is embedded) must be returned to the community, that same original code must be made available to the application's users (often by a link to the mozilla.org website), and the application must indicate in some obvious way (for example, a logo on the box or on the About: page) that the product embeds Gecko. Exact descriptions of the possible licensing arrangements are presented at www.mozilla.org/MPL/, which is the only legally complete source for licensing information.

What You Need to Embed

Once you've decided to embed, there are three main steps that you must take. First you must get the code. Then you must understand some specific technologies used in the manipulation of the Gecko codebase. Finally, you must decide which additional functionalities you may wish to add. This section will guide you through these steps.

Getting the Code

At the moment, the best way to get the files you will need to embed Gecko is to download and build the entire Mozilla source tree. This is actually a fairly simple process. Full instructions and appropriate links are available at mozilla.org/source.html. A second, component by component, method is under development, but is still at a beta stage. Information on this project can be found at www.mozilla.org/projects/embedding/bootstrap.html. In addition, work is also being done on developing a Gecko Runtime Environment, or GRE, which would support multiple applications built on Mozilla components using a single set of core libraries. This project lives at www.mozilla.org/projects/embedding/MRE.html.(If you intend to work component by component you must be particularly aware of issues of binary compatibility. For help in this area, look at mozilla.org/projects/xpcom/glue/Component_Reuse.html.)

First you must acquire some tools (basically a supported compiler, a Perl distribution, and some general purpose utilities). Then you must set up your computer environment. Next you must download the source. Assuming you are going to download the entire tree, there are two ways to do this: you can FTP a tarball of the entire tree (this is the simplest way, and it's guaranteed to compile, but it may not include the most recent additions to the code) or you can use CVS to get the absolutely most recent code or to do incremental updates. Once you have the tree and the tools and your environment is properly set up, all you have to do is run the appropriate provided makefile. There are detailed instructions for each of the supported platforms.

When the build is done, navigate to the mozilla/embedding/config directory. There you will find sample manifest files (all the names begin with "basebrowser") for embedding on each of the different platforms. These are samples only and they may not fit your needs completely, but they are a good place to start. There are also sample

embedding projects for each platform that you can use as models. See mozilla.org/projects/embedding/examples/index.html.

Understanding the Coding Environment

Mozilla was set up from the beginning to support design and development across multiple platforms and programming languages. To this end, a number of in-house programming technologies were developed, all based around an ideal of object encapsulation. Embedding Gecko necessarily implies acquiring a working knowledge of these technologies, including XPCOM, XPIDL, XPConnect, special string classes, and, optionally, XUL. The following provides a brief introduction to them. More information can be found at the mozilla.org site.

XPCOM

The most important of the Mozilla technologies is XPCOM, the Cross-Platform Component Object Model. XPCOM provides a framework which manages the creation, ownership, and deletion of objects and other data throughout Mozilla. If you have used MSCOM, you will recognize certain basic similarities. But there are also significant differences - XPCOM is cross-platform and designed to run largely in a single thread - and the two are not at this time compatible.

The interface

At the core of XPCOM is the concept of the interface. An interface is simply a description of a set of methods, attributes, and related constants all associated with a particular functionality: it is completely distinct from the class that implements those things. The interface serves as a kind of contract: any object that supports a particular interface guarantees that it will perform the services described in it. To keep the interface as language neutral as possible, it is written in a special language, the Interface Definition Language, or IDL. Interface files are often referred to as .idl files. In addition to specifying the functionality of the interface, these files also carry the interface's IID, its globally unique identifying number.

Much of the communication within Gecko takes place in terms of these abstract structures (by convention, their names follow the form `nsISomething`).

```
//this

void ProcessSample(nsISample* aSample) {
    aSample->Poke("Hello");

//not this

void ProcessSample(nsSampleImpl* aSample) {
    aSample->Poke("hello");
```

@status FROZEN

XPCOM's level of abstraction produces great flexibility in the system. Implementations are free to change as needed. But, to work, the interfaces themselves must remain fixed. Throughout Mozilla's initial design and development period, interfaces have been somewhat fluid, but as the project has matured, more and more of the interfaces have been marked FROZEN. Any interface so marked is guaranteed not to change in the future.

Most of the main interfaces key to the embedding effort are now frozen, but it's always a good idea to check before using any interface. An interface's status is listed in the .idl file's comments. A frozen interface is marked @status FROZEN. You can search for frozen interfaces by using the mozilla cross referencing tool at lxr.mozilla.org/seamonkey/search?string=%40status+FROZEN. Until it is frozen, an interface may change at any time. For more information on the freezing process, see the embedding project page at: mozilla.org/projects/embedding/.

Once an interface has been frozen, it is added to the Gecko Embedding API Reference at mozilla.org/projects/embedding/embedapiref/embedapi.html.

nsISupports

A single object can support more than one interface. In fact, essentially all objects support at least two interfaces — a minimum of one that does something specifically useful and one, nsISupports, that serves a more general purpose. In a sense, nsISupports is the progenitor of all XPCOM interfaces. All interfaces inherit from it, most directly so. It serves two main functions — runtime type discovery and object lifetime management. It is functionally identical to IUnknown in MSCOM.

Since an object can support multiple interfaces, it is perfectly possible to have a pointer to one interface and want to know whether that same object also supports a different interface whose functionality you might also need. The first nsISupports method, QueryInterface, does exactly that: it asks, in effect, I know that this object is

of type A (supports interface A) but is it also of type B (supports interface B)? If it is (or does), QueryInterface returns to the caller a pointer bound to the newly requested interface.

```
void ProcessSample(nsISample* aSample) {
  nsIExample *example;
  nsresult rv;
  rv = aSample->QueryInterface(NS_GET_IID(nsIExample),
                               (void **)&example);
  if (NS_SUCCEEDED(rv)) {
    example->DoSomeOperation();
    NS_RELEASE(example); // using a macro to call Release
  }
}
```

Because XPCOM uses an indirect method, the Component Manager, to actually instantiate objects, and because multiple pointers to the same object — often bound to different interfaces — can exist, it can quickly become very difficult for callers to keep accurate track of all of the objects to which those pointers point. Objects could be kept around in memory longer than they need to be, causing leaks, or objects could be deleted prematurely, causing dangling pointers. The other two methods in nsISupports, AddRef and Release, are designed to deal with this issue. Every time a pointer is given out AddRef must be called on the object, incrementing an internal counter. Every time a pointer is released, Release must be called, decrementing that same counter. When the counter reaches zero, there are no pointers to the object remaining and the object can safely delete itself. Control of the object's lifetime stays within the object itself. See below for information on XPCOM's "smart" pointer, nsCOMPtr, a utility which helps automate this process.

Object creation

The instantiation of objects is also an indirect process in XPCOM. Just as interfaces have a globally unique ID number (the IID), XPCOM classes are assigned their own GUIDs, the CID. In addition, they are also often given a text-based ID, called a contract ID. One or the other of these IDs is passed to a method on a persistent XPCOM component, the Component Manager, which actually creates the object. When a new library of classes (called a module in XPCOM) is first introduced into the system, it must register itself with the Component Manager, which maintains a registry that maps classes (with their IDs) to the libraries in which they reside.

A limited number of persistent services, supplied by singleton objects, are created and controlled by a companion to the Component Manager, the Service Manager. The Component Manager itself is an example of such a persistent service.

Summing up

Functionality in XPCOM is described by abstract interfaces, and most communication among parts of the system takes place in terms of those interfaces. The underlying objects that implement the interfaces, on the other hand, are created indirectly by the Component Manager based on a cross-indexed registry that it maintains.

One functionality shared by all interfaces is the ability to query the underlying object at runtime to see if also implements other interfaces. In theory an interface is fixed and unchangeable, but at this stage in the Mozilla codebase, only interfaces that have been declared FROZEN are guaranteed not to change significantly. Object lifetime management takes place inside the object itself through an internal counter that keeps track of the number of pointers to the object that have been added or released. The client's only responsibility is to increment and decrement the counter. When the internal counter reaches zero, the object deletes itself.

nsCOMPtr

Sometimes, however, even remembering to call AddRef and Release at the right times can be difficult. To make this process easier and more reliable, XPCOM has a built-in "smart" pointer, nsCOMPtr. This pointer takes care of calling AddRef and Release for you. Using nsCOMPtr whenever possible will make your code cleaner and more efficient. For more information on the smart pointer, see "The Complete nsCOMPtr User's Manual" at www.mozilla.org/projects/xpcom/nsCOMPtr.html.

Mozilla actually provides a large number of built-in macros (by convention, written in all caps in the code) and utilities like nsCOMPtr that can make the entire process of coding with XPCOM easier. Many of these can be found in the following files: nsCom.h, nsDebug.h, nsError.h, nsIServiceManager.h, and nsISupportsUtils.h Mozilla also supplies other development tools for tracking memory usage and the like. More information on these can be found at www.mozilla.org/performance/

For more information

More information on XPCOM in general can be found at mozillla.org/projects/xpcom/. For an overview of creating XPCOM components, see Chapter 8 of O'Reilly's *Creating Applications with Mozilla*, an open source version of which is available at books.mozdev.org/chapters/ch08.html. For a fuller explanation of some of the underlying logic to COM systems, see also the early chapters of *Essential COM* by Don Box. While it focusses on MSCOM in particular, the book does provide an excellent background on some of the core rationales for using such an object model.

XPIDL

Interfaces are abstract classes written in XPIDL, the Cross Platform Interface Definition Language. Yet to be useful the functionality promised in those interfaces must be implemented in some regular programming language. Facilitating this is the job of the

XPIDL compiler. Once an interface is defined in an .idl file, it can be processed by the XPIDL compiler.

The compiler can be set to output a number of things, but generally the output is two-fold: a C++ .h file that includes a commented out template for a full C++ implementation of the interface and an XPT file that contains type library information which works with XPConnect to make the interface available to JavaScript. More information on the syntax of XPIDL (a simple C-like language) and the use of the compiler can be found at mozilla.org/scriptable/xpidl/index.html.

XPConnect and XPT files

XPConnect is an XPCOM module that allows code written in JavaScript to access and manipulate XPCOM components written in C++ and vice versa. By means of XPConnect, components on either side of an XPCOM interface do not, in general, need to know or care about which of these languages the object on the other side is implemented in.

When an interface is run through the XPIDL compiler, it produces an XPT or type library file. Because XPconnect uses the information in this file to implement transparent communication between C++ objects and JavaScript objects across XPCOM interfaces, it is important to make sure they are generated and included with your code even if you are developing exclusively in C++. Not only is a substantial part of the browser, in fact, implemented in JS, it is possible that in the future someone may wish to use JS-based code to interact with whatever components you create .

As is from Mozilla, XPConnect currently facilitates interoperability between C++ and JS. Modules to extend it to allow access from other languages (including Python) are under independent development. Further information can be found at mozilla.org/scriptable/index.html.

String classes

Web browsing typically involves a large amount of string manipulation. Mozilla has developed a hierarchy of C++ classes to facilitate such manipulation and to render it efficient and quick. To make communication among objects simpler and more error free, Mozilla uses interfaces, which are, in essence, abstract classes. The string hierarchy is also headed up by a set of abstract classes, nsAString, nsASingleFragmentString, and nsAFlatString, and for the same reasons. (These refer to double-byte strings. There is a parallel hierarchy topped with nsACString, etc., that refers to single-byte strings.) nsAString guarantees only a string of characters. nsASingleFragmentString guarantees that the characters will be stored in a single buffer. nsAFlatString guarantees that the characters will be stored in a single null-terminated buffer. While there are underlying concrete classes, in general it is best to use the most abstract type possible in a given situation. For example, concantenation can be done virtually, through the use of pointers, resulting in an nsAString that can be used like

any other string. This saves the allocating and copying that would otherwise have to be done. For more information, see "Guide to the Mozilla string classes" at www.mozilla.org/projects/xpcom/string-guide.html.

XUL/XBL

Use of this final Mozilla technology is optional, depending on how you decide to create the user interface for your application. XUL is Mozilla's highly flexible XML UI Language. It provides a number of largely platform independent widgets from which to construct a UI. Netscape and Mozilla both use XUL for their interfaces, but not all embedders chose to use it. XBL or the eXtensible Binding Language allows you to attach behaviors to XUL's XML elements. More information on XUL can be found at www.mozilla.org/xpfe/xulref/ and on XBL at www.mozilla.org/projects/xbl/xbl.html. There is also a wealth of good information on XUL at XulPlanet, www.xulplanet.com/.

Choosing Additional Functionalities

As of this writing (11/6/02), Gecko is a partially modularized rendering engine. Some functionalities beyond basic browsing are always embedded with Gecko, and, as a result of certain architectural decisions, always will be; some are at present always embedded with Gecko, but may, at some point in the future, be separable; and some are now available purely as options. The following table describes the present status of these additional functionalities:

Functions	Status Now	Status in Future
FTP support	Optional	
HTTPS support	Optional	
International character support	Optional	
XUL support	Required	Probably optional
Network support	Required	Maybe optional

Functions	Status Now	Status in Future
JavaScript support	Required	Maybe optional
CSS support	Required	Always required
DOM support	Required	Probably always
XML support	Required	Probably always

At this time embedding Mozilla's editor along with the rendering engine Gecko is an uncertain proposion. A substantial part of the API is in flux. For more information on the status of the embeddable editor, see www.mozilla.org/editor/editor-embedding.html.

What Gecko Provides

The following is a description of some of the interfaces most commonly used in embedding Gecko. It is by no means an exhaustive list of the available interfaces. The interfaces in this section are on classes provided by Mozilla. There is also a set of interfaces for which Gecko expects the embedder to provide the implementation. A sample of those are covered in the next section.

Initialization and Teardown

There are two C++ only functions which serve to initalize and terminate Gecko. The initialization function (NS_InitEmbedding) must be called before attempting to use Gecko. It ensures XPCOM is started, creates the component registry if necessary, and starts global services. The shutdown function (NS_TermEmbedding) terminates the Gecko embedding layer, ensuring that global services are unloaded, files are closed and XPCOM is shut down.

nsIWebBrowser

Use of this interface during initialization allows embedders to associate a new nsWebBrowser instance (an object representing the "client-area" of a typical browser window) with the embedder's chrome and to register any listeners. The interface may also be used at runtime to obtain the content DOM window and from that the rest of the DOM.

nsIWebBrowserSetup

This interface is used to set basic properties (like whether image loading will be allowed) before the browser window is open.

nsIWebNavigation

The nsIWebNavigation interface is used to load URIs into the web browser instance and provide access to session history capabilities - such as back and forward. It is not, at this writing (11/6/02) frozen.

nsIWebBrowserPersist

The nsIWebBrowserPersist interface allows a URI to be saved to file. It is not, at this writing (11/6/02) frozen.

nsIBaseWindow

The nsIBaseWindow interface describes a generic window and basic operations (size, position, window title retrieval, etc.) that can be performed on it. It is not, at this writing (11/6/02), frozen.

nsISHistory

The nsISHistory interface provides access to session history information and allows that information to be purged.

nsIWebBrowserFind

The nsIWebBrowserFind interface controls the setup and execution of text searches in the browser window.

What You Provide

The following is a description of some of the more common embedder-provided interfaces used in embedding Gecko. It is by no means an exhaustive list of the available interfaces.

nsIWebBrowserChrome

The nsIWebBrowserChrome interface corresponds to the top-level, outermost window containing an embedded Gecko web browser. You associate it with the WebBrowser through the nsIWebBrowser interface. It provides control over window setup and whether or not the window is modal. It must be implemented.

nsIEmbeddingSiteWindow

The nsIEmbeddingSiteWindow interface provides Gecko with the means to call up to the host to resize the window, hide or show it and set/get its title. It must be implemented.

nsIWebProgressListener

The nsIWebProgressListener interface provides information on the progress of loading documents. It is added to the WebBrowser through the nsIWebBrowser interface. It must be implemented. As of this writing (11/6/02), it is not frozen.

nsISHistoryListener

The nsISHistoryListener interface is implemented by embedders who wish to receive notifications about activities in session history. A history listener is notified when pages are added, removed and loaded from session history. It is associated with Gecko through the nsIWebBrowser interface. Implementation is optional.

nsIContextMenuListener

The nsIContextMenuListener interface is implemented by embedders who wish to receive notifications for context menu events, i.e. generated by a user right-mouse clicking on a link. It should be implemented on the web browser chrome object associated with the window for which notifications are required. When a context menu event occurs, the browser will call this interface if present. Implementation is optional.

nsIPromptService

The nsIPromptServices interface allows the embedder to override Mozilla's standard prompts: alerts, dialog boxes, and check boxes and so forth. The class that implements these embedder specific prompts must be registered with the Component Manager using the same CID and contract ID that the Mozilla standard prompt service normally uses. Implementation is optional. As of this writing (11/6/02), this interface is not frozen.

Common Embedding Tasks

The following is a series of code snippets (taken from MFCEmbed, the Windows based embedding Gecko sample) which demonstrate very briefly implementation associated with common embedding tasks.To see all the files associated with this sample, go to lxr.mozilla.org/seamonkey/source/embedding/tests/mfcembed/. There are also Linux- and Mac OS-based examples.

Gecko setup

The Gecko embedding layer must be initialized before you can use Gecko. This ensures XPCOM is started, creates the component registry if necessary, and starts global services. There is an equivalent shutdown procedure.

Note that the embedding layer is started up by passing it two parameters. The first indicates where the executable is stored on the file system (nsnull indicates the working directory). The second indicates the file location object "provider" that specifies to Gecko where to find profiles, the component registry preferences, and so on.

```
nsresult rv;
rv = NS_InitEmbedding(nsnull, provider);
if(NS_FAILED(rv))
{
ASSERT(FALSE);
return FALSE;
}
```

Creating a browser instance

The embedder-provided BrowserView object calls its method `CreateBrowser`. Each browser object (a webbrowser) represents a single browser window. Notice the utility directive `do_CreateInstance` and the use of macros.

```
//Create an instance of the Mozilla embeddable browser

HRESULT CBrowserView::CreateBrowser()
{
// Create a web shell
nsresult rv;
mWebBrowser =
      do_CreateInstance(NS_WEBBROWSER_CONTRACTID, &rv);
if(NS_FAILED(rv))
return rv;
```

Once the nsWebBrowser object is created the application uses `do_QueryInterface` to load a pointer to the msIWebNavigation interface into the mWebNav member variable. This will be used later for web page navigation.

```
rv = NS_OK;
mWebNav = do_QueryInterface(mWebBrowser, &rv);
if(NS_FAILED(rv))
return rv;
```

Next the embedder-provided CBrowserImpl object is created. Gecko requires that some interfaces be implemented by the embedder so that Gecko can communicate with the embedding application. See the What You Provide section. In the sample, CBrowserImpl is the object that implements those required interfaces. It will be passed into the SetContainerWindow() call below.

```
mpBrowserImpl = new CBrowserImpl();
if(mpBrowserImpl == nsnull)
return NS_ERROR_OUT_OF_MEMORY;
```

The mWebBrowser interface pointer is then passed to the CBrowserImpl object via its `Init` method. A second pointer to the platform specific BrowserFrameGlue interface is also passed in and saved. The BrowserFrameGlue pointer allows CBrowserImpl to call methods to update status/progress bars, etc.

```
mpBrowserImpl->Init(mpBrowserFrameGlue, mWebBrowser);
mpBrowserImpl->AddRef();
```

Next the embedder-supplied chrome object is associated with the webbrowser. Note the use of an nsCOMPtr.

```
mWebBrowser->SetContainerWindow
      (NS_STATIC_CAST(nsIWebBrowserChrome*,
      mpBrowserImpl));

nsCOMPtr<nsIWebBrowserSetup> setup(do_QueryInterface
                                (mWebBrowser));
if (setup)
    setup->SetProperty
      (nsIWebBrowserSetup::SETUP_IS_CHROME_WRAPPER,
                          PR_TRUE);
```

The real webbrowser window is created.

```
rv = NS_OK;
mBaseWindow = do_QueryInterface(mWebBrowser, &rv);
if(NS_FAILED(rv))
return rv;
```

Binding a window

Basic location information is passed in.

```
RECT rcLocation;
GetClientRect(&rcLocation);
if(IsRectEmpty(&rcLocation))
{
      rcLocation.bottom++;
      rcLocation.top++;
}
rv = mBaseWindow->InitWindow(nsNativeWidget(m_hWnd),
      nsnull,0, 0, rcLocation.right - rcLocation.left,
      rcLocation.bottom - rcLocation.top);
rv = mBaseWindow->Create();
```

Note the m_hWnd passed into the call above to InitWindow(). (CBrowserView inherits the m_hWnd from CWnd). This m_hWnd will be used as the parent window by the embeddable browser.

Adding a listener

The BrowserImpl object is added as an nsIWebProgressListener. It will now receive progress messages. These callbacks will be used to update the status/progress bars.

```
nsWeakPtr weakling
      (dont_AddRef(NS_GetWeakReference(NS_STATIC_CAST
      (nsIWebProgressListener*, mpBrowserImpl))));
void)mWebBrowser->AddWebBrowserListener(weakling,
      NS_GET_IID(nsIWebProgressListener));
```

Finally the webbrowser window is shown.

```
mBaseWindow->SetVisibility(PR_TRUE);

nsCOMPtr<nsIWebBrowserPrint>
      print(do_GetInterface(mWebBrowser));
if (print)
      {print->GetNewPrintSettings
      (getter_AddRefs(m_PrintSettings));
      }
return S_OK;
}
```

Using session history to navigate

The pointer to nsIWebNavigation saved above is used to move back through session history.

```
void CBrowserView::OnNavBack()
{
if(mWebNav)
mWebNav->GoBack();
}
```

Resources

The mozilla.org home page: www.mozilla.org

The embedding project area home page: mozilla.org/projects/embedding/

Cross-reference for the entire codebase: lxr.mozilla.org/seamonkey/

Bugzilla, the bug tracking tool: bugzilla.mozilla.org

Licensing information: www.mozilla.org/MPL/

Getting the source and build instructions: mozilla.org/source.html

Embedding samples: mozilla.org/projects/embedding/examples/index.html

The list of frozen interfaces: lxr.mozilla.org/seamonkey/search?string=%40status+FROZEN

The Complete nsCOMPtr User's Manual:www.mozilla.org/projects/xpcom/nsCOMPtr.html

Performance tools: www.mozilla.org/performance/

XPCOM project home page: mozilla.org/projects/xpcom/

Building XPCOM components: books.mozdev.org/chapters/ch08.html

XPIDL: mozilla.org/scriptable/xpidl/index.html

XPConnect: mozilla.org/scriptable/index.html

Guide to the Mozilla string classes: mozilla.org/projects/xpcom/string-guide.html

XUL: www.mozilla.org/docs/xul/xulnotes/index.html

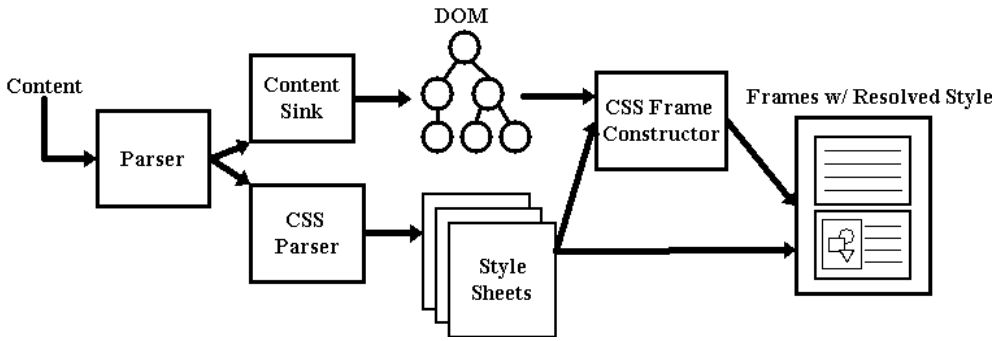
XBL: www.mozilla.org/projects/xbl/xbl.html

Editor embedding: www.mozilla.org/editor/editor-embedding.html

Source for the MFCEmbed sample: lxr.mozilla.org/seamonkey/source/embedding/tests/mfcembed/

Appendix: Data Flow Inside Gecko

While it isn’t strictly necessary for embedders to understand how Gecko does what it does, a brief overview of the main structures involved as Gecko puts bits on a display may be helpful.



HTML data comes into Gecko either from the network or a local source. The first thing that happens is that it is parsed, using Gecko's own HTML parser. Then the Content Model arranges this parsed data into a large tree. The tree is also known as the "Document" and its structure is based on the W3C Document Object Model. Any use of DOM APIs manipulates the data in the Content Model.

Next the data is put into frames using CSS and the Frame Constructor. A frame in this sense (which is not the same thing as an HTML frame) is basically an abstract box within which a DOM element will be displayed. This process produces a Frame Tree, which, like the Content Model, is a tree of data, but this time focussed not on the logical relationship among the elements but on the underlying calculations needed to display the data. In the beginning a frame has no size. Using CSS rules specifying how the elements of the DOM should look when they are displayed, including information like font type or image size, the eventual size of each frame is calculated. Because the same data may need to be displayed in different ways — e.g. to a monitor and to a printer, etc. - a particular Content Model may have more than one Frame Tree associated with it. In such a case, each individual Frame Tree would belong to a different "presentation" mode.

Calculations continue as new information flows into the system using a process called Reflow. As information in the Frame Tree changes, the section of the Frame Tree involved is marked "dirty" by the Frame Constructor. Reflow repeatedly steps through the tree, processing every "dirty" item it encounters until all the items it encounters are "clean". Every item in the Frame Tree has a pointer back to its corresponding item in the Content Model. A change in the Content Model, say through using the DOM APIs to change an element from hidden to visible, produces an equivalent change in the Frame Tree. It's important to note that all of these operations are purely data manipulations. Painting to the display itself is not yet involved to this point.

The next stage is the View Manager. With a few small exceptions that have to do with prompting the Frame Constructor to load graphics, the View Manager is the first place

in the process that accesses the native OS. Delaying OS access until this point both helps Gecko to run more quickly and makes cross-platform issues easier to deal with. The View Manger is the place where Gecko figures out where on the display the data will need to be drawn. It then tells the system that that area is "invalid" and needs to be repainted. The actual painting is managed by the gfx submodule, while other low-level system operations are run through the widget submodule, which handles things like platform specific event (like mouse clicks) processing loops and accessing system defaults (colors, fonts, etc.) Both gfx and widget are system specific.

If you want to take a look at the code underlying these structures, the code for the Content Model can be found in /mozilla/content, for the Frame Constructor, CSS, and Reflow in /mozilla/layout, for the View Manager in /mozilla/view, and for the DOM APIs in /mozilla/dom.