Embedding gecko/xulrunner into your app.

This article describes some steps in using the gecko/xulrunner code as an embedded browser in your C++ project. It is of most use for those who own the website for which you are displaying pages, and wish to communicate back and forth between the hosting C++ app and JavaScript running on the web page.

The Mozilla project has some sample code at https://wiki.mozilla.org/Embedding/NewApi that is the basis for this code. Our new code just has a couple of small changes that are useful when embedding xulrunner into a C++ application for which you intend to make an installer.

First, clone the source code from the github repo.

```
git clone git@github.com:gistinc/Gecko.git geckosample
```

The gecko subdirectory contains a VS2008 project that can be built as a static lib.

The mozembed directory is the original sample code from the Mozilla project. The differences from the mozembed project are in building as a static lib for Visual Studio and to specify the path rather than relying on registration. This eases the use embedded in an application, since we no longer have to explicitly register the xulrunner binaries when installing, and can simply refer to the path to them during initialization.

GeckoHost.cpp is sample code containing a simple MFC dialog that will host xulrunner.

GeckoListerner.cpp is a sample of where we listen to event notifications coming from any JavaScript running on the web pages we're displaying.

Usage

If you are hosting xulrunner as a generic web browser component, then I'd recommend that you make the GeckoHost class inherit from MozViewListener to receive notifications concerned with navigating to a webpage. Eg.

```
Class GeckoHost : public CDialog, MozViewListener
```

For Gist, we are narrowly targeting the embedded browser to display Gist pages only, so that we more seamlessly integrate web content with our desktop application. We aren't concerned with navigation to other sites, and so don't need the standard navigation controls nor notifications.

What we do need, though, is a way to communicate between our embedding host (C++ code), and the JavaScript that is running on the page. Due to our use of Ajax, updating the data displayed on the web page is a call from the C++ host to the JavaScript, which then uses Ajax to retrieve data from the server.

Communication from the embedding host to JavaScript running on the web page is easy enough, since we have the EvaluateJavascript method available:

```
                CStringA strCall = "myJavascriptFunction(" +
ToUTF8(strMyJSONparams) + ");";
                char * pszResult = m_pmozView-
>EvaluateJavaScript(strCall);
                delete [] pszResult;
```

This will call the JavaScript function named 'myJavascriptFunction' passing in a block of JSON as an actual JSON param. This is probably the most useful way to call the function. You can pass strings just by putting quotes around any parameters:

```
char * pszResult = m_pmozview-
>EvaluateJavaScript("myJavascriptStringFunction('quoted
string');");
delete [] pszResult;
```

On the JavaScript side:

```
function myJavascriptFunction(json) {
…
}
```

It's harder to have any JavaScript on the web page call back to the host, but we can use a custom event to do it:

```
var node = document.createTextNode("JSON goes here");
document.documentEleemnt.appendChild(node);
var ev = document.createEvent("Events");
ev.initEvent("MyEvent1", true, false);
node.dispatchEvent(ev);
```

This will create a text node, and fire an event on it. On the hosting side, we need to listen to that event and then walk the DOM to find the data we wish to read.

If we need multiple parameters, then create something like:

```
 <div>
     <div>json 1</div>
     <div>json 2</div>
</div>
```

This would be (in JavaScript):

```
var divElement = document.createElement("div");
var node1 = document.createTextNode("JSON parameter 1");
var node2 = document.createTextNode("JSON parameter 2");
divElement.appendChild(node1);
divElement.appendChild(node2);
document.documentElement.appendChild(divElement);
var ev = document.createEvent("Events");
```

```
ev.initEvent("MyEvent2", true, false);
divElement.dispatchEvent(ev);
```

On the hosting side, we need to explicitly listen to the custom events:

```
    nsCOMPtr<nsIDOMEventTarget> eventTarget;
    nsCOMPtr<nsIDOMWindow2> window2 = pMozView->GetDOMWindow();
    if (window2)
    {
        window2->GetWindowRoot(getter_AddRefs(eventTarget));
        if (eventTarget)
        {
            nsCOMPtr<nsIDOMNSEventTarget>
nsEventTarget(do_QueryInterface(eventTarget));

            if (nsEventTarget)
            {
                //TODO: Use an array!
                nsEventTarget-
>AddEventListener(NS_LITERAL_STRING("MyEvent"),
                                                  this,
                                                  PR_FALSE,
                                                  PR_TRUE);
                nsEventTarget-
>AddEventListener(NS_LITERAL_STRING("MyEvent2"),
                                                  this,
                                                  PR_FALSE,
                                                  PR_TRUE);
```

We use the nsIDOMNSEventTarget rather than the nsIDOMEventTarget, because we need the fourth parameter of the AddEventListener method, which specifies that we are going to listen to custom events. This is potentially unsafe for a generic web browser, but for our constrained needs (only displaying Gist web pages), this allows us to received notification events from the JavaScript running on the web page.

Our event handling code:

```
    nsCOMPtr<nsIDOMEventTarget> target;
    if (pEvent != NULL)
    {
           nsString type;
           pEvent->GetType(type);
           if (type.Compare(NS_LITERAL_STRING("MyEvent1") == 0)
           {
                m_pHost->HandleEvent1();
           }
           else if (type.Compare(NS_LITERAL_STRING("MyEvent2") ==
0)
           {
            // Format of the nodes that we will receive the event
is as follows:
            //
            //<div>
```

```
            //  <div>json 1</div>
            //  <div>json 2</div>
            //</div>

                // Get the original event target
                pEvent->GetTarget(getter_AddRefs(target));
                if (target)
                {
                nsCOMPtr<nsIDOMNode>
nsdomRoot(do_QueryInterface(target));
                if (nsdomRoot)
                {
                    nsCOMPtr<nsIDOMNode> nsJSON1Root;

                    // First child will contain a textnode with
the JSON 1
                    nsdomRoot-
>GetFirstChild(getter_AddRefs(nsJSON1Root));
                    if (nsJSON1Root)
                    {
                        nsCOMPtr<nsIDOMNode> nsJSON1;
                        nsCOMPtr<nsIDOMNode> nsJSON2Root;
                                nsString JSON1;
                                nsString JSON2;

                        // This child node is the actual text
node for JSON 1.
                        nsJSON1Root-
>GetFirstChild(getter_AddRefs(nsJSON1));
                        if (nsJSON1)
                        {
                            nsJSON1->GetNodeValue(JSON1);
                        }

                        // Sibling of the JSON1Root will be the
<div> node for JSON2
                        nsJSON1Root-
>GetNextSibling(getter_AddRefs(nsJSON2Root));
                        if (nsJSON2Root)
                        {
                            nsCOMPtr<nsIDOMNode> nsJSON2;

                            // This child node is the actual text
node for JSON 2.
                            nsJSON2Root-
>GetFirstChild(getter_AddRefs(nsJSON2));
                            if (nsJSON2)
                            {
                                nsJSON2->GetNodeValue(JSON2);
                            }
                        }
```

```
                                        m_pHost-
>HandleMyEvent2(JSON1.get(), JSON2.get());
                        }
                    }
                }
```

Initialization

I'd recommend you do this on your main thread. There are warnings in Mozilla forums about using the main thread for this, but I haven't checked the code to verify exactly why.

```
void InitXul()
{
      if (g_mozApp == NULL)
      {
            char szXulPath[] = "<path to the installed location of
the xulrunner binaries>";
            if (SUCCEEDED(SHGetFolderPathA(NULL,
CSIDL_LOCAL_APPDATA | CSIDL_FLAG_CREATE, NULL, SHGFP_TYPE_CURRENT,
szAppData)))
            {
                  PathAppendA(szAppData,
"MyEmbeddedXulProfileDirectoryName");
                  if (!CreateDirectoryA(szAppData, NULL))
                  {
                        if (GetLastError() != ERROR_ALREADY_EXISTS)
                        {
                              log.Trace("Failed to create/open
directory %S, error %d", szAppData, GetLastError());
                        }
                  }
            }

            g_mozApp = new MozApp(szAppData, szXulPath);
      }
```

Since GeckoHost is an MFC dialog, we can create it simply:

```
            mydialog = new GeckoHost(this /*CWnd*/);
            mydialog->Create(IDD_GECKOHOST, this);
```

It will create the embedded web browser during the OnInitDialog:

```
      GetClientRect(&rcClient);
      CHRg(InitXul(), "InitXul");
      if (m_pmozView == NULL)
      {
            m_pmozView = new MozView();
            m_pGeckoListener = new GeckoListener();
      }

      m_pmozView->CreateBrowser(m_hWnd, rcClient.left,
rcClient.top,
```

```
            rcClient.right - rcClient.left, rcClient.bottom -
rcClient.top);
      m_pmozView->LoadURI(strURL8);
      m_pGeckoListener->Init(this, m_pmozView);
```

It's important to call the GeckoListener::Init function after calling the LoadURI, to ensure that enough of the DOM is loaded before we register the event listeners.