file BrowserHost.cpp

BrowserHost.cpp

```
/**********************************************************\
Original Author: Richard Bateman (taxilian)

Created: Jan 23, 2010
License: Dual license model; choose one of two:
New BSD License
http://www.opensource.org/licenses/bsd-license.php
- or -
GNU Lesser General Public License, version 2.1

http://www.gnu.org/licenses/lgpl-2.1.html

Copyright 2009 Richard Bateman, Firebreath development team
\**********************************************************/

#include <cstdio>
#include <cassert>
#include <algorithm>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
#include <boost/lambda/construct.hpp>
#include <boost/format.hpp>
#include <boost/foreach.hpp>
#include <boost/smart_ptr/enable_shared_from_this.hpp>
#include "JSObject.h"
#include "DOM/Window.h"
#include "variant_list.h"
#include "logging.h"
#include "../PluginCore/BrowserStreamManager.h"
#include "precompiled_headers.h" // On windows, everything above this line in PCH

#include "BrowserHost.h"
#include "BrowserStreamRequest.h"
#include "SystemProxyDetector.h"
#include "SimpleStreamHelper.h"

// This is used to keep async calls from
// crashing the browser on shutdown

namespace FB {
 struct _asyncCallData : boost::noncopyable {
 _asyncCallData(void (*func)(void*), void* userData, int id, AsyncCallManagerPtr mgr)
 : func(func), userData(userData), uniqId(id), called(false), mgr(mgr)
 {}
 void call();
 void (*func)(void *);
 void *userData;
 int uniqId;
 bool called;
 AsyncCallManagerWeakPtr mgr;
 };

 class AsyncCallManager : public boost::enable_shared_from_this<AsyncCallManager>, boost::noncopyable {
 public:
 int lastId;
 AsyncCallManager() : lastId(1) {}
 ~AsyncCallManager();

 boost::recursive_mutex m_mutex;
 void shutdown();

 _asyncCallData* makeCallback(void (*func)(void *), void * userData );
 void call( _asyncCallData* data );
 void remove( _asyncCallData* data );

 std::set<_asyncCallData*> DataList;
 std::set<_asyncCallData*> canceledDataList;
 };
}

volatile int FB::BrowserHost::InstanceCount(0);

```

```
75 FB::BrowserHost::BrowserHost()
76  : _asyncManager(boost::make_shared<AsyncCallManager>()), m_threadId(boost::this_thread::get_id()),
77  m_isShutDown(false), m_streamMgr(boost::make_shared<FB::BrowserStreamManager>()), m_htmlLogEnabled
(true)
78 {
79  ++InstanceCount;
80 }
81
82 FB::BrowserHost::~BrowserHost()
83 {
84  --InstanceCount;
85 }
86
87 void FB::BrowserHost::shutdown()
88 {
89  BOOST_FOREACH(FB::JSAPIPtr ptr, m_retainedObjects) {
90  // Notify each JSAPI object that we're shutting down
91  ptr->shutdown();
92  }
93  freeRetainedObjects();
94  boost::upgrade_lock<boost::shared_mutex> _l(m_xtmutex);
95  m_isShutDown = true;
96  _asyncManager->shutdown();
97  m_streamMgr.reset();
98 }
99
100 void FB::BrowserHost::htmlLog(const std::string& str)
101 {
102  FBLOG_INFO("BrowserHost", "Logging to HTML: " << str);
103  if (m_htmlLogEnabled) {
104  try {
105  this->ScheduleAsyncCall(&FB::BrowserHost::AsyncHtmlLog,
106  new FB::AsyncLogRequest(shared_from_this(), str));
107  } catch (const std::exception&) {
108  // This fails during shutdown; ignore it
109  }
110  }
111 }
112
113 void FB::BrowserHost::AsyncHtmlLog(void *logReq)
114 {
115  FB::AsyncLogRequest *req = (FB::AsyncLogRequest*)logReq;
116  try {
117  FB::DOM::WindowPtr window = req->m_host->getDOMWindow();
118
119  if (window && window->getJSObject()->HasProperty("console")) {
120  FB::JSObjectPtr obj = window->getProperty<FB::JSObjectPtr>("console");
121  printf("Logging: %s\n", req->m_msg.c_str());
122  if (obj)
123  obj->Invoke("log", FB::variant_list_of(req->m_msg));
124  }
125  } catch (const std::exception &) {
126  // printf("Exception: %s\n", e.what());
127  // Fail silently; logging should not require success.
128  FBLOG_TRACE("BrowserHost", "Logging to browser console failed");
129  return;
130  }
131  delete req;
132 }
133
134 void FB::BrowserHost::evaluateJavaScript(const std::wstring &script)
135 {
136  evaluateJavaScript(FB::wstring_to_utf8(script));
137 }
138
139 void FB::BrowserHost::initJS(const void* inst)
140 {
141  assertMainThread();
142  // Inject javascript helper function into the page; this is neccesary to help
143  // with some browser compatibility issues.
144
145  const char* javascriptMethod =
146  "window.__FB_CALL_%1% = "
147  "function(delay, f, args, fname) {"
148  " if (arguments.length == 3)"
149  " return setTimeout(function() { f.apply(null, args); }, delay);"
150  " else"
151  " return setTimeout(function() { f[fname].apply(f, args); }, delay);"
152  "};";
153
```

```
154  // hash pointer to get a unique key for this plugin instance
155  std::size_t inst_key = static_cast<std::size_t>(
156  reinterpret_cast<std::ptrdiff_t>(inst));
157  inst_key += (inst_key >> 3);
158
159  unique_key = boost::lexical_cast<std::string>(inst_key);
160
161  call_delegate = (boost::format("__FB_CALL_%1%") % inst_key).str();
162
163  evaluateJavaScript((boost::format(javascriptMethod) % inst_key).str());
164 }
165
166 int FB::BrowserHost::delayedInvoke(const int delayms, const FB::JSObjectPtr& func,
167  const FB::VariantList& args, const std::string& fname)
168 {
169  assertMainThread();
170  FB::JSObjectPtr delegate = getDelayedInvokeDelegate();
171  if (!delegate)
172  return -1; // this is wrong (the return is meant to be the result of setTimeout)
173  if (fname.empty())
174  return delegate->Invoke("", FB::variant_list_of(delayms)(func)(args)).convert_cast<int>();
175  else
176  return delegate->Invoke("", FB::variant_list_of(delayms)(func)(args)(fname)).convert_cast<int>();
177 }
178
179 FB::JSObjectPtr FB::BrowserHost::getDelayedInvokeDelegate() {
180  FB::DOM::WindowPtr win(getDOMWindow());
181  if (win) {
182  if (call_delegate.empty()) {
183  initJS(this);
184  }
185  FB::JSObjectPtr delegate;
186  try {
187  delegate = win->getProperty<FB::JSObjectPtr>(call_delegate);
188  } catch (const FB::script_error&) {
189  }
190  if (!delegate) {
191  // Sometimes the first try doesn't work; for some reason retrying generally does,
192  // and from then on it works fine
193  initJS(this);
194  delegate = win->getProperty<FB::JSObjectPtr>(call_delegate);
195  }
196  return delegate;
197  }
198  return FB::JSObjectPtr();
199 }
200
201 FB::DOM::WindowPtr FB::BrowserHost::_createWindow(const FB::JSObjectPtr& obj) const
202 {
203  return FB::DOM::WindowPtr(new FB::DOM::Window(obj));
204 }
205
206 FB::DOM::DocumentPtr FB::BrowserHost::_createDocument(const FB::JSObjectPtr& obj) const
207 {
208  return FB::DOM::DocumentPtr(new FB::DOM::Document(obj));
209 }
210
211 FB::DOM::ElementPtr FB::BrowserHost::_createElement(const FB::JSObjectPtr& obj) const
212 {
213  return FB::DOM::ElementPtr(new FB::DOM::Element(obj));
214 }
215
216 FB::DOM::NodePtr FB::BrowserHost::_createNode(const FB::JSObjectPtr& obj) const
217 {
218  return FB::DOM::NodePtr(new FB::DOM::Node(obj));
219 }
220
221 void FB::BrowserHost::assertMainThread() const
222 {
223 #ifdef _DEBUG
224  if (!isMainThread()) {
225  FBLOG_FATAL("BrowserHost", "Trying to call something from the wrong thread!");
226  }
227  assert(isMainThread());
228 #endif
229 }
230
231 bool FB::BrowserHost::isMainThread() const
232 {
233  return m_threadId == boost::this_thread::get_id();
```

```
234 }
235
236 void FB::BrowserHost::freeRetainedObjects() const
237 {
238  boost::recursive_mutex::scoped_lock _l(m_jsapimutex);
239  // This releases all stored shared_ptr objects that the browser is holding
240  m_retainedObjects.clear();
241
242  // This allows the browserhost to release any browser objects that were held by the retained
243  // objects
244  DoDeferredRelease();
245 }
246
247 void FB::BrowserHost::retainJSAPIPtr( const FB::JSAPIPtr& obj ) const
248 {
249  boost::recursive_mutex::scoped_lock _l(m_jsapimutex);
250  m_retainedObjects.push_back(obj);
251 }
252
253 void FB::BrowserHost::releaseJSAPIPtr( const FB::JSAPIPtr& obj ) const
254 {
255  boost::recursive_mutex::scoped_lock _l(m_jsapimutex);
256  std::list<FB::JSAPIPtr>::iterator it = std::find_if(m_retainedObjects.begin(), m_retainedObjects.
end(), boost::lambda::_1 == obj);
257  if (it != m_retainedObjects.end()) {
258  m_retainedObjects.erase(it);
259  }
260
261  if (isMainThread())
262  DoDeferredRelease();
263 }
264
265 void FB::_asyncCallData::call()
266 {
267  if (func) {
268  void (*f)(void *) = func;
269  func = NULL;
270  called = true;
271  f(userData);
272  }
273 }
274
275
276 void FB::AsyncCallManager::call( _asyncCallData* data )
277 {
278  {
279  // Verify _asyncCallData is still in DataList. If not, the _asyncCallData has already been dealt
with.
280  boost::recursive_mutex::scoped_lock _l(m_mutex);
281  std::set<_asyncCallData*>::iterator fnd = DataList.find(data);
282  if (DataList.end() != fnd)
283  DataList.erase(fnd);
284  else
285  data = NULL;
286  }
287  if (data) {
288  data->call();
289  delete data;
290  }
291 }
292
293 FB::_asyncCallData* FB::AsyncCallManager::makeCallback(void (*func)(void *), void * userData)
294 {
295  boost::recursive_mutex::scoped_lock _l(m_mutex);
296  _asyncCallData *data = new _asyncCallData(func, userData, ++lastId, shared_from_this());
297  DataList.insert(data);
298  return data;
299 }
300
301 void FB::AsyncCallManager::remove(_asyncCallData* data)
302 {
303  boost::recursive_mutex::scoped_lock _l(m_mutex);
304  DataList.erase(data);
305 }
306
307 void FB::AsyncCallManager::shutdown()
308 {
309  boost::recursive_mutex::scoped_lock _l(m_mutex);
310  // Store these so that they can be freed when the browserhost object is destroyed -- at that
311  // point it's no longer possible for the browser to finish the async calls
```

```
312  canceledDataList.insert(DataList.begin(), DataList.end());
313
314  std::for_each(DataList.begin(), DataList.end(), boost::lambda::bind(&_asyncCallData::call, boost::
lambda::_1));
315  DataList.clear();
316 }
317
318 FB::AsyncCallManager::~AsyncCallManager()
319 {
320  std::for_each(canceledDataList.begin(), canceledDataList.end(), boost::lambda::bind(boost::lambda::
delete_ptr(), boost::lambda::_1));
321 }
322
323
324 void asyncCallWrapper(void *userData)
325 {
326  // Verify AsyncCallManager still exists. If not, the _asyncCallData has already been dealt with.
327  FB::_asyncCallData* data(static_cast<FB::_asyncCallData*>(userData));
328  FB::AsyncCallManagerPtr ptr(data->mgr.lock());
329  if (ptr) {
330  ptr->call(data);
331  }
332 }
333
334 bool FB::BrowserHost::ScheduleAsyncCall( void (*func)(void *), void *userData ) const
335 {
336  if (isShutDown()) {
337  return false;
338  } else {
339  _asyncCallData* data = _asyncManager->makeCallback(func, userData);
340  bool result = _scheduleAsyncCall(&asyncCallWrapper, data);
341  if (!result) {
342  _asyncManager->remove(data);
343  }
344  return result;
345  }
346 }
347
348 FB::BrowserStreamPtr FB::BrowserHost::createStream( const std::string& url,
349  const PluginEventSinkPtr& callback, bool cache /*= true*/, bool seekable /*= false*/,
350  size_t internalBufferSize /*= 128 * 1024 */ ) const
351 {
352  BrowserStreamRequest req(url, "GET");
353  req.setEventSink(callback);
354  req.setCacheable(cache);
355  req.setSeekable(seekable);
356  req.setBufferSize(internalBufferSize);
357  return createStream(req);
358 }
359
360 FB::BrowserStreamPtr FB::BrowserHost::createStream( const BrowserStreamRequest& req, const bool
enable_async ) const
361 {
362  assertMainThread();
363  if (enable_async && req.getCallback() && !req.getEventSink()) {
364  // If a callback was provided, use SimpleStreamHelper to create it;
365  // This will actually call back into this function with an event sink
366  BrowserStreamRequest newReq(req);
367  FB::SimpleStreamHelperPtr asyncPtr(SimpleStreamHelper::AsyncRequest(shared_from_this(), req));
368  return asyncPtr->getStream();
369  } else { // Create the stream with the EventSink
370  FB::BrowserStreamPtr ptr(_createStream(req));
371  if (ptr) {
372  m_streamMgr->retainStream(ptr);
373  }
374  return ptr;
375  }
376 }
377
378 FB::BrowserStreamPtr FB::BrowserHost::createPostStream( const std::string& url,
379  const PluginEventSinkPtr& callback, const std::string& postdata, bool cache /*= true*/,
380  bool seekable /*= false*/, size_t internalBufferSize /*= 128 * 1024 */ ) const
381 {
382  BrowserStreamRequest req(url, "POST");
383  req.setEventSink(callback);
384  req.setCacheable(cache);
385  req.setSeekable(seekable);
386  req.setBufferSize(internalBufferSize);
387  req.setPostData(postdata);
388  return createStream(req);
```

```
389 }
390
391 FB::BrowserStreamPtr FB::BrowserHost::createUnsolicitedStream( const BrowserStreamRequest& req )
const
392 {
393  assertMainThread();
394  FB::BrowserStreamPtr ptr(_createUnsolicitedStream(req));
395  if (ptr) {
396  m_streamMgr->retainStream(ptr);
397  }
398  return ptr;
399 }
400
401 bool FB::BrowserHost::DetectProxySettings( std::map<std::string, std::string>& settingsMap, const
std::string& url )
402 {
403  return FB::SystemProxyDetector::get()->detectProxy(settingsMap, url);
404 }
```

FB::DOM::Node
DOM Node wrapper.
Definition: ScriptingCore/DOM/Node.h:39
FB::BrowserHost::AsyncHtmlLog
static void AsyncHtmlLog(void *data)
Don't call this; it is a helper function used by htmlLog.
Definition: BrowserHost.cpp:113
FB::AsyncLogRequest
This class is used by BrowserHost for the BrowserHost::AsyncHtmlLog method.
Definition: BrowserHost.h:46
FB::JSObjectPtr
boost::shared_ptr< FB::JSObject > JSObjectPtr
Defines an alias representing a JSObject shared_ptr (you should never use a JSObject* directly) ...
Definition: APITypes.h:109
FB::SimpleStreamHelper::AsyncRequest
static FB::SimpleStreamHelperPtr AsyncRequest(const BrowserHostConstPtr &host, const BrowserStreamRequest
&req)
Creates an asynchronous HTTP request from the provided BrowserStreamRequest.
FB::BrowserHost::retainJSAPIPtr
void retainJSAPIPtr(const FB::JSAPIPtr &obj) const
retains an instance of the JSAPI object until the plugin shuts down
Definition: BrowserHost.cpp:247
FB::BrowserStreamRequest::setBufferSize
void setBufferSize(size_t size)
Call this to indicate the preferred internal buffer size of the BrowserStream object.
Definition: BrowserStreamRequest.h:169
FB::BrowserHost::releaseJSAPIPtr
void releaseJSAPIPtr(const FB::JSAPIPtr &obj) const
releases the specified JSAPI object to allow it to be invalidated and freed. This is done automatical...
Definition: BrowserHost.cpp:253
FB::variant_list_of
FB::detail::VariantListInserter variant_list_of(FB::variant v)
Allows convenient creation of an FB::VariantList.
Definition: variant_list.h:122
FB::DOM::NodePtr
boost::shared_ptr< Node > NodePtr
shared_ptr for a FB::DOM::Node
Definition: ScriptingCore/DOM/Node.h:29
FB::VariantList
std::vector< variant > VariantList
Defines an alias representing list of variants.
Definition: APITypes.h:64
FB::BrowserStreamRequest::setSeekable
void setSeekable(bool s)
Call with true to request that the stream be seekable; default is false.
Definition: BrowserStreamRequest.h:104
FB::DOM::WindowPtr
boost::shared_ptr< Window > WindowPtr
shared_ptr for a FB::DOM::Window
Definition: ScriptingCore/DOM/Window.h:40
FB::DOM::DocumentPtr
boost::shared_ptr< Document > DocumentPtr
shared_ptr for a FB::DOM::Document
Definition: ScriptingCore/DOM/Document.h:25
FB::BrowserHost::DetectProxySettings
virtual bool DetectProxySettings(std::map< std::string, std::string > &settingsMap, const std::string
&url="")
Detects the proxy settings from the browser.
Definition: BrowserHost.cpp:401
FB::BrowserHost::isMainThread
bool isMainThread() const
Query if this object is on the main thread.

Definition: BrowserHost.cpp:231
FB::BrowserStreamRequest
Information about an HTTP request to be made.
Definition: BrowserStreamRequest.h:38
FB::JSAPIPtr
boost::shared_ptr< FB::JSAPI > JSAPIPtr
Defines an alias for a JSAPI shared_ptr (you should never use a JSAPI* directly)
Definition: APITypes.h:94
FB::script_error
Exception type; when thrown in a JSAPI method, a javascript exception will be thrown.
Definition: JSExceptions.h:28
FB::BrowserHost::createUnsolicitedStream
virtual BrowserStreamPtr createUnsolicitedStream(const BrowserStreamRequest &req) const
Used internally to create a BrowserStream to handle an unsolicited NPP_NewStream. ...
Definition: BrowserHost.cpp:391
FB::BrowserStreamRequest::setEventSink
void setEventSink(const PluginEventSinkPtr &ptr)
To use a PluginEventSink such as DefaultBrowserStreamHandler or a derivative call this method to spec...
Definition: BrowserStreamRequest.h:187
FB::DOM::Window
DOM Window abstraction for manipulating and accessing the javascript window object that the plugin is...
Definition: ScriptingCore/DOM/Window.h:51
FB::wstring_to_utf8
std::string wstring_to_utf8(const std::wstring &src)
Accepts a std::wstring and returns a UTF8-encoded std::string.
Definition: utf8_tools.cpp:37
FB::BrowserHost::htmlLog
virtual void htmlLog(const std::string &str)
Sends a log message to the containing web page using Console.log (firebug)
Definition: BrowserHost.cpp:100
FB::BrowserHost::delayedInvoke
int delayedInvoke(const int delayms, const FB::JSObjectPtr &func, const FB::VariantList &args, const std::
string &fname="")
Executes the provided method object after a delay using window.setTimeout.
Definition: BrowserHost.cpp:166
FB::DOM::Element
DOM Element wrapper.
Definition: ScriptingCore/DOM/Element.h:35
FB::BrowserStreamRequest::getCallback
HttpCallback getCallback() const
Returns the HttpCallback functor assigned to the object, or a NULL HttpCallback if none...
Definition: BrowserStreamRequest.h:225
FB::BrowserHost::shutdown
virtual void shutdown()
Notifies the browserhost object that the associated plugin object is shutting down.
Definition: BrowserHost.cpp:87
FB::BrowserHost::~BrowserHost
virtual ~BrowserHost()
Finaliser.
Definition: BrowserHost.cpp:82
FB::BrowserHost::assertMainThread
void assertMainThread() const
When running in debug mode, asserts that the call is made on the main thread.
Definition: BrowserHost.cpp:221
FB::BrowserHost::evaluateJavaScript
virtual void evaluateJavaScript(const std::string &script)=0
Evaluates arbitrary javascript; note that it does not return the result due to cross- browser compati...
Definition: BrowserHost.h:334
FB::BrowserStreamRequest::getEventSink
PluginEventSinkPtr getEventSink() const
Returns the PluginEventSink assigned to be the observer for the BrowserStream, or a NULL BrowserStrea...
Definition: BrowserStreamRequest.h:196
FB::BrowserHost::createPostStream
virtual BrowserStreamPtr createPostStream(const std::string &url, const PluginEventSinkPtr &callback,
const std::string &postdata, bool cache=true, bool seekable=false, size_t internalBufferSize=128 *1024)
const
Creates a BrowserStream.
Definition: BrowserHost.cpp:378
FB::BrowserHost::BrowserHost
BrowserHost()
Default constructor.
Definition: BrowserHost.cpp:75
FB::BrowserHost::freeRetainedObjects
void freeRetainedObjects() const
releases all JSAPI objects that have been passed to the browser
Definition: BrowserHost.cpp:236
FB::DOM::ElementPtr
boost::shared_ptr< Element > ElementPtr
shared_ptr for a FB::DOM::Element
Definition: ScriptingCore/DOM/Element.h:25

FB::BrowserStreamRequest::setCacheable
void setCacheable(bool c)
Call with true to indicate that the browser's cache may be used for this request; default is false...
Definition: BrowserStreamRequest.h:115
FB::BrowserHost::ScheduleAsyncCall
bool ScheduleAsyncCall(void(*func)(void *), void *userData) const
Schedule asynchronous call to be executed on the main thread.
Definition: BrowserHost.cpp:334
FB::DOM::Document
Abstraction for accessing and manipulating a DOM Document.
Definition: ScriptingCore/DOM/Document.h:38
