file JSAPIAuto.cpp

JSAPIAuto.cpp

```
/**********************************************************\
Original Author: Georg Fritzsche

Created: November 7, 2009
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 Georg Fritzsche, Firebreath development team
\**********************************************************/

#include "utf8_tools.h"
#include "boost/thread/mutex.hpp"
#include "boost/make_shared.hpp"
#include "JSFunction.h"
#include "JSEvent.h"
#include <cassert>
#include "precompiled_headers.h" // On windows, everything above this line in PCH

#include "JSAPIAuto.h"

bool FB::JSAPIAuto::s_allowDynamicAttributes = true;
bool FB::JSAPIAuto::s_allowRemoveProperties = false;
bool FB::JSAPIAuto::s_allowMethodObjects = true;

FB::JSAPIAuto::JSAPIAuto(const std::string& description)
  : FB::JSAPIImpl(SecurityScope_Public),
  m_description(description),
  m_allowDynamicAttributes(FB::JSAPIAuto::s_allowDynamicAttributes),
  m_allowRemoveProperties(FB::JSAPIAuto::s_allowRemoveProperties),
  m_allowMethodObjects(FB::JSAPIAuto::s_allowMethodObjects)
{
  init();
}

FB::JSAPIAuto::JSAPIAuto( const SecurityZone& securityLevel, const std::string& description /*=
"<JSAPI-Auto Secure Javascript Object>"*/ )
  : FB::JSAPIImpl(securityLevel),
  m_description(description),
  m_allowDynamicAttributes(FB::JSAPIAuto::s_allowDynamicAttributes),
  m_allowRemoveProperties(FB::JSAPIAuto::s_allowRemoveProperties),
  m_allowMethodObjects(FB::JSAPIAuto::s_allowMethodObjects)
{
  init();
}

void FB::JSAPIAuto::init( )
{
  {
  scoped_zonelock _l(this, SecurityScope_Public);
  registerMethod("toString", make_method(this, &JSAPIAuto::ToString));
  registerMethod("getAttribute", make_method(this, &JSAPIAuto::getAttribute));
  registerMethod("setAttribute", make_method(this, &JSAPIAuto::setAttribute));

  registerProperty("value", make_property(this, &JSAPIAuto::ToString));
  registerProperty("valid", make_property(this, &JSAPIAuto::get_valid));
  }

  setReserved("offsetWidth");
  setReserved("offsetHeight");
  setReserved("width");
  setReserved("height");
  setReserved("attributes");
  setReserved("nodeType");
  setReserved("namespaceURI");
  setReserved("localName");
  setReserved("wrappedJSObject");
  setReserved("prototype");
  setReserved("style");
```

```
72  setReserved("id");
73  setReserved("constructor");
74  setReserved("nodeName");
75  setReserved("hasAttribute");
76 }
77
78 FB::JSAPIAuto::~JSAPIAuto()
79 {
80
81 }
82
83 void FB::JSAPIAuto::registerMethod(const std::string& name, const CallMethodFunctor& func)
84 {
85  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
86  m_methodFunctorMap[name] = func;
87  m_zoneMap[name] = getZone();
88 }
89
90 void FB::JSAPIAuto::unregisterMethod( const std::string& name )
91 {
92  FB::MethodFunctorMap::iterator fnd = m_methodFunctorMap.find(name);
93  if (fnd != m_methodFunctorMap.end()) {
94  m_methodFunctorMap.erase(name);
95  m_zoneMap.erase(name);
96  }
97 }
98
99 void FB::JSAPIAuto::registerProperty(const std::wstring& name, const PropertyFunctors& func)
100 {
101  registerProperty(FB::wstring_to_utf8(name), func);
102 }
103
104 void FB::JSAPIAuto::registerProperty(const std::string& name, const PropertyFunctors& propFuncs)
105 {
106  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
107  m_propertyFunctorsMap[name] = propFuncs;
108  m_zoneMap[name] = getZone();
109 }
110
111 void FB::JSAPIAuto::unregisterProperty( const std::wstring& name )
112 {
113  unregisterProperty(FB::wstring_to_utf8(name));
114 }
115
116 void FB::JSAPIAuto::unregisterProperty( const std::string& name )
117 {
118  FB::PropertyFunctorsMap::iterator fnd = m_propertyFunctorsMap.find(name);
119  if (fnd != m_propertyFunctorsMap.end()) {
120  m_propertyFunctorsMap.erase(name);
121  m_zoneMap.erase(name);
122  }
123 }
124
125 void FB::JSAPIAuto::getMemberNames(std::vector<std::string> &nameVector) const
126 {
127  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
128  nameVector.clear();
129  for (ZoneMap::const_iterator it = m_zoneMap.begin(); it != m_zoneMap.end(); ++it) {
130  if (getZone() >= it->second)
131  nameVector.push_back(it->first);
132  }
133 }
134
135 size_t FB::JSAPIAuto::getMemberCount() const
136 {
137  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
138  size_t count = 0;
139  for (ZoneMap::const_iterator it = m_zoneMap.begin(); it != m_zoneMap.end(); ++it) {
140  if (getZone() >= it->second)
141  ++count;
142  }
143  return count;
144 }
145
146 bool FB::JSAPIAuto::HasMethod(const std::string& methodName) const
147 {
148  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
149  if(!m_valid)
150  return false;
151
```

```
152  return (m_methodFunctorMap.find(methodName) != m_methodFunctorMap.end()) && memberAccessible
(m_zoneMap.find(methodName));
153 }
154
155 bool FB::JSAPIAuto::HasMethodObject( const std::string& methodObjName ) const
156 {
157  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
158
159  return m_allowMethodObjects && HasMethod(methodObjName);
160 }
161
162 bool FB::JSAPIAuto::HasProperty(const std::string& propertyName) const
163 {
164  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
165  if(!m_valid)
166  return false;
167
168  // To be able to set dynamic properties, we have to respond true always
169  if (m_allowDynamicAttributes && !HasMethod(propertyName) && !isReserved(propertyName))
170  return true;
171  else if (m_allowMethodObjects && HasMethod(propertyName) && memberAccessible(m_zoneMap.find
(propertyName)))
172  return true;
173
174  return m_propertyFunctorsMap.find(propertyName) != m_propertyFunctorsMap.end()
175  || m_attributes.find(propertyName) != m_attributes.end();
176 }
177
178 bool FB::JSAPIAuto::HasProperty(int idx) const
179 {
180  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
181  if(!m_valid)
182  return false;
183
184  // To be able to set dynamic properties, we have to respond true always
185  if (m_allowDynamicAttributes)
186  return true;
187
188  return m_attributes.find(boost::lexical_cast<std::string>(idx)) != m_attributes.end();
189 }
190
191 FB::variant FB::JSAPIAuto::GetProperty(const std::string& propertyName)
192 {
193  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
194  if(!m_valid)
195  throw object_invalidated();
196
197  ZoneMap::const_iterator zoneName = m_zoneMap.find(propertyName);
198  PropertyFunctorsMap::const_iterator it = m_propertyFunctorsMap.find(propertyName);
199  if(it != m_propertyFunctorsMap.end() && memberAccessible(zoneName)) {
200  return it->second.get();
201  } else if (memberAccessible(zoneName)) {
202  if (HasMethodObject(propertyName))
203  return GetMethodObject(propertyName);
204
205  AttributeMap::iterator fnd = m_attributes.find(propertyName);
206  if (fnd != m_attributes.end())
207  return fnd->second.value;
208  else if (m_allowDynamicAttributes) {
209  return FB::FBVoid(); // If we allow dynamic attributes then we need to
210  // return void if the property doesn't exist;
211  // otherwise checking a property will throw an exception
212  } else {
213  throw invalid_member(propertyName);
214  }
215  } else {
216  if (m_allowDynamicAttributes) {
217  return FB::FBVoid();
218  } else {
219  throw invalid_member(propertyName);
220  }
221  }
222 }
223
224 void FB::JSAPIAuto::SetProperty(const std::string& propertyName, const variant& value)
225 {
226  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
227  if(!m_valid)
228  throw object_invalidated();
229
```

```
 PropertyFunctorsMap::iterator it = m_propertyFunctorsMap.find(propertyName);
 // Note that if an explicit property exists but is not accessible in the current security context,
 // we throw an exception.
 if(it != m_propertyFunctorsMap.end()) {
 if (memberAccessible(m_zoneMap.find(propertyName))) {
 try {
 it->second.set(value);
 } catch (const FB::bad_variant_cast& ex) {
 std::string errorMsg("Could not convert from ");
 errorMsg += ex.from;
 errorMsg += " to ";
 errorMsg += ex.to;
 throw FB::invalid_arguments(errorMsg);
 }
 } else {
 throw invalid_member(propertyName);
 }
 } else if (m_allowDynamicAttributes || (m_attributes.find(propertyName) != m_attributes.end() && !
m_attributes[propertyName].readonly)) {
 registerAttribute(propertyName, value);
 } else {
 throw invalid_member(propertyName);
 }
}

void FB::JSAPIAuto::RemoveProperty(const std::string& propertyName)
{
 boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
 if(!m_valid)
 throw object_invalidated();

 // If there is nothing with this name available in the current security context,
 // we throw an exception -- whether or not a real property exists
 if (!memberAccessible(m_zoneMap.find(propertyName)))
 throw invalid_member(propertyName);

 if(m_allowRemoveProperties && m_propertyFunctorsMap.find(propertyName) != m_propertyFunctorsMap.
end()) {
 unregisterProperty(propertyName);
 } else if (m_allowDynamicAttributes && m_attributes.find(propertyName) != m_attributes.end()
 && !m_attributes[propertyName].readonly) {
 unregisterAttribute(propertyName);
 }

 // If nothing is found matching, we'll just let it slide -- no sense causing exceptions
 // when the end goal is reached already.
}

FB::variant FB::JSAPIAuto::GetProperty(int idx)
{
 boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
 if(!m_valid)
 throw object_invalidated();

 std::string id = boost::lexical_cast<std::string>(idx);
 AttributeMap::iterator fnd = m_attributes.find(id);
 if (fnd != m_attributes.end() && memberAccessible(m_zoneMap.find(id)))
 return fnd->second.value;
 else if (m_allowDynamicAttributes) {
 return FB::FBVoid(); // If we allow dynamic attributes then we need to
 // return void if the property doesn't exist;
 // otherwise checking a property will throw an exception
 } else {
 throw invalid_member(boost::lexical_cast<std::string>(idx));
 }

 // This method should be overridden to access properties in an array style from javascript,
 // i.e. var value = pluginObj[45]; would call GetProperty(45)
 // Default is to throw "invalid member" unless m_attributes has something matching
}

void FB::JSAPIAuto::SetProperty(int idx, const variant& value)
{
 if (!m_valid)
 throw object_invalidated();

 boost::recursive_mutex::scoped_lock lock(m_zoneMutex);

 std::string id(boost::lexical_cast<std::string>(idx));
```

```
307  if (m_allowDynamicAttributes || (m_attributes.find(id) != m_attributes.end() && !m_attributes[id].
readonly)) {
308  registerAttribute(id, value);
309  } else {
310  throw invalid_member(FB::variant(idx).convert_cast<std::string>());
311  }
312 }
313
314 void FB::JSAPIAuto::RemoveProperty(int idx)
315 {
316  if (!m_valid)
317  throw object_invalidated();
318
319  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
320
321  std::string id(boost::lexical_cast<std::string>(idx));
322  if (m_allowDynamicAttributes && m_attributes.find(id) != m_attributes.end() && !m_attributes[id].
readonly) {
323  unregisterAttribute(id);
324  } else {
325  throw invalid_member(FB::variant(idx).convert_cast<std::string>());
326  }
327 }
328
329 FB::variant FB::JSAPIAuto::Invoke(const std::string& methodName, const std::vector<variant> &args)
330 {
331  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
332  if(!m_valid)
333  throw object_invalidated();
334
335  if (memberAccessible(m_zoneMap.find(methodName))) {
336  try {
337  MethodFunctorMap::iterator it = m_methodFunctorMap.find(methodName);
338  if(it == m_methodFunctorMap.end())
339  throw invalid_member(methodName);
340
341  return it->second.call(args);
342  } catch (const FB::bad_variant_cast& ex) {
343  std::string errorMsg("Could not convert from ");
344  errorMsg += ex.from;
345  errorMsg += " to ";
346  errorMsg += ex.to;
347  throw FB::invalid_arguments(errorMsg);
348  }
349  } else {
350  throw invalid_member(methodName);
351  }
352 }
353
354 FB::variant FB::JSAPIAuto::Construct(const std::vector<variant> &args)
355 {
356  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
357  if(!m_valid)
358  throw object_invalidated();
359
360  throw invalid_member("constructor");
361 }
362
363 FB::JSAPIPtr FB::JSAPIAuto::GetMethodObject( const std::string& methodObjName )
364 {
365  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
366  if(!m_valid)
367  throw object_invalidated();
368
369  if (memberAccessible(m_zoneMap.find(methodObjName)) && HasMethod(methodObjName)) {
370  MethodObjectMap::const_iterator fnd = m_methodObjectMap.find(boost::make_tuple(methodObjName,
getZone()));
371  if (fnd != m_methodObjectMap.end()) {
372  return fnd->second;
373  } else {
374  FB::JSFunctionPtr ptr(boost::make_shared<FB::JSFunction>(shared_from_this(), methodObjName,
getZone()));
375  m_methodObjectMap[boost::make_tuple(methodObjName, getZone())] = ptr;
376  return ptr;
377  }
378  } else {
379  throw invalid_member(methodObjName);
380  }
381 }
382
```

```
383 void FB::JSAPIAuto::registerAttribute( const std::string &name, const FB::variant& value, bool
readonly /*= false*/ )
384 {
385  boost::recursive_mutex::scoped_lock lock(m_zoneMutex);
386  Attribute attr = {value, readonly};
387  m_attributes[name] = attr;
388  m_zoneMap[name] = getZone();
389 }
390
391 void FB::JSAPIAuto::unregisterAttribute( const std::string& name )
392 {
393  AttributeMap::iterator fnd = m_attributes.find(name);
394  if ( fnd != m_attributes.end() ) {
395  if (fnd->second.readonly ) {
396  throw FB::script_error("Cannot remove read-only property " + name);
397  } else {
398  m_attributes.erase(fnd);
399  m_zoneMap.erase(name);
400  }
401  } else {
402  return; // No attribute of that name? success!
403  }
404 }
405
406 FB::variant FB::JSAPIAuto::getAttribute( const std::string& name )
407 {
408  if (m_attributes.find(name) != m_attributes.end()) {
409  return m_attributes[name].value;
410  }
411  return FB::FBVoid();
412 }
413
414 void FB::JSAPIAuto::setAttribute( const std::string& name, const FB::variant& value )
415 {
416  AttributeMap::iterator fnd = m_attributes.find(name);
417  if (fnd == m_attributes.end() || !fnd->second.readonly) {
418  Attribute attr = {value, false};
419  m_attributes[name] = attr;
420  m_zoneMap[name] = getZone();
421  } else {
422  throw FB::script_error("Cannot set read-only property " + name);
423  }
424 }
425
426 void FB::JSAPIAuto::FireJSEvent( const std::string& eventName, const FB::VariantMap &members, const F
B::VariantList &arguments )
427 {
428  JSAPIImpl::FireJSEvent(eventName, members, arguments);
429  FB::variant evt(getAttribute(eventName));
430  if (evt.is_of_type<FB::JSObjectPtr>()) {
431  VariantList args;
432  args.push_back(FB::CreateEvent(shared_from_this(), eventName, members, arguments));
433  try {
434  evt.cast<JSObjectPtr>()->InvokeAsync("", args);
435  } catch (...) {
436  // Apparently either this isn't really an event handler or something failed.
437  }
438  }
439 }
440
441 void FB::JSAPIAuto::fireAsyncEvent( const std::string& eventName, const std::vector<variant>& args )
442 {
443  JSAPIImpl::fireAsyncEvent(eventName, args);
444  FB::variant evt(getAttribute(eventName));
445  if (evt.is_of_type<FB::JSObjectPtr>()) {
446  try {
447  FB::JSObjectPtr handler(evt.cast<JSObjectPtr>());
448  if (handler) {
449  handler->InvokeAsync("", args);
450  }
451  } catch (...) {
452  // Apparently either this isn't really an event handler or something failed.
453  }
454  }
455 }
FB::CallMethodFunctor
boost::function< variant(const std::vector< variant > &)> CallMethodFunctor
Defines an alias representing a method functor used by FB::JSAPIAuto, created by FB::make_method().
Definition: APITypes.h:286
FB::JSAPIImpl
```

JavaScript API base class implementation – provides basic functionality for C++ JSAPI objects...
Definition: JSAPIImpl.h:49
FB::JSAPIAuto::RemoveProperty
virtual void RemoveProperty(const std::string &propertyName)
Removes a property.
Definition: JSAPIAuto.cpp:254
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::JSAPIAuto::ToString
virtual std::string ToString()
Default method called when a string value is requested for the scriptable object. ...
Definition: JSAPIAuto.h:293
FB::make_property
PropertyFunctors make_property(C *instance, F1 getter, F2 setter)
Generate read-write property functors for use with registerProperty() of FB::JSAPIAuto.
Definition: PropertyConverter.h:144
FB::JSAPIAuto::registerAttribute
virtual void registerAttribute(const std::string &name, const FB::variant &value, bool readonly=false)
Registers an attribute name and sets the value to _value. Optionally read-only.
Definition: JSAPIAuto.cpp:383
FB::variant
Accepts any datatype, used in all interactions with javascript. Provides tools for getting back out t...
Definition: variant.h:198
FB::JSAPIAuto::FireJSEvent
virtual void FireJSEvent(const std::string &eventName, const FB::VariantMap &members, const FB::
VariantList &arguments)
Fires an event into javascript asynchronously using a W3C-compliant event parameter.
Definition: JSAPIAuto.cpp:426
FB::invalid_arguments
Thrown by a JSAPI object when the argument(s) provided to a SetProperty or Invoke call are found to b...
Definition: JSExceptions.h:47
FB::object_invalidated
Thrown by a JSAPI object when a call is made on it after the object has been invalidated.
Definition: JSExceptions.h:69
FB::JSAPIAuto::HasMethodObject
virtual bool HasMethodObject(const std::string &methodObjName) const
Query if 'methodObjName' is a valid methodObj.
Definition: JSAPIAuto.cpp:155
FB::JSAPIImpl::FireJSEvent
virtual void FireJSEvent(const std::string &eventName, const FB::VariantMap &members, const FB::
VariantList &arguments)
Fires an event into javascript asynchronously using a W3C-compliant event parameter.
Definition: JSAPIImpl.cpp:185
FB::VariantList
std::vector< variant > VariantList
Defines an alias representing list of variants.
Definition: APITypes.h:64
FB::SecurityZone
int SecurityZone
Used to set a SecurityZone for a method or property – used by JSAPIAuto.
Definition: APITypes.h:275
FB::bad_variant_cast
Thrown when variant::cast<type> or variant::convert_cast<type> fails because the type of the value st...
Definition: variant.h:133
FB::JSAPIAuto::getMemberNames
virtual void getMemberNames(std::vector< std::string > &nameVector) const
Called by the browser to enumerate the members of this JSAPI object.
Definition: JSAPIAuto.cpp:125
FB::JSAPIAuto::HasProperty
virtual bool HasProperty(const std::string &propertyName) const
Query if 'propertyName' is a valid property.
Definition: JSAPIAuto.cpp:162
FB::JSAPIAuto::HasMethod
virtual bool HasMethod(const std::string &methodName) const
Query if the JSAPI object has the 'methodName' method.
Definition: JSAPIAuto.cpp:146
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::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::JSAPIAuto::SetProperty

virtual void SetProperty(const std::string &propertyName, const variant &value)
Sets the value of a property.
Definition: JSAPIAuto.cpp:224
FB::JSAPIAuto::Invoke
virtual variant Invoke(const std::string &methodName, const std::vector< variant > &args)
Called by the browser to invoke a method on the JSAPI object.
Definition: JSAPIAuto.cpp:329
FB::JSAPIAuto::GetMethodObject
virtual JSAPIPtr GetMethodObject(const std::string &methodObjName)
Gets a method object (JSAPI object that has a default method)
Definition: JSAPIAuto.cpp:363
FB::JSAPIAuto::getAttribute
virtual FB::variant getAttribute(const std::string &name)
Returns the attribute with the given name, empty variant if none.
Definition: JSAPIAuto.cpp:406
FB::JSAPIAuto::GetProperty
virtual variant GetProperty(const std::string &propertyName)
Gets a property value.
Definition: JSAPIAuto.cpp:191
FB::JSAPIAuto::get_valid
virtual bool get_valid()
Property exposed by default to javascript useful for checking to make sure that the JSAPI is working...
Definition: JSAPIAuto.h:306
FB::JSAPIAuto::Construct
virtual variant Construct(const std::vector< variant > &args)
Called by the browser to construct the JSAPI object.
Definition: JSAPIAuto.cpp:354
FB::PropertyFunctors
used by FB::JSAPIAuto to store property implementation details, created by FB::make_property().
Definition: APITypes.h:311
FB::invalid_member
Thrown when an Invoke, SetProperty, or GetProperty call is made for a member that is invalid (does no...
Definition: JSExceptions.h:83
FB::VariantMap
std::map< std::string, variant > VariantMap
Defines an alias representing a string -> variant map.
Definition: APITypes.h:72
FB::variant::cast
T cast() const
returns the value cast as the given type; throws bad_variant_type if that type is not the type of the...
Definition: variant.h:370
FB::JSAPIAuto::JSAPIAuto
JSAPIAuto(const std::string &description="<JSAPI-Auto Javascript Object>")
Description is used by ToString().
Definition: JSAPIAuto.cpp:29
FB::JSAPIAuto::setAttribute
virtual void setAttribute(const std::string &name, const FB::variant &value)
Assigns a value to the specified attribute, if it is not reserved or read-only.
Definition: JSAPIAuto.cpp:414
FB::JSAPIAuto::getMemberCount
virtual size_t getMemberCount() const
Gets the member count.
Definition: JSAPIAuto.cpp:135
FB::variant::is_of_type
bool is_of_type() const
Query if this object is of a particular type.
Definition: variant.h:355
FB::JSAPIAuto
JSAPI class with automatic argument type enforcement.
Definition: JSAPIAuto.h:94
