WRAPPER EXTENSIONS

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/guide/wrapper-extensions

An additional feature of Construct's plugin SDK is that it allows bundling a *wrapper extension* for deeper platform integration. This is currently supported with the following exporters:

- Windows WebView2
- Xbox UWP (WebView2)
- macOS WKWebView
- Linux (CEF)

The Windows WebView2 exporter uses a traditional desktop Windows application using low-level Win32 APIs that embeds Microsoft WebView2 to load web content. Similarly the Xbox UWP (WebView2) exporter embeds WebView2 to load web content, but in the context of a Universal Windows Platform (UWP) app. The macOS WKWebView uses the system web view (based on WebKit, the Safari browser engine) to load web content in a macOS app. The Linux CEF exporter uses the Chromium Embedded Framework (CEF) instead of a system webview, as Linux does not provide a consistent webview in the platform.

These applications can be thought of as a "wrapper" around the web content. Plugins can provide a dynamic link library that extends the wrapper with custom features using the full capabilities of the wrapper application - hence the name *wrapper extension*. The model is similar to Cordova on mobile, where a Cordova plugin can be used for platform-specific integration and called from JavaScript, performing a similar role to a wrapper extension.

Dynamic link libraries use the extension **.dll** on Windows, **.dylib** on macOS and **.so** on Linux. For brevity this guide will use the term DLL to refer to any of these.

The wrapper extension system uses a minimal message-passing system to send small amounts of JSON data between the Construct plugin and the wrapper extension. This allows them to communicate so the wrapper extension can perform tasks for the Construct plugin that are not normally achievable in JavaScript alone. It is specifically designed for integrating C/C++ SDKs such as Steamworks.

The Construct Addon SDK includes the wrapper extension SDK under the path *plugin-sdk/wrapperExtensionPlugin*. A wrapper extension works as follows:

- A Visual Studio 2022 (the Community edition is a free download) solution for Windows, an Xcode project for macOS, and a CMake project for Linux, in the *extension* subfolder that uses C++ code to build a DLL which integrates custom features, such as a C/C++ SDK like Steamworks.

- The DLL uses *.ext.dll* (Windows), *.ext.dylib* (macOS) or *.ext.so* (Linux) as the file extension. The wrapper application looks for files with this name in the same folder as the executable, and will automatically load them on startup.
- The Construct plugin bundles the DLL file by calling `AddFileDependency()` with the type `"wrapper-extension"`. This means when a project using the addon is exported, it will also export the necessary file (e.g. the *.ext.dll* file for Windows).
- The Construct plugin can then detect that the wrapper extension is available, and if it is, send messages instructing the wrapper extension to perform certain tasks.

The sample in the SDK implements a wrapper extension that demonstrates returning data from C++ back to JavaScript, and implements an action that shows a message box to the user. This uses the Windows MessageBox API, the macOS NSAlert API, and GTK on Linux.

Note that in a UWP app (for the Xbox UWP exporter), the DLL must be configured as a Universal Windows DLL. See the Xbox Live UWP plugin code on GitHub for an example of a wrapper extension configured this way.

Messaging

In order to exchange messages, both the wrapper extension and the Construct plugin must set the same *component ID*. This must uniquely identify your plugin/extension combination. If any other plugin/extension uses the same component ID, it will cause a conflict and one of the plugins will fail.

The wrapper extension should call `iApplication->RegisterComponentId()` in the `WrapperExtension` constructor to register its component ID. The JavaScript plugin should call `this.SetWrapperExtensionComponentId()` in its constructor to register the same component ID. Then the two can exchange messages.

The JavaScript plugin should call `this._isWrapperExtensionAvailable()` after setting the component ID to check that the wrapper extension is available. This is because it is unavailable in other exporters, and also because it's possible that loading the wrapper extension could fail for some reason. If the wrapper extension is unavailable, async messages will return a promise that never resolves, which could cause the project to hang. It's advisable to also provide an Is available condition so users can check the plugin features are available in their event sheets.

There are two kinds of ways messages can be sent from JavaScript to the wrapper extension: a one-off message, and an async message.

One-off messages

A one-off message is a "fire and forget" scheme: a message will be sent but no attempt is made to receive a result or identify if the operation completed.

A one-off message can be sent from JavaScript with a call `this._sendWrapperExtensionMessage("message-id", [params...])`. The message ID identifies the kind of message. The second parameter is an optional array of parameters to pass with the message. These must only be boolean, number or string type values. Messages sent from the wrapper extension can be received with `this._addWrapperExtensionMessageHandler("message-id", handlerFunc)`. The handler function is passed an object with a small amount of JSON data sent from the wrapper extension.

A one-off message can be sent from the wrapper extension with a call like:

```
// C++
SendWebMessage("message-id", {
        { "sampleString1",     "Hello world!" },
        { "sampleString2",     "Foo bar baz" },
});
```

In this case the second parameter is a small amount of JSON data that is passed to the JavaScript message handler. The keys must be strings, and the values may only be boolean, number (`double` type to match JavaScript's number type), or string. (Strings in the C++ SDK must be `std::string` or C-style `char*` in UTF-8 encoding.)

Note: *when reading JavaScript object properties sent from C++, be sure to use the minify-proof string syntax (e.g.* `result["sampleString1"]`*), as these properties come from an external source and so should not be changed by the minifier. See Script minification for more details.*

Wrapper extensions receive all messages from JavaScript to the same `HandleWebMessage()` method. That method receives a string of the message ID, and it's up to the wrapper extension to examine that string and respond appropriately depending on the kind of message. The recommended architecture is to use that method solely to distinguish the kind of message, unpack parameters, and then call a dedicated handler method.

Async messages

JavaScript can also send an asynchronous message to the wrapper extension. (This is only supported for JavaScript - there is not currently any support for the wrapper extension to send an asynchronous message to JavaScript.) This is done by calling `this._sendWrapperExtensionMessageAsync()` which works similarly to `this._sendWrapperExtensionMessage()`, except it returns a promise that resolves when the wrapper extension responds to the message. It is a useful way to retrieve data from the wrapper extension, including whether a requested operation completed successfully. It can also be used on startup to perform initialization work.

The wrapper extension receives asynchronous messages the same way as one-off messages, except the `asyncId` parameter is set to a unique number for the message. In order to respond to the message, it must call `SendAsyncResponse()` passing the same `asyncId` the message was received with, e.g.:

```
SendAsyncResponse({
        { "sampleString1",                      "Hello world!" },
        { "sampleString2",                      "Foo bar baz" },
}, asyncId);
```

The provided JSON data works the same as with `SendWebMessage()`, and is used as the value that the JavaScript call to `_sendWrapperExtensionMessageAsync()` resolves with.

The wrapper extension must respond to asynchronous messages on all codepaths, including in the event of an error. If it does not, the promise returned by `_sendWrapperExtensionMessageAsync()` *will never resolve, which could result in the project hanging.*

Exporting properties to package.json

A Construct plugin that uses a wrapper extension can make use of the IPluginInfo method `SetWrapperExportProperties()` to export the values of some plugin properties to the exported package.json file. The wrapper extension can then parse the contents of package.json on startup and find the values of these properties before any web content has loaded at all. This can be useful for loading SDKs with plugin properties specifying initialization details (like an app ID or API key) before anything else loads, which some SDKs recommend.

Suggested architecture

It is recommended that as much of your plugin logic as possible is implemented in JavaScript. Only send messages to the wrapper extension to make specific API calls that aren't possible from JavaScript. This way it minimizes the amount of platform-specific C++ code necessary, and ensures as much logic as possible happens in the same place, rather than spread across different codebases. Also, JavaScript is an easier programming language to work with, as it has easier-to-use facilities for async code and avoids the need for manual memory management (while still providing excellent performance).

Strings on Windows

For historical reasons, Windows APIs called from C++ that use strings generally use "wide strings" with UTF-16 encoding. These are strings of "wide characters" which are 16-bit types on Windows. This uses the `wchar_t` type for a character, and `std::wstring` for the STL string equivalent (as well as types like `LPCWSTR` for the C-style equivalent in Windows header files).

On the other hand, most modern software and recent C++ codebases use UTF-8 encoding. This uses the standard 8-bit `char` type and `std::string` in the STL (as well as types like `char*` for the C-style equivalent).

> *Remember that in both UTF-8 and UTF-16, a single "character" is in fact a Unicode code unit and doesn't necessarily correspond to a single visible character.*

Consistent with the modern style, the wrapper extension SDK uses UTF-8 encoding when dealing with strings. However this means strings must be converted when calling Windows APIs that use wide strings. The SDK provides the utility methods `Utf8ToWide()` which converts a UTF-8 `std::string` to a UTF-16 `std::wstring` suitable for passing to Windows APIs. The `c_str()` method of STL strings also provides a C-style string that Windows usually expects. The `WideToUtf8()` method can then also convert a UTF-16 `std::wstring` back to a UTF-8 `std::string`, suitable for converting back wide strings returned by Windows APIs. The recommended approach is to use UTF-8 everywhere, and only convert to UTF-16 to call a Windows API that requires it; if the API call returns a UTF-16 string then it should immediately be converted back to UTF-8. This means as much code as possible only uses UTF-8 and UTF-16 is used minimally solely to interact with Windows APIs.

> *The Windows WebView2 wrapper application also configures the process code page to UTF-8. This makes it possible to directly call '-A' variant Windows APIs (e.g. `MessageBoxA()`) with UTF-8 strings. However it is only supported in Windows 10 version 1903 (May 2019 Update) and newer, and may not be supported with all available Windows APIs. For more information see the Microsoft documentation Use UTF-8 code pages in Windows apps. While this option may be useful, particularly in future, for the most straightforward and consistent approach we recommend continuing to call all Windows APIs with wide strings in UTF-16 format.*

macOS architecture

Apple platforms typically use Objective-C or Swift. It is easiest to interoperate with C++ code with Objective-C and this is the approach we recommend. In Xcode you can configure .cpp files to be the type *Objective-C++ code*, and this allows mixing both Objective-C and C++ code. This means a .cpp file can also use `#import` and Objective-C style calls like `[alert setMessageText:nsTitle]`. The sample wrapper extension code uses this approach.

Apple platforms typically use NSString for strings. Similar to our advice for Windows, we recommend using `std::string` with UTF-8 encoding as much as possible, and only use `NSString` when interacting with platform APIs, converting to `NSString` to make Objective-C calls and immediately converting any `NSString` results back to `std::string`. Some sample code for converting between `NSString` and `std::string` is shown below.

```
std::string message = "Hello world";
```

```
// Convert std::string to NSString
NSString* nsMessage = [NSString stringWithUTF8String:message.c_str()];

// Convert NSString back to std::string
message = std::string([nsMessage UTF8String]);
```

By default Xcode builds universal binaries which include code for both Intel (x64) and Apple Silicon (ARM64) architectures.

Platform-specific code

You can use the following preprocessor definitions to identify the platform being built and so incorporate platform-specific code. Note these definitions only identify the OS, not the architecture.

- `_WIN32` is defined by Visual Studio for Windows builds
- `__APPLE__` is defined by Xcode for macOS
- `__linux__` is defined by gcc for Linux builds

Additional examples

There are real-world examples of using the wrapper extension SDK to integrate SDKs on the Scirra GitHub account, including open-source plugins that integrate the Steamworks SDK and the Epic Games Online Services (EOS) SDK. These should help provide sample code that demonstrates how to build a useful wrapper extension.