IPLUGININFO INTERFACE

View online: https://www.construct.net/en/make-games/manuals/addon-sdk/reference/iplugininfo

IPluginInfo defines the configuration for a plugin. It is typically accessed in the plugin constructor via `this._info`.

Methods

SetName(name)

Set the name of the addon. Typically this is read from the language file.

SetDescription(description)

Set the description of the addon. Typically this is read from the language file.

SetVersion(version)

Set the version string of the addon, in A.B.C.D form. Typically this is set to the `PLUGIN_VERSION` constant.

SetCategory(category)

Set the category of the addon. Typically this is set to the `PLUGIN_CATEGORY` constant. It must be one of `"data-and-storage"`, `"form-controls"`, `"general"`, `"input"`, `"media"`, `"monetisation"`, `"platform-specific"`, `"web"`, `"other"`.

SetAuthor(author)

Set a string identifying the author of the addon.

SetHelpUrl(url)

Set a string specifying a URL where the user can view help and documentation resources for the addon. The website should be hosted with HTTPS.

SetPluginType(type)

Set the plugin type. This can be `"object"` or `"world"`. The world typeh represents a plugin that appears in the Layout View, whereas the object type represents a hidden plugin, similar to the Audio plugin (a single-global type) or Dictionary. World type plugins must derive from `SDK.IWorldInstanceBase` instead of `SDK.IInstanceBase` and implement a `Draw()` method.

SetIcon(url, type)

Set the addon icon URL and type. By default the URL is `"icon.svg"` and the type is `"image/svg+xml"`. It is recommended to leave this at the default and use an SVG icon, since it will scale well to any display size or density. However you can change your addon to load a PNG icon with `SetIcon("icon.png", "image/png")`.

SetIsResizable(isResizable)

For `"world"` type plugins only. Pass `true` to enable resizing instances in the Layout View.

SetIsRotatable(isRotatable)

For `"world"` type plugins only. Pass `true` to enable the *Angle* property and rotating instances in the Layout View.

SetIs3D(is3d)

For `"world"` type plugins only. Pass `true` to specify that this plugin renders in 3D. This will cause the presence of the plugin in a project to enable 3D rendering when the project *Rendering mode* property is set to *Auto* (which is the default setting).

SetHasImage(hasImage)

For `"world"` type plugins only. Pass `true` to add a single editable image, such as used by the Tiled Background plugin.

SetDefaultImageURL(url)

For plugins that use a single editable image only. Set the URL to an image file in your addon to use as the default image when the object is added to a project, e.g. `"default.png"`.

> *When using developer mode addons, remember to add the image file to the file list in addon.json.*

SetIsTiled(isTiled)

For `"world"` type plugins only. Pass `true` to indicate that the image is intended to be tiled. This adjusts the texture wrapping mode when Construct creates a texture for its image.

SetIsDeprecated(isDeprecated)

Set a boolean of whether the addon is deprecated or not. If you wish to replace your addon with another one, the old one can be deprecated with `SetIsDeprecated(true)`. This makes it invisible in the editor so it cannot be used in new projects; however old projects with the

addon already added can continue to load and work as they did before. This discourages use of the deprecated addon without breaking existing projects that use it.

SetIsSingleGlobal(isSingleGlobal)

Pass `true` to set the plugin to be a *single-global* type. The plugin type must be `"object"`. Single-global plugins can only be added once to a project, and they then have a single permanent global instance available throughout the project. This is the mode that plugins like Touch and Audio use.

SetSupportsZElevation(supportsZElevation)

Pass `true` to allow using Z elevation with this plugin. The plugin type must be `"world"`. By default the renderer applies the Z elevation before calling the `Draw()` method on an instance, which in many cases is sufficient to handle rendering Z elevation correctly, but be sure to take in to account Z elevation in the drawing method if it does more complex rendering.

> `AddCommonZOrderACEs()` *will add common ACEs for Z elevation if supported.*

SetSupportsColor(supportsColor)

Pass `true` to allow using the built-in color property to tint the object appearance. The plugin type must be `"world"`. By default the renderer sets the color before calling the `Draw()` method on an instance, which in many cases is sufficient to handle rendering with the applied color, but be sure to take in to account the instance color in the drawing method if it does more complex rendering.

> `AddCommonAppearanceACEs()` *will add common ACEs for color if supported.*

SetSupportsEffects(supportsEffects)

Pass `true` to allow using effects, including the *Blend mode* property, with this plugin. The plugin type must be `"world"`. If the plugin does not simply draw a texture the size of the object (as Sprite does), you should also call `SetMustPreDraw(true)`.

SetMustPreDraw(mustPreDraw)

Pass `true` to disable an optimisation in the effects engine for objects that simply draw a texture the size of the object (e.g. Sprite). This is necessary for effects to render correctly if the plugin draws anything other than the equivalent the Sprite plugin would.

SetCanBeBundled(canBeBundled)

Pass `false` to prevent the addon from being bundled via the *Bundle addons* project property. By default all addons may be bundled with a project, and it is recommended to leave this enabled for best user convenience. However if you publish a commercial addon and want to prevent it being distributed by project-bundling, you may wish to disable this.

AddCommonPositionACEs()

AddCommonSceneGraphACEs()

AddCommonSizeACEs()

AddCommonAngleACEs()

AddCommonAppearanceACEs()

AddCommonZOrderACEs()

Add common built-in sets of actions, conditions and expressions (ACEs) to the plugin relating to various built-in features.

> ***Note:*** *if adding common scene graph ACEs, your plugin must be prepared to handle being added in to a scene-graph hierarchy, and having its position, size and angle controlled automatically. It must also support all the properties modifiable by hierarchies, otherwise the scene graph feature may not work as expected.*

SetProperties(propertiesArray)

Set the available addon properties by passing an array of PluginProperty. See Configuring Plugins for more information.

AddCordovaPluginReference(opts)

Add a dependency on a Cordova plugin, that will be included when using the Cordova exporter. For more information see Specifying dependencies.

AddCordovaResourceFile(opts)

Add a resource file to be included with Cordova exports. For more information see Specifying dependencies.

AddFileDependency(opts)

Add a dependency on another file included in the addon. For more information see Specifying dependencies.

AddRemoteScriptDependency(url, type)

Add a script dependency to a remote URL (on a different origin). By default it loads the URL as a "classic" script; the `type` parameter is optional and can be set to the string `"module"`

to load the dependency as a module script instead. For more information see Specifying dependencies.

SetGooglePlayServicesEnabled(enabled)

Pass `true` to enable Google Play Services in Cordova Android exports. `<preference name="GradlePluginGoogleServicesEnabled" value="true" />` will be added in config.xml.

> *Since this can only be configured once, if any plugin in the project specifies to enable Google Play Services, it will be enabled for the entire project.*

SetC3RuntimeScripts(arr)

Pass an array of strings to set the list of runtime scripts the addon uses. The default list is the following, and note that this method entirely replaces it: *"c3runtime/plugin.js", "c3runtime/type.js", "c3runtime/instance.js", "c3runtime/conditions.js", "c3runtime/actions.js", "c3runtime/expressions.js"*.

AddC3RuntimeScript(path)

Add a single runtime script path to the existing list of runtime scripts the addon uses, e.g. "c3runtime/additionalScript.js".

SetRuntimeModuleMainScript(path)

Set the main script that the runtime loads as a module. When this method is called, Construct will only load that script, and it is expected that all your other scripts are imported in the main script. If this method is not called, Construct automatically generates a main script that imports every single runtime script - but note that makes it difficult to use modules properly. See runtime scripts for more information.

SetDOMSideScripts(arr)

Specify an array of script paths to load in the main document context rather than the runtime context. For more information see the section *DOM calls in the C3 runtime* in Runtime scripts.

SetScriptInterfaceNames(opts)

Use this method to tell Construct the names of your script interface classes. This is necessary to generate the correct TypeScript definition files. `opts` is an object which allows specifying the names for the `instance`, `objectType` and `plugin` interface names as necessary, e.g.:

```
this._info.SetScriptInterfaceNames({
        instance: "ISpriteInstance"
```

```
});
```

SetTypeScriptDefinitionFiles(arr)

Specify an array of TypeScript definition files (.d.ts) your addon provides. This should be used to provide full TypeScript definitions of any script interfaces your addon provides, which is necessary for projects using TypeScript with your addon. Example:

```
this._info.SetTypeScriptDefinitionFiles(["c3runtime/ISpriteInstance.d.ts"]);
```

SetWrapperExportProperties(componentId, propertyIds)

For use with single-global plugins using wrapper extensions. Specify an array of property IDs which will have their values exported to package.json under the key `"exported-properties"`. For example the following call:

```
this._info.SetWrapperExportProperties("my-component-id", ["first-property", "second-property
```

will result in the following content being included in the exported package.json:

```
{
	"exported-properties": {
		"my-component-id": {
			"first-property": "...",
			"second-property": "..."
		}
	}
}
```

This can then be read by the wrapper extension using the IApplication method `GetPackageJsonContent()` and parsing it with a library like json.hpp. This allows the wrapper extension to initialize early, before any web content is loaded, while still making use of settings specified in plugin properties.