Design: Paragon Plugin Model

On This Page

- Context
- Requirements
  - Development
  - Hosting
  - Runtime
- Design
  - Considerations
  - Outstanding Items
  - High Level Architecture
  - Assemblies & Dependencies
  - Plugin Metadata
  - Plugin Abstraction
    - Communication
    - Interop
  - Bridging Plugins To The V8 Engine
  - Bridging The Browser And Render Processes
  - Bridging Registered Native Code Within An Extension
  - Extension Repository

Context

Paragon bridges its core native functions (windowing, state storage, application registration lookup etc) into JavaScript by registering callbacks with the V8 contexts created by CEF.

This bridging should be generalised so that all native functions are exposed via an plugin model. This will allow modularisation of the kernel functions and also the means for JavaScript to call into native code for scenarios where it is not possible to call remote services from JavaScript directly. It should be possible for application teams to create plugins when required.

Requirements

Development

- Plugins must be written in .NET
- Plugin developers should not be required to understand CEF or the Paragon container itself - only the plugin/host API
- Plugin assemblies must be signed with a standard (Paragon) key which the container will verify

Hosting

- Plugin binaries should be stored in a central repository that the container connects to
- Must be possible to have QA and PROD versions of plugins to load into the corresponding container instance
- There must be a plugins whitelist/blacklist that the container checks when asked to load a plugin
- Kernel plugins can be in the browser (container) or render processes
- Application plugins can only be in the render processes
- Embedded Paragon control will expose the appropriate kernel plugins but will not allow the embedding application to register its own plugins - there will be a generic messaging plugin the embedding application can hook to if needed

Runtime

- All plugin methods will be accessed from JavaScript using calls that supply an asynchronous callback
- JavaSript should be able to access properties on plugin objects
- Plugins must be able to expose events to JavaScript (via the usual addListener/removeListener convention in JS)
- Web applications can request that application (non-kernel) plugins be loaded
- Must be possible to provide a separate instance of a given plugin to each Paragon Application
  - e.g. the local storage API shouldn't allow a window in one application to access state from another loaded application
  - e.g. the app.window API needs to be scoped so that applications can only manipulate their own windows
- **NEW**: plugins should be allowed to specify a JS file that they replace - if that JS file is loaded by a web control it is silently denied, with the plugins taking its place.

- Allows replacement of JS in apps such as Symphony

Design

Considerations

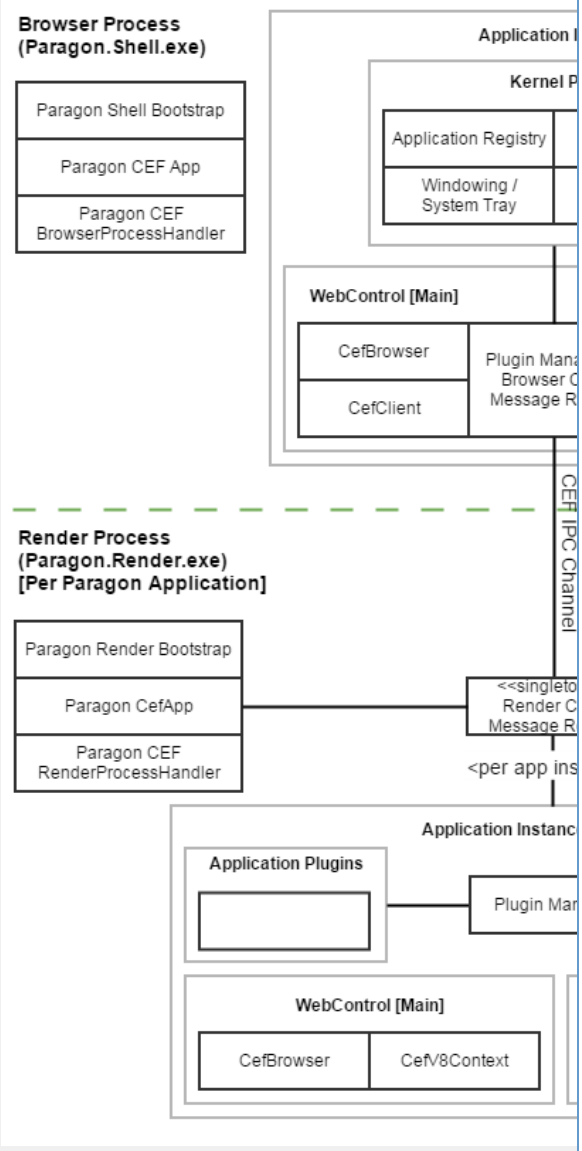
- The container/embedded code will need to define the the concept of an application (whether hosted or packaged) that can have multiple related web controls that share a common set of plugins
- The container/embedded code will need the ability to construct a set of browser side plugin instances for a given application (to implement cases where plugin usage needs to be segregated per application)
- To allow different plugin instances across applications hosted by the same container, the browser side plugins need to be provided to the render process per CefBrowser, in the CefLifeSpanhandler.OnAfterCreated hook. This allows the plugin proxies to be created in the render process and associated per CefBrowser, **before** the browser performs any navigation (which subsequently creates V8 contexts)
    - The web control will expose an event or method for the container to supply it with the appropriate plugin objects to expose to JavaScript
- Opening child windows from the application's main (initial) page will be via the standard window.open() - this ensures all pages/windows for a given application stay in the same CEF render process as the Chromium process-per-tab model will be used
    - This means no equivalent of the chrome.app.window.create() call will be implemented
    - The container's application lifecycle will ensure that any windows opened via window.open() are closed when the application is stopped
- Support for setting and getting plugin properties from JavaScript could be done by sending a snapshot of the native object's exposed property values initially, then listening to INotifyPropertyChanged if implemented - updates being sent via CefProcessMessage to the render process to update the local snapshot - this local snapshot is what the V8 property integration accesses
- The Chrome App APIs are being used as a starting point but there will not be support for the actual Chrome App APIs themselves. However the container may support the CRX package format for loading Paragon Apps
- Some APIs that will be available in JavaScript will need to be implemented completely in the container scenario, but be hooks in the embedded control scenario. Window functions are a good example, where in the embedded control it is likely that the host application will want to create the host windows for new web pages/dialogs etc.
- For the container scenario, the application launcher and other core UI components should be HTML5 applications.
- Plugins that are returned by existing plugins (i.e. dynamic objects) will be held in the owner process in a plugin cache so that subsequent calls can be made to them. However, the plugin code will need to signal to the plugin hosting layer when that dynamic plugin should be discarded, via an attributed event on the plugin type.
    - If a V8 context tries to make calls to a discarded plugin, it should be returned a standard error that the plugin isn't available any more.
    - The plugin cache should also act as the point the plugin manager locates plugins to perform operations on.

Outstanding Items

- Plugin package format, e.g. a plugin code base could be several assemblies - what sort of archive should be supported
- Plugin whitelist/blacklist service

High Level Architecture

Container



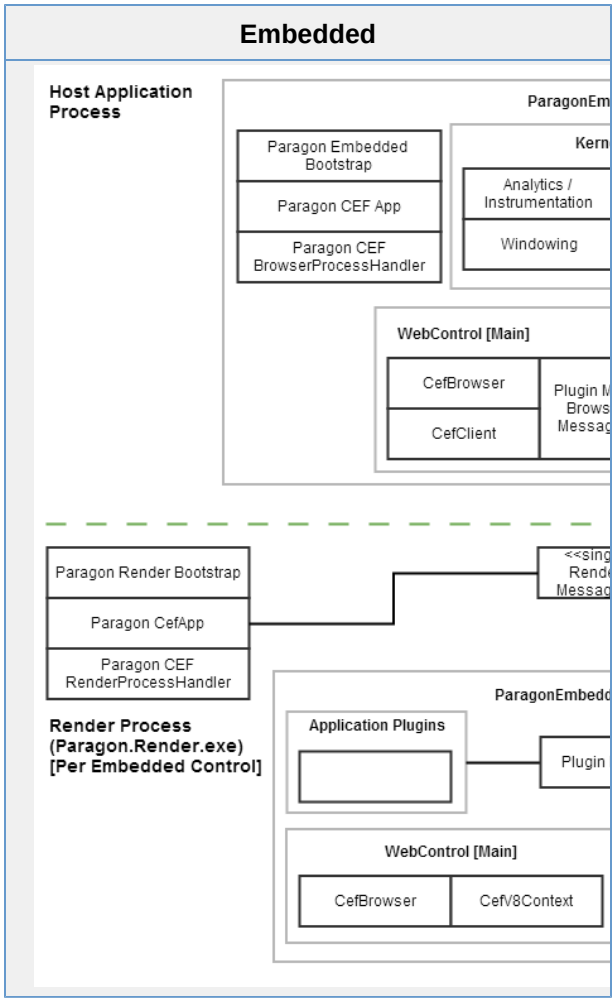
When the container shell process starts up its configuration or command line parameters should include its environment (Test, Prod) and connection to the main plugin repository.

The startup includes the following plugin related steps:

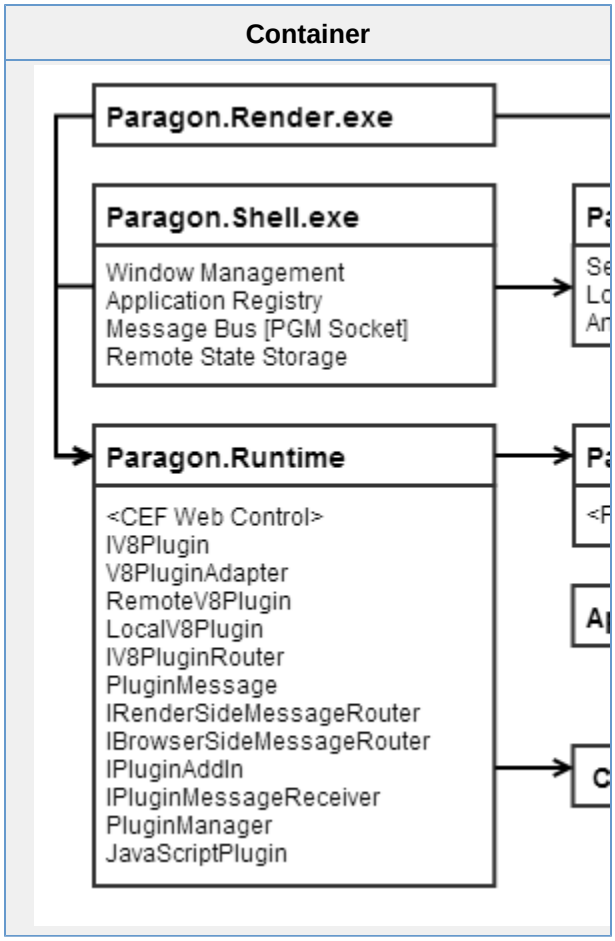
- The shell establishes a connection to the repository and retrieves details of the kernel plugins for the user.
- The kernel code is initialized, and the plugins it wishes to expose added to the overall set managed by the browser process

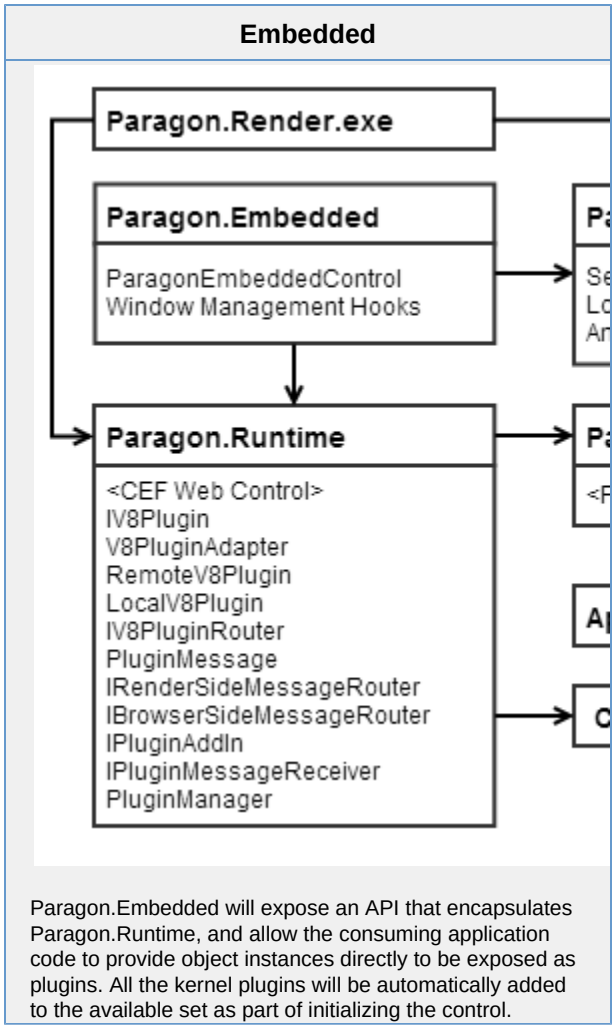
These steps should happen before the initial HTML application is started up.

When a Paragon application is started up, its manifest will be examined to see if it references any plugins that aren't kernel or shared. If it does, those plugins are downloaded and the paths to them on disk are provided to the render process to load up.



Assemblies & Dependencies

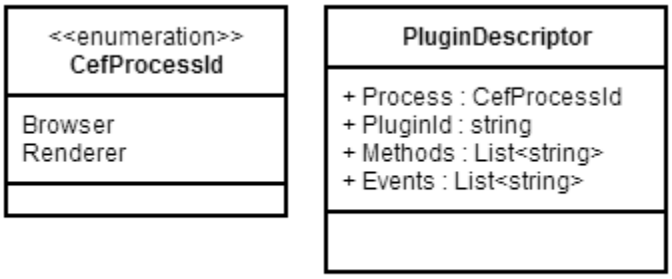

Container
Paragon.Render.exe
Paragon.Shell.exe
Window Management
Application Registry
Message Bus [PGM Socket]
Remote State Storage
Paragon.Runtime
<CEF Web Control>
IV8Plugin
V8PluginAdapter
RemoteV8Plugin
LocalV8Plugin
IV8PluginRouter
PluginMessage
IRenderSideMessageRouter
IBrowserSideMessageRouter
IPluginAddIn
IPluginMessageReceiver
PluginManager
JavaScriptPlugin



Plugin Metadata

The Descriptor types define all the metadata required to remotely interact with a plugin:

- PluginProcessId specifies which of the Paragon processes the plugin resides in - this is used to route calls, e.g. when code in an application plugin wishes to communicate with a browser located kernel plugin, the call will be routed by the render side message router based on this field.
- PluginId will be a unique JavaScript object path for static plugins, but a Guid for dynamic plugins (plugins returned from other plugin methods).
- The set of events that can be attached to.
- The methods that can be invoked. Note that method overloads aren't supported.
- The IsSynchronous flag on MethodDescriptor allows the runtime to know when to wait on the render thread for a function response message to arrive to then translate into a V8 return value, as opposed to setting up an asynchronous callback into V8.



Plugin Abstraction

The plugin abstraction covers several areas:

Communication

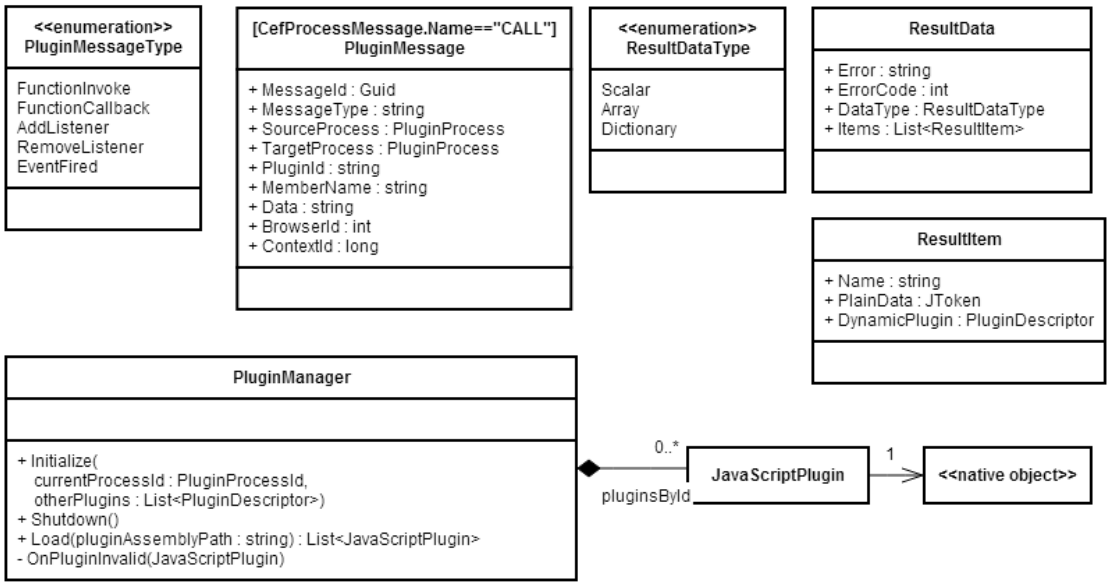
- **PluginMessage** provides a common format for the various interactions as defined by **PluginMessageType**, facilitating interaction of V8 and plugins and also interaction between plugins
    - **MessageId** provides the mechanism for correlating function callbacks and event subscriptions
    - **MessageType** specifies which interaction the message covers:
        - FunctionInvoke and corresponding FunctionCallback (which should only be needed for asynchronous functions, and synchronous methods that have a return type)
        - AddListener and RemoveListener for managing event subscriptions
        - EventFired - raised per listener by the wrapper for the underlying native plugin object
    - **SourceProcessId** specifies the process that sent the message
    - **TargetProcessId** specifies which process the target plugin is in
    - **MemberName** specifies the name of the function or event
    - **Data** contains the JSON serialized payload for the message. For function invokes this should be a JArray
    - **ContextId** - present only for calls originating from a V8 context. Intended as a round trip value used to retrieve the originating context to execute a V8 callback in
- **ResultItem** defines the ability to return either a plain data construct (struct or scalar value) or a dynamic plugin. The Name property is only relevant to dictionaries
- **ResultData** defines the overall result - either a single value, an array or a dictionary. The ErrorCode and Error properties only apply to function callback.
- **IPluginMessageReceiver** defines the ability to receive and process a message. It is implemented by the render and browser side routers, as well as the MAF plugin addin

The following table shows how PluginMessage is populated for the various message types:

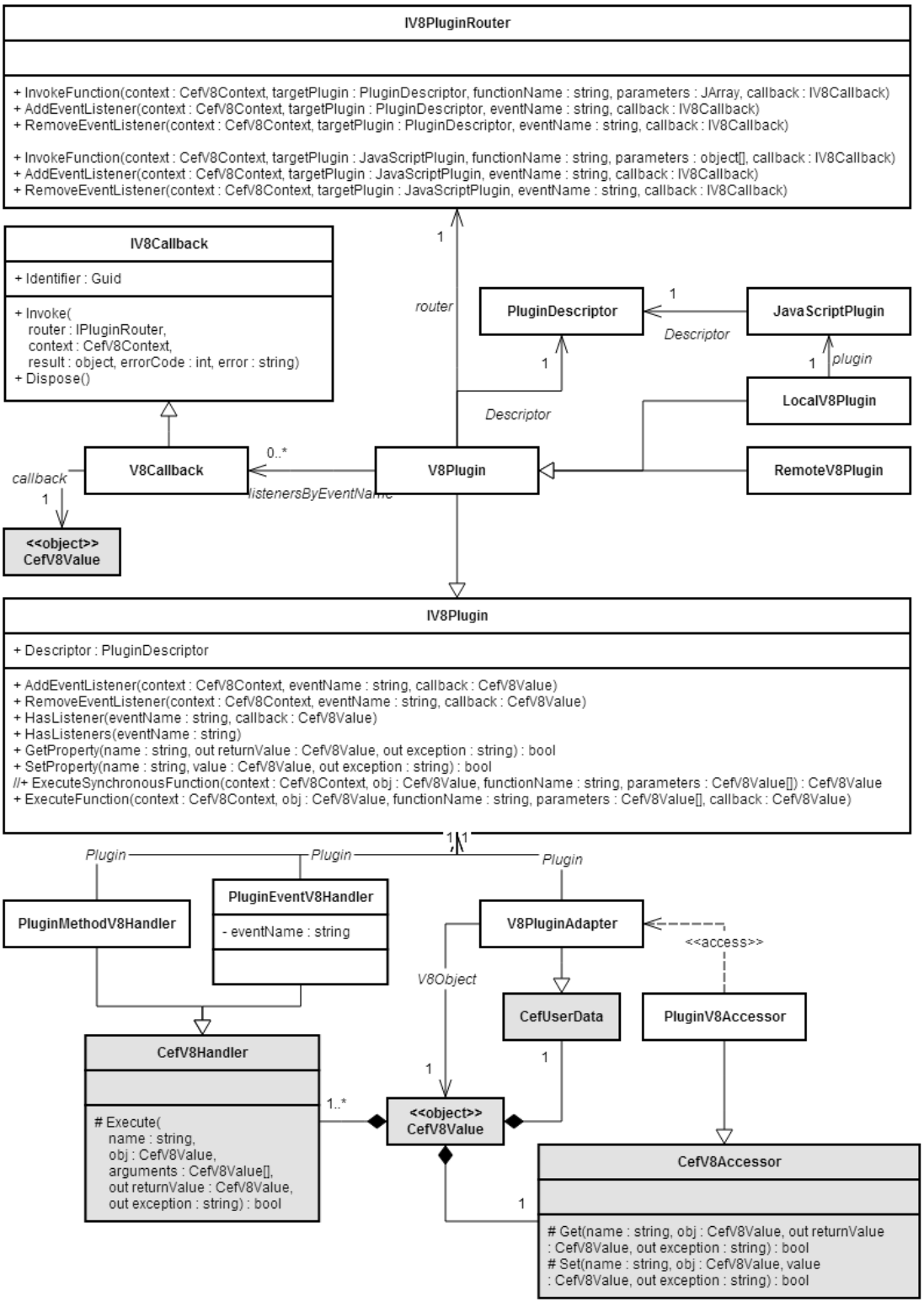
PluginMessage Field	Invoke a function	Send function response	Add event handler	Remove event handler	Publish event
MessageId	New Guid	Copied from FunctionInvoke message	New guid	Copied from AddListener message	Copied from AddListener message
MessageType	FunctionInvoke	FunctionCallback	AddListener	RemoveListener	EventFired
SourceProcessId	Source process	TargetProcessId from FunctionInvoke message	Source process	Source process	Source process
TargetProcessId	Id of process of the target plugin	SourceProcessId from FunctionInvoke message	Id of process of the target plugin	Id of process of the target plugin	SourceProcessId from AddListener message
PluginId	The Id of the target plugin	N/A	The Id of the target plugin	The Id of the target plugin	N/A
MemberName	Name of function to invoke	N/A	Name of event to attach to	Name of event to detach from	N/A
Data	JSON serialized array of parameters	JSON serialized ResultData object	N/A	N/A	JSON serialized ResultData object
BrowserId	Present if invocation originates from V8	Copied from FunctionInvoke message	Present if invocation originates from V8	Present if invocation originates from V8	Copied from AddListener message
ContextId	Present if invocation originates from V8	Copied from FunctionInvoke message	Present if invocation originates from V8	Present if invocation originates from V8	Copied from AddListener message

Interop

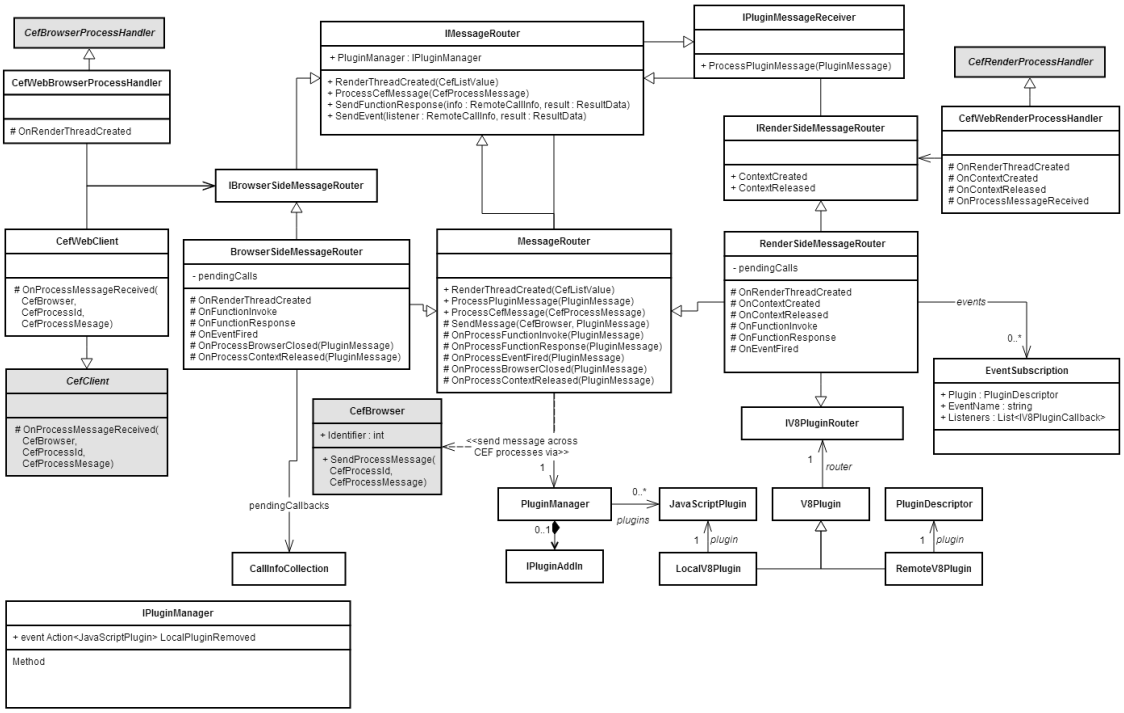
JavaScriptPlugin defines the reflection wrapper that manipulates a given C# plugin object



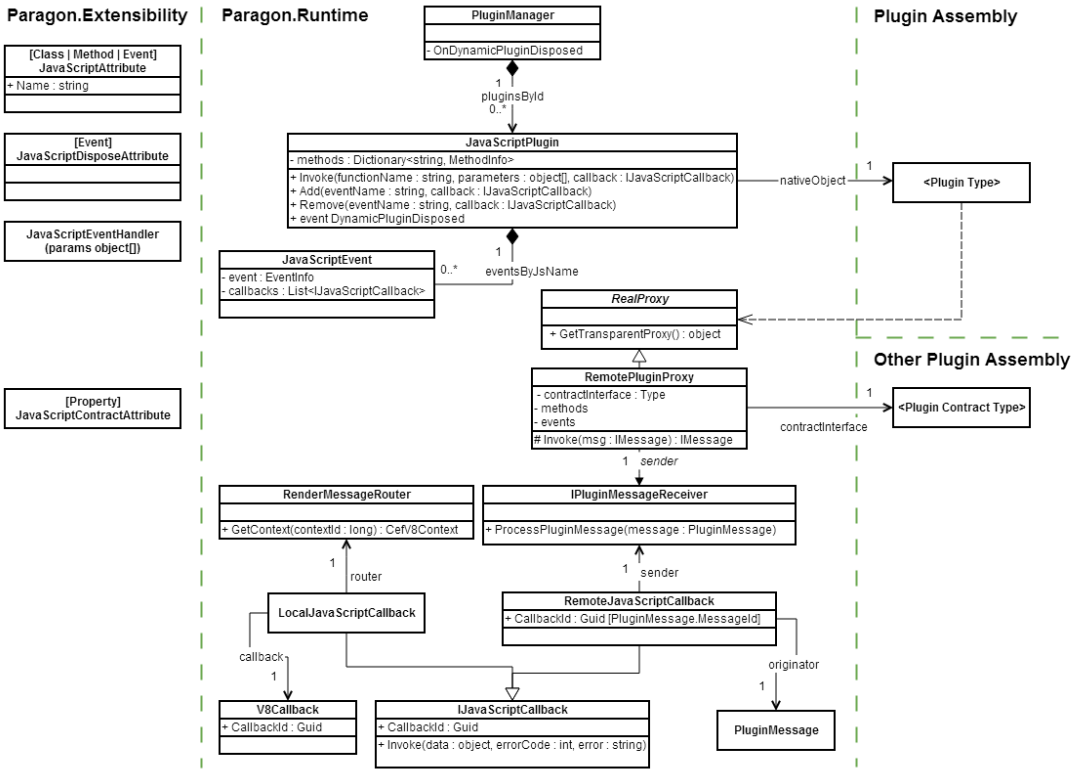
Bridging Plugins To The V8 Engine



Bridging The Browser And Render Processes



Bridging Registered Native Code Within An Extension



Extension Repository

Need to look at whether App Store could support a plugin category for Paragon - needs to handle concept of QA and Prod environments: this should be part of a wider discussion about line of business apps and associated back-end environments.

White/black-listing may well need to be a separate service to the repository itself.