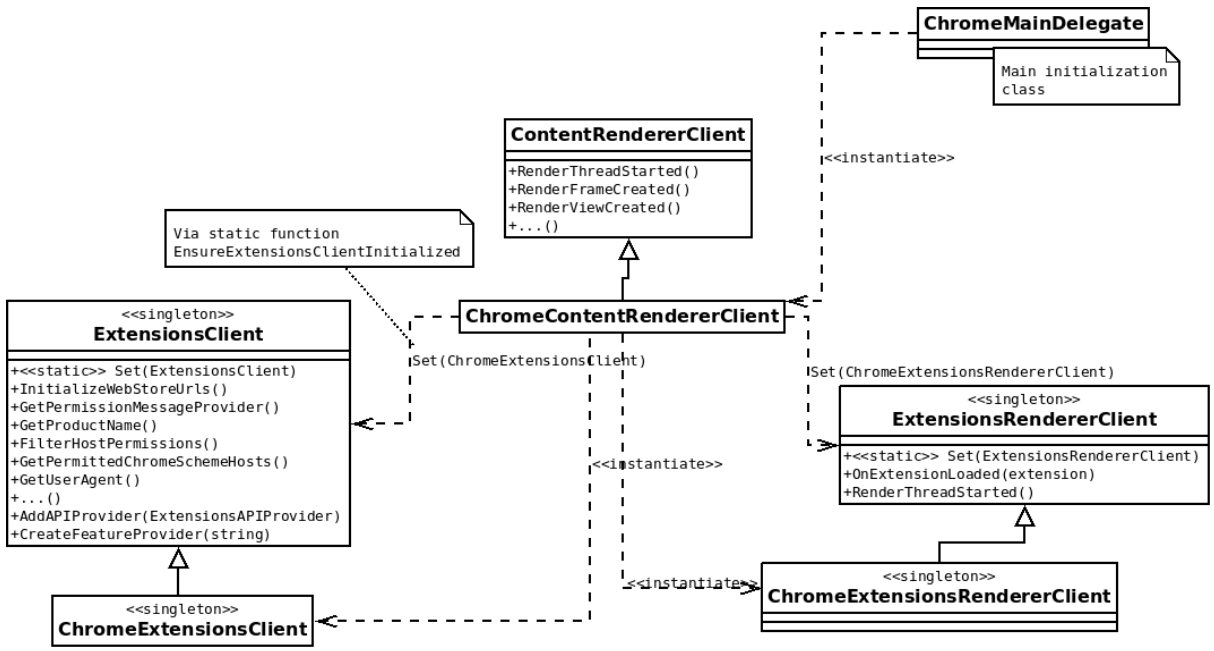
Initialization of the Chromium extension API system

Chromium has infrastructure in place to extend the JavaScript API that is available to web clients that fulfill some conditions. Those web clients can be Chrome extensions, certain websites or local resources like the chrome:// pages. Also, different shell implementations can provide their own extensions. This is an overview of how it works, and what would be required for a custom shell implementation to provide its own API extensions.

Render process

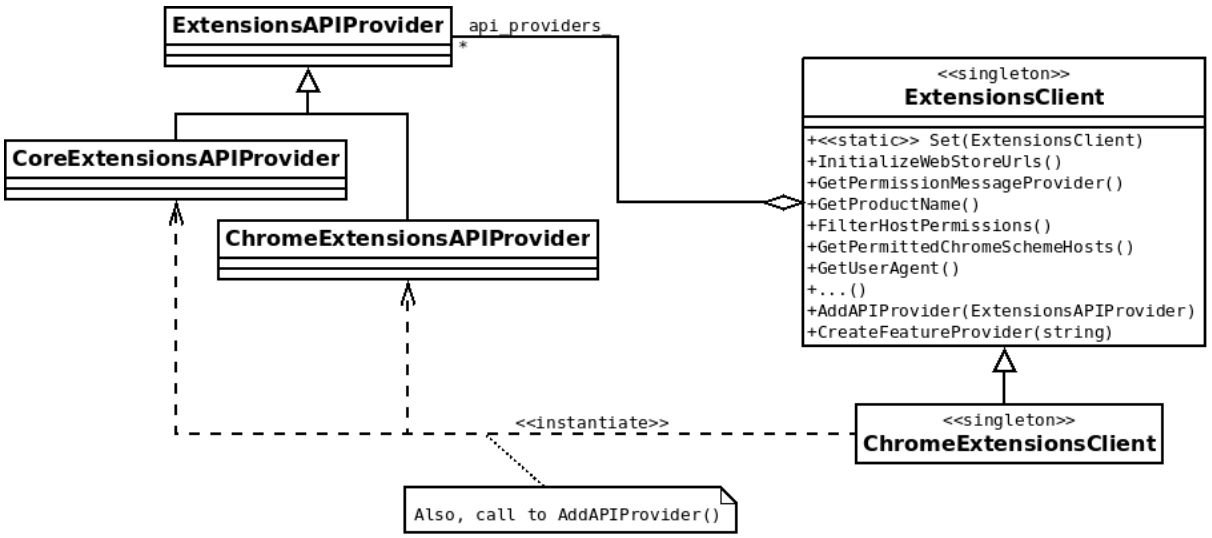
In our previous notes about the Chromium initialization process, we had identified ChromeMainDelegate as the main initialization class. It creates an instance of ChromeContentRendererClient to setup the render process.

Two important classes related with the extension API system are to be initialized by the ChromeContentRendererClient: ExtensionsClient and ExtensionsRendererClient. Both are singletons, the content renderer client sets them up to contain the corresponding Chrome* implementations: ChromeExtensionsClient and ChromeExtensionsRendererClient.



The ExtensionsClient class maintains a list of ExtensionsAPIProvider objects. The constructor of the specific implementation of the client will create instances of the relevant ExtensionsAPIProvider and add them to the list by calling the parent method

AddAPIProvider(). In the case of ChromeExtensionsClient, it will add CoreExtensionsAPIProvider and ChromeExtensionsAPIProvider.



The purpose of the ExtensionAPIProvider objects is to act as a proxy to a set of json configuration files. On one hand, we have _api_features.json, _permission_features.json, _manifest_features.json and _behavior_features.json. These files are documented in _features.md, their purpose is to define the requirements to extension features. The first file is for access to API features: you specify in which contexts or for which kinds of extensions a certain API would be available. The next two files can limit extension usage of permission and manifest entries, and the last one for miscellaneous extension behaviors. The ExtensionAPIProvider class provides four methods to run the code autogenerated from these four files. Notice that not all of them are required.

The providers are later used by ExtensionsClient::CreateFeatureProvider. This method will be called with “api”, “permission”, “manifest” or “behavior” parameters, and it will run the corresponding method for every registered ExtensionsAPIProvider.

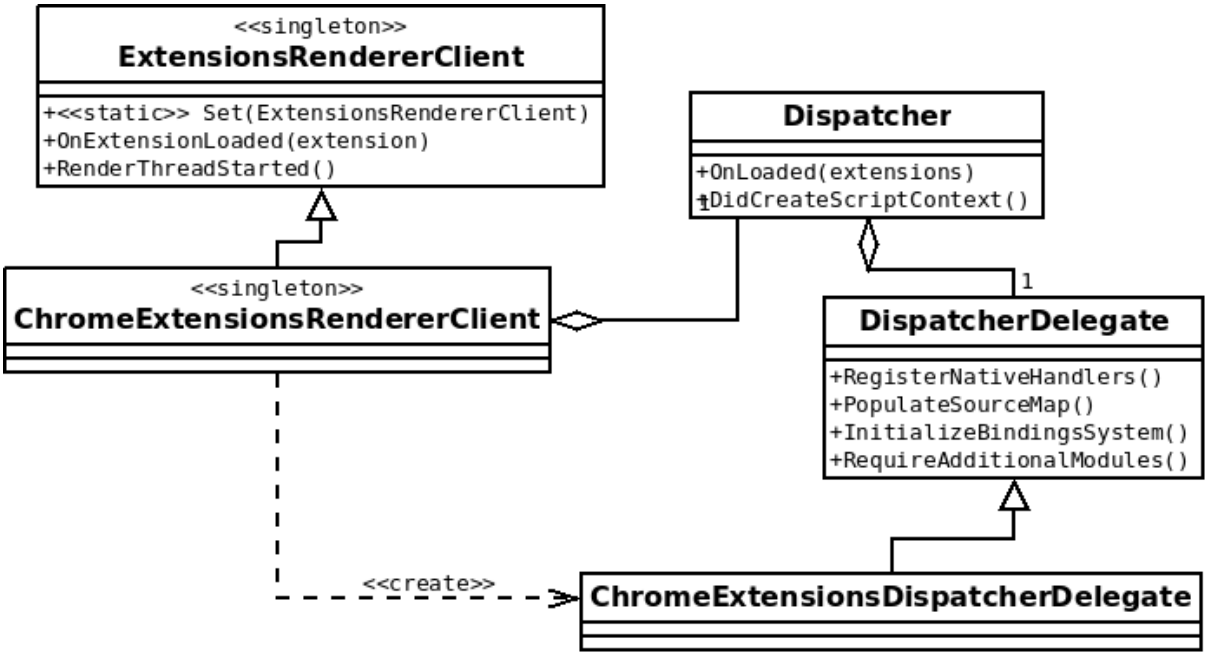
There is also a method called AddAPIJSONSources, where the *_features.json files are expected to be loaded into a JSONFeatureProviderSource. To be able to do that, the files should have been included and packed as resources in one of the shell’s .grd files.

The ExtensionAPIProvider class also proxies the json files that define the extended JS API features. These files are located together with the *_features.json files. The methods GetAPISchema and IsAPISchemaGenerated provide access to them.

If you are creating a custom shell, a correct BUILD.gn file should be placed together with the aforementioned json files, to generate the necessary code based on them. A production-level example is located at ./extensions/common/api/BUILD.gn, or a simpler one at ./extensions/shell/common/api/BUILD.gn for the app_shell.

The ExtensionsRendererClient contains hooks for specific events in the life of the renderer process.
It’s also the place where a specific delegate for the Dispatcher class is created, if needed. In this context, it’s interesting the call to RenderThreadStarted, where an instance of ChromeExtensionsDispatcherDelegate is created. The Chrome implementation of this

delegate has many purposes, but in the context of the extension system, we are interested in the methods PopulateSourceMap, where any custom bindings can be registered, and RequireAdditionalModules, which can require specific modules from the module system.



The DispatcherDelegate has mainly the purpose of fine-tuning the extension system by:

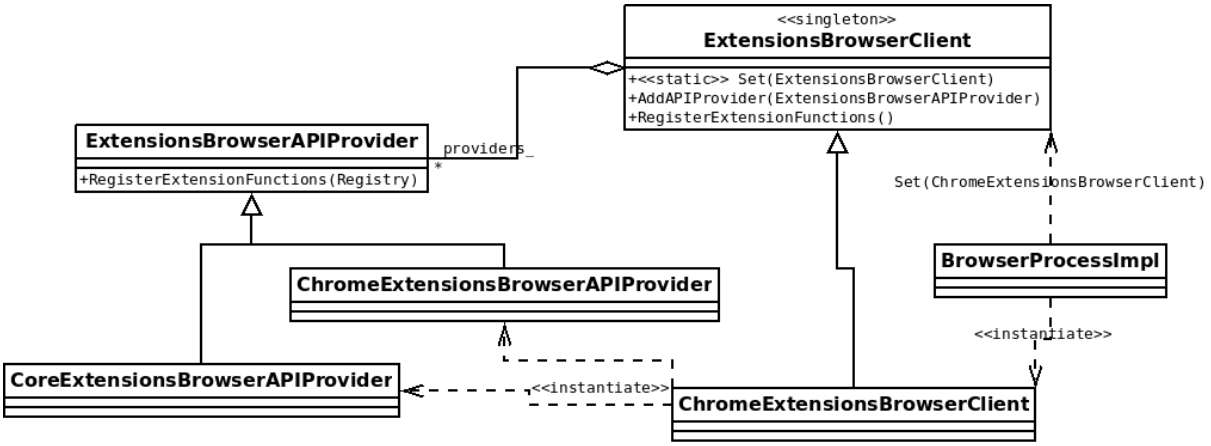
- Registering native handlers in the module system at RegisterNativeHandlers
- Injecting custom JS bindings and other JS sources in PopulateSourceMap
- Binding delegate classes to any extension API that needs them in InitializeBindingsSystem
- Require specific modules from the module system in RequireAdditionalModules. Otherwise, modules would be required by extensions, according to the permissions in their manifest, when they are enabled.

The specific case of the ChromeExtensionsDispatcherDelegate implements the first three methods, but not RequireAdditionalModules. This one could be added in a setup where extension APIs are expected to be used outside the context of an extension and, for that reason, there is not a manifest listing them.

Browser process

In the browser process, there is a related interface called ExtensionsBrowserClient, used for extensions to make queries specific to the browser process. In the case of Chrome, the

implementation is called ChromeExtensionsClient and it has knowledge of profiles, browser contexts or incognito mode.



Similarly to the ExtensionsClient in the render process, the ExtensionsBrowserClient maintains a list of ExtensionsBrowserAPIProvider objects. The ChromeExtensionsClient instantiates two ExtensionsBrowserAPIProvider, the Core ExtensionsBrowserAPIProvider and the Chrome ExtensionsBrowserAPIProvider, and adds them to the list to be used by the function RegisterExtensionFunctions, which calls a method of the same name for every registered ExtensionsBrowserAPIProvider.

These objects are just expected to implement that method, which populates the registry with the functions declared by the API in that provider. This registry links the API with the actual native functions that implement it. From the point of view of the ExtensionsBrowserAPIProvider:: RegisterExtensionFunctions, they just have to call the generated function GeneratedFunctionRegistry::RegisterAll(), which should have been generated from the json definition of the API.

A cheatsheet to define new extension APIs in custom shells

This is a summary of all the interfaces that must be implemented in a custom shell, and which files must be added or modified, to define new extension APIs.

- Write your custom _api_features.json and a .json or .idl definition of the API.
- Write your BUILD.gn so it triggers the necessary code generators for those files.
- Add your _api_features.json as a resource of type “BINDATA” to one of the shell’s .grd files.

- Implement your own ExtensionsBrowserAPIProvider.
- Implement your own ExtensionsBrowserClient and add the corresponding ExtensionsBrowserAPIProvider.
- Implement your own ExtensionsAPIProvider.
- Implement your own ExtensionsClient and add the corresponding ExtensionsAPIProvider.
- Implement your own DispatcherDelegate.
- Implement your own ExtensionsRendererClient and set the corresponding delegate to the Dispatcher.
- Implement your API extending UIThreadExtensionFunction for every extension function you declared.
- If the API is meant to be used in webpage-like contexts, it must be added to kWebAvailableFeatures in ExtensionBindingsSystem.