Plugins for Quink

Version 0.2
21 September 2013
Arthur Seaton

Table of Contents

Overview

Tl;dr - Read the appendices.

Quink is intended to support the addition of new functionality via plugins. Plugins should all be JavaScript tools. It's assumed that each plugin will consist of the following pieces:

- a plugin definition
- a plugin adapter script
- JavaScript libraries, markup, css etc

To use a plugin Quink has to be able to communicate with it and to exchange data. This is the role of the plugin adapter: it mediates between Quink and the plugin thus adapting the plugin for use inside Quink.

Interactions with the plugin are all initiated by the Quink user, picked up by Quink and finally passed through to the plugin adapter and on to the plugin itself. Quink communicates with the adapter via callback functions provided by the plugin adapter. The adapter communicates with Quink using Quink's publish/subscribe mechanism. Communication happens around plugin lifecycle events which are:

- load
- open
- save
- exit

The load event is slightly different to the others since it only ever happens once and Quink initiates this lifecycle phase by loading in the plugin adapter. Quink only loads plugins on request, so before the load event there is no plugin or plugin adapter available and there are no callbacks to invoke.

All other events can be initiated many times and are started by Quink invoking a callback function. The event ends when the plugin adapter publishes on a specific topic.

Plugin definitions file

Plugins are defined in `Quink/resources/plugins.json`. The file has two types of information: that which relates to all plugins (e.g. user interface data) and definitions for supported plugins.

User Interface Section

The user interface (`ui`) section holds configuration data that is used for all plugins. Currently the only item that can be specified here is the location of the plugin close menu button which can be specified as `ui.menu` entry.

The plugin close menu can be positioned in any of the four corners of the viewport. This is done using the `lateral` and `vertical` properties within the `menu` object. Each of the properties can take one of two values as shown in the table below.

Property		
lateral	left	right
vertical	top	bottom

When the plugin close button's position is configured at the top level in the plugin definitions file the configuration applies to all plugins. This configuration can be overriden on a per-plugin basis by providing a `ui.menu` entry within a plugin definition.

If no `ui.menu` entry is provided in the plugin definitions file the location defaults to top, right.

Plugins Section

The plugins are defined in a structure that looks like this:

```
"plugins": {
    "86": {
        "container": {
            "element": "div",
            "class": "svg",
            "alt-data-tags": ["data-drawing"],
            "alt-classes": ["drawing"]
        },
        "url": "Quink/pluginadapters/method-draw/MethodDrawAdapter.js",
        "name": "Method Draw",
        "topic-prefix": "plugin.method-draw",
        "insert-key": "v"
    }
}
```

Each plugin is keyed on a single character which is identified in the file by a scan code. These codes are are used as the key bindings when inserting new objects of the type supported by the plugin. In the example above the code `86` (which is generated by the letter `v`) is used to invoke method draw to insert svg drawings. So, when in command mode within Quink, hitting `i` gets to the insert commands and `v` invokes method draw. The advantage of using the command key as the key for the plugin definition is that it guarantees that each plugin will have a unique key binding. Anything else wouldn't be valid json.

The example above contains an `insert-key` property which is used to document the scan code. At present his property is not used by Quink and exists purely for documentation, although Quink may make use of it at a later date to provide information to the user.

The container section

Each plugin definition has to have a way to identify an area of the document as being of interest to that plugin and a way to define how objects created by the plugin are inserted into the document. These tasks are achieved through the `container` section.

The container section must have `element` and `class` properties. These are used by Quink to create a container for new plugin objects within the main document. The `container` section also lets you define a list of alternate css classes, data tags or element types that will contain data handled by the plugin. Along with the `element` and `class` properties these are used to identify which plugin will handle any edits initiated by the user via a double tap on an existing object.

In the example above double tapping on an object that has a tag such as `data-drawing` or on an object which has a css class of `drawing` will result in Quink opening the Method Draw plugin to edit that object.

There are some issues to understand when defining containers for plugins. It's important not to have multiple plugins defined that use the same container identifiers. When this happens it's not possible to guarantee which plugin will be used to edit the container content, so avoid this situation.

Note that the data that's passed to the plugin for editing will be the content of the container that matches within the container section and this does not include the matched container itself. In the example above Method Draw needs to be passed the serialised version of the `svg` element. If the container section had an entry of the form:

```
	"alt-elements": ["svg"]
```

when this entry matched it would result in Quink passing the content of the `svg` element to method draw. The content doesn't include the `svg` element itself, so method draw wouldn't be able to work with this.

Additional plugin properties

Each plugin must provide a JavaScript adapter. The plugin adapter is responsible for loading in the plugin and communicating with Quink as events occur within the plugin lifecycle. The `url` entry above shows the location of the adapter for the method draw plugin. Plugin adapters are expected to live in the `Quink/pluginadapters` directory. This is separate from core Quink (which lives in `Quink/js`) since the adapters are only downloaded if needed and so shouldn't be included in the built Quink core file.

The `name` entry in the plugin definitions file contains the name that will be displayed by Quink when insert is done via the toolbar[1].

The `topic-prefix` entry provides the prefix used by the plugin adapter when publishing messages around lifecycle events. All messages coming from the plugin adapter will have this prefix.

Plugin adapters

Each plugin needs to have a plugin adapter that adapts the plugin for use within Quink. Plugin adapters should live in `Quink/pluginadapters/<plugin-name>` directory, so each plugin adapter lives in its own directory along with any other artifacts that it needs (such as markup, css etc).

Currently plugin adapters are expected to be written using `require.js` so that they can access the plugin adapter context module that is provided by Quink. When the adapter code is loaded the

1 Not implemented as yet.

adapter in turn has to load the plugin and make it ready for use. The plugin should not be made visible on screen during the `load` event, this is done later as part of the `open` event. The reason for this is twofold: first to cleanly split the two events and to make sure that plugins are only loaded once and not on each subsequent invocation. Second to ensure that there's only a single way to open a plugin.

A typical form for an adapter file is shown in Appendix A – A Skeleton Plugin Adapter. From this it can be seen that when loaded the plugin adapter loads in the plugin then publishes the callback functions. These callback functions are used by Quink to initiate further lifecycle events.

`Underscore` and `jQuery` are used by Quink and can be used in the plugin adapter as needed. `PluginAdapterContext` is the only other module that is expected to be needed by an adapter implementation.

PluginAdapterContext

The `PluginAdapterContext` is a module that can be used by plugin adapter implementations. It allows access to core Quink functionality that will be needed by the adapter.

The functions provided within the plugin adapter context are:

- `adapterUrl(fileName)` – this returns the path within the Quink plugin adapter directory for the given file. Any markup needed by the plugin adapter should be placed in the same directory as the adapter code itself and referenced within the adapter code using this function.
- `pluginUrl(fileName)` – returns the path within the Quink plugins directory for the given file. This is the directory in which the plugin itself lives.
- `publish(suffix, data)` – this uses the Quink publish/subscribe mechanism to publish a message. The publication will have a topic which is created from the `topic-prefix` entry in the plugin definitions file with the given suffix appended. Any data passed to this function will be published under this topic.
  This is how the plugin adapter communicates around lifecycle events.

Files within the plugins and plugin adapters directories can be retrieved at runtime because these directories are not part of core Quink and so are not included built Quink file.

The Plugin lifecycle

As plugins are loaded and used they go through a number of lifecycle events. All events are initiated by Quink and complete when the plugin adapter publishes on a specific topic. Plugin events are:

- load
- open
- save
- exit

Plugin topics all have a standard format:

```
plugin.<plugin name>.<lifecycle event>
```

The `plugins.json` file has a `topic-prefix` entry which specifies the standard prefix for all topics coming from the plugin. In the example above the topic prefix was:

```
      "topic-prefix": "plugin.method-draw"
```

which means that all publications coming from that plugin adapter have a topic that uses `plugin.method-draw` as the prefix. So the save publication would be sent on the topic:

```
      "plugin.method-draw.saved"
```

This convention is guaranteed when the adapter implementation uses the plugin adapter context to publish lifecycle events.

Note that the lifecycle event name used in the topic is the past tense form of the lifecycle event name to indicate that the event has completed. E.g. the `open` event is complete when the adapter publishes on the `opened` topic.

Some plugin events, such as `load` and `save`, require data to be sent with the message.

Load

The purpose of the `load` event is to download the plugin artifacts (JavaScript, markup etc) and make them ready for use by Quink. The plugin shouldn't be made visible to the user within the `load` event.

The `load` event is slightly different to the other lifecycle events in that it can only ever happen once for each plugin and it's initiated by Quink loading the plugin adapter. All other lifecycle events can happen many times and are initiated by Quink invoking a callback function. Quink only loads plugins on request which has the advantage that Quink is always only as big as it needs to be[2]. Load on request means that when the user makes the initial request for a plugin, that plugin isn't available so Quink has to load it. Quink initiates the plugin load by loading the plugin adapter. It's the adapter's job to load in the plugin and this should be done when the adapter itself is initially loaded.

When the plugin adapter has loaded the plugin it has to indicate to Quink that the load is complete so that Quink can use the plugin. This communication is done by the adapter publishing a callback object on the `loaded` topic. The callback object has three properties, each named after the remaining lifecycle events (`open, save` and `exit`), with the values of these properties being a function within the adapter that can be invoked by Quink to initiate the event.

Quink provides a standard way for the user to control the use of plugins. This is in the form of a menu button and menu which is displayed on top of the plugin. The menu is displayed when the user taps the menu button and offers `save, continue` and `exit` options. Quink must then indicate to the plugin adapter the function that the user wants to be executed. The `save` and `exit` functions, provided by the adapter as part of the `loaded` event are used for this purpose.

Open

The `open` event is initiated by Quink invoking the `open` callback. `Open` is used to avoid having to reload all the plugin libraries and markup each time the plugin is invoked. The `open` function should re-use the plugin artifacts that have been previously downloaded to add the plugin markup to

2 Not quite true as currently Quink never unloads anything.

the DOM and show the plugin on the page ready to be used.

`Open` has the following signature:

```
      function open(data)
```

The `data` argument is used to supply the `open` function with any data that is to be used by the plugin. This will usually be the current state of whatever object the plugin operates on (e.g. an svg drawing). Edit operations are likely to supply data to `open`. Inserting a new object that is being created by the plugin is less likely to pass data into the `open` function.

The plugin adapter publishes on the `opened` topic when the adapter has completely visualised the plugin and it's ready to be used. No data is needed with this publication. It's used by Quink to decorate the plugin with the close menu button. Using the `opened` publication to add the decoration helps avoid updating the page multiple times as the state of the page is changed to show the open plugin.

Save

The `save` callback is invoked by Quink when the user indicates that they want to quit the plugin, saving the state of their work and adding the generated markup to the main document. The `save` callback has the following signature:

```
      function save()
```

It takes no arguments. The plugin adapter should publish on a `saved` topic and include the serialised data to be saved as part of the publication. The plugin manager will subscribe to this topic and insert the saved data into the main document at the right place.

The save callback is responsible for removing all visual trace of the plugin from the page as well as collecting up and serialising the data that is to be inserted into the document.

Exit

The `exit` callback is called by Quink when the user indicates, via the button menu, that they want to quit the plugin without saving any state changes. `Exit` takes no arguments

```
      function exit()
```

When the plugin adapter is ready to exit it should publish on an `exited` topic. No data is necessary for the `exited` message. As with the `save` callback, the `exit` function must remove the plugin from the screen.

Although the explanation above may seem lengthy[3], the adapter for method draw is less than 100 lines of executable code (written in a fairly non-terse style and including diagnostic output).

Message publication

The plugin adapter is expected to publish on topics around plugin lifecycle events. In order to do

3 Error handling is yet to be taken care of. Currently the plugin adapter just logs errors. One simple alternative is to publish an error message instead.

this the adapter should use Quink's publish/subscribe mechanism which is made available to the adapter via the `PluginAdapterContext`. The context provides a `publish` function to which the adapter supplies a topic suffix and any data. The topic prefix is added by the context using the `topic-prefix` property from the plugin definitions file.

```
      Context.publish(<topic-suffix>, <message>);
```

All publications from a plugin adapter will have the same topic prefix. The topic suffixes provided by the adapter are the `ed` form of the lifecycle events (`loaded, opened, saved, exited`).

Currently Quink assumes that plugin adapters will be written using `require.js`, so the `PluginAdapterContext` can be `required`.

Some examples from the method draw adapter:

```
        Context.publish('opened');
        Context.publish('loaded', {
            open: open,
            save: save,
            exit: exit
        });
```

The second example shows the `loaded` event publishing the callbacks object.

Appendix A – A Skeleton Plugin Adapter

This shows a skeleton outline for a typical plugin adapter.

```
require([
    'Underscore',   // Can be used in the plugin adapter if needed
    'jquery',       // Can be used in the plugin adapter if needed
    'ext/PluginAdapterContext'   // Provides access to Quink functionality
], function (_, $, Env, Context) {
    'use strict';

    // Callback
    function save() {
        Get data to save from the plugin
        Remove the plugin from the screen
            Context.publish('saved', data);
    }

    // Callback
    function exit() {
        Remove the plugin from the screen
            Context.publish('exited');
    }

    // Callback
    function open(data) {
        Show the plugin on screen
        when opened...
            Context.publish('opened');
    }

    function fetchPluginArtifacts() {
        load plugin scripts, markup and css ready to use
        when plugin loaded...
            Context.publish('loaded', {
                open: open,
                save: save,
                exit: exit
            });
    }

    fetchPluginArtifacts();
});
```

Appendix B – Lifecycle Events

This table describes each of the lifecycle events.

Event	Tasks	Cardinality	Initiated	Callback	Topic
Load	Load the plugin artifacts into Quink ready for use. Do not show them on screen.	Once per plugin.	Quink loads the plugin adapter. The plugin adapter should load the plugin.	None. Initiated when the plugin adapter is loaded.	`loaded` A callback object is published that contains the `open, save` and `exit` callback functions.
Open	Shows the plugin ready for use. If data is provided loads the plugin with that state.	Once every time the plugin is used for insert or edit.	Quink invokes the `open` callback with any data that is to be worked on by the plugin.	`open` Data can be provided if the plugin is to work on existing state (e.g. edit).	`opened` No data.
Save	Serialises the edit state of the plugin and publishes that state to be included in the document. Removes the plugin from the page.	Once every time the user wants to save plugin state.	The user selects `save` from the plugin menu. Quink invokes the `save` callback.	`save` No arguments.	`save` The serialised data to be saved is published.
Exit	Remove the plugin from the page discarding any changed state.	Once every time the user wants to exit the plugin without saving.	The user selects `exit` from the plugin menu. Quink invokes the `exit` callback.	`exit` No arguments.	`exited` No data.