

How to write a Plugin?

Last Modified on 02/01/2023 2:18 pm IST

To build your own plugin, you'll need to go through a few simple steps:

- Import the player library.
- Implement the plugin API methods and properties.
- Register the plugin in the player framework.

In this guide, we'll go through these steps and elaborate with examples of both Plain JavaScript & ES6.

Table of Contents

- [Base Plugin API](#)
 - `constructor(name, player, config)`
 - [Properties](#)
 - `config`
 - `player`
 - `name`
 - `logger`
 - `eventManager`
 - [Properties to Implement](#)
 - `(static) defaultConfig`
 - [Building methods](#)
 - `getConfig(attr)`
 - `updateConfig(update)`
 - `getName()`
 - `dispatchEvent(name, payload)`
 - [Methods to Implement / hooks](#)
 - `(static) isValid(player)`
 - `getUIComponents()`
 - `loadMedia()`
 - `destroy()`
 - `reset()`
- [Register Plugin API](#)
 - `registerPlugin(pluginName, pluginClass)`
- [Writing a Basic Plugin](#)
 - [Step 1 - Import the Player Library](#)

- [Step 2 - Create Your Plugin Class](#)
- [Step 3 - Implement the `BasePlugin` Methods](#)
- [Step 4 - Fill in a Simple Implementation](#)
- [Step 5 - Register Your Plugin](#)
- [Step 6 - Configure Your Plugin](#)
- [Writing an Advanced Plugin](#)

The player core library exposes two main utilities:

- `BasePlugin` - The class that your plugin will inherit from.
- `registerPlugin` - The static player method that will register your plugin in the player registry.

Let's take a look at the `BasePlugin` API.

Base Plugin API

```
constructor(name, player, config)
```

Parameters

| Name | Type | Description |
|---------------------|----------------------------------|-----------------|
| <code>name</code> | <code>string</code> | Plugin name |
| <code>player</code> | <code>playkit.core.Player</code> | Player instance |
| <code>config</code> | <code>Object</code> | Plugin config |

Properties

```
config
```

Type: `Object` - The runtime plugin config.

```
player
```

Type: `playkit.core.Player` - The player instance.

```
name
```

Type: `string` - The plugin registry name.

```
logger
```

Type: `playkit.core.Logger` - The logger of the plugin.

`EventManager`

Type: `playkit.core.EventManager` - The plugin event manager.

Properties to Implement

`(static) defaultConfig`

Type: `Object` - The default plugin config.

Methods

`getConfig(attr)`

Parameters

| Name | Type | Description |
|-------------------|---------------------|-----------------------|
| <code>attr</code> | <code>string</code> | config key (optional) |

Returns: `any` - If an attribute is provided, returns the value of the config attribute. If not, returns the plugin config.

`updateConfig(update)`

Parameters

| Name | Type | Description |
|---------------------|---------------------|----------------------------|
| <code>update</code> | <code>Object</code> | New full or partial config |

Returns: `void` - Merges the update config with the existing config.

`getName()`

Returns: `string` - The plugin name.

`dispatchEvent(name, payload)`

Parameters

| Name | Type | Description |
|---------|--------|--------------------------|
| name | string | The event name |
| payload | Object | The event payload object |

Returns: `void` - Dispatches an event from the player.

Methods to Implement

`(static) isValid(player)`

Parameters

| Name | Type | Description |
|--------|---------------------|-----------------|
| player | playkit.core.Player | Player instance |

Returns: `boolean` - Whether the plugin is valid or not.

You can help the player instance API with your decision logic. For example, if you want to enable your plugin on a Chrome browser only, you can simply implement the following:

```
class MyPlugin extends BasePlugin {
  static isValid(player) {
    return player.env.browser.name === 'Chrome';
  }
}
```

`getUIComponents(uiComponents: KUIComponent\[\])`

Returns: `void` - merge the provided ui components array to the `config.ui.uiComponents` of the player configuration. see [here](#) for more details.

Equivalent to the `config.ui.uiComponents` of the [player configuration](#):

`loadMedia()`

Returns: `void` - Runs on player media loaded.

The player will call this method on [media loaded](#) event. Useful in case of logic depends on the media data / metadata

`destroy()`

Returns: `void` - Runs the plugin destroy logic.

The player will call this method before destroying itself.

```
reset()
```

Returns: `void` - Runs the plugin reset logic.

The player will call this method before changing media.

```
get ready()
```

Returns: `Promise<*>` - a Promise which is resolved when plugin is ready for the player load

Signal player that plugin has finished loading its dependencies and player can continue to loading and playing states. Use this when your plugin requires to load 3rd party dependencies which are required for the plugin operation. By default the base plugin returns a resolved plugin. If you wish the player load and play (and middlewares interception) to wait for some async action (i.e loading a 3rd party library) you can override and return a Promise which is resolved when the plugin completes all async requirements.

Now, lets take a look at the `registerPlugin` API.

Register Plugin API

```
registerPlugin(pluginName, pluginClass)
```

Parameters

| Name | Type | Description |
|--------------------------|--------------------------------------|------------------|
| <code>pluginName</code> | <code>string</code> | The plugin name |
| <code>pluginClass</code> | <code>playkit.core.BasePlugin</code> | The plugin class |

Writing a Basic Plugin

In this section, we'll learn how to write a simple plugin, one that doesn't require any installation or setup steps except for a player script in hand (remotely or locally).

In the examples we'll use the ES6 syntax.

Step 1 - Import the Player Library

1 . In your *index.html* file, include a path to the player script:

```
<!DOCTYPE html>
<html lang="en"> <head> <meta charset="UTF-8" /> <title>MyPlugin</title> <script src=""></script>
</head>
</html>
```

2 . Create a new file called *my-plugin.js* and include it in your `index.html` :

```
<!DOCTYPE html>
<html lang="en"> <head> <meta charset="UTF-8" /> <title>MyPlugin</title> <script src=""></script>
<script type="module" src=""></script> </head> <body></body>
</html>
```

Step 2 - Create Your Plugin Class

1 . In *my-plugin.js*, define the plugin class:

```
class MyPlugin { constructor(name, player, config) {}
}
```

2 . Extends the `BasePlugin` base class:

```
import {BasePlugin} from 'kaltura-player-js';

class MyPlugin extends BasePlugin { constructor(name, player, config) { super(name, player, config); }
}
```

Step 3 - Implement the `BasePlugin` Methods

In *my-plugin.js*, add the necessary methods and properties to override as explained above:

```
import {BasePlugin} from 'kaltura-player-js';

class MyPlugin extends BasePlugin { static defaultConfig = {};
  static isValid(player) {}
  constructor(name, player, config) { super(name, player, config); }
  destroy() {}
  reset() {}
}
```

Step 4 - Fill in Simple Implementations

To see the plugin in action, lets fill in some simple implementations:

```
import {BasePlugin} from 'kaltura-player-js';

class MyPlugin extends BasePlugin { _myCollection;
  static defaultConfig = { myValue: 1 };
  static isValid(player): boolean { return true; }
  constructor(name, player, config) { super(name, player, config); this.logger.debug('Hello from ' + this.getName()
+ ' plugin constructor!'); this._myCollection = [this.config.myValue]; this._addBindings(); }
  destroy() { this.logger.debug('Empty collection'); this._myCollection = []; }
  reset() { this.logger.debug('Reset collection'); this._myCollection = [this.config.myValue]; }
  _addBindings() { this.eventManager.listen(this.player, this.player.Event.SEEKED, () => {
this.logger.debug(this.player.currentTime + ' Added to my collection');
this._myCollection.push(this.player.currentTime); this.dispatchEvent('collectionUpdate', {collection:
this._myCollection}); }); }
}
```

Step 5 - Register Your Plugin

Use the factory `registerPlugin` method to register your plugin in the player framework:

```
<!DOCTYPE html>
<html lang="en"> <head> <meta charset="UTF-8" /> <title>MyPlugin</title> <script src=""></script>
<script type="module" src=""></script> </head> <body> <script>
KalturaPlayer.core.registerPlugin('myPlugin', MyPlugin); </script> </body>
</html>
```

Step 6 - Configure Your Plugin

All that's left now is to verify that the player activates the plugin during runtime.

1 . Include the new plugin name and plugin configuration in the player configuration.
For example:

```
var config = { plugins: { myPlugin: { myValue: 10 } }
};
```

2 . Let's expand our test page and add:

a . A placeholder div to the player:

```
<div id="player-div" style="width: 360px; height: 640px;"></div>
```

b . A basic player configuration:

```
var config = { log: { level: 'DEBUG' }, targetId: 'player-div', provider: { partnerId: 'YOUR_PARTNER_ID' },
plugins: { myPlugin: { myValue: 10 } }
};
```

c . Setup code.

```
var mediaInfo = {entryId: 'YOUR_ENTRY_ID'};
var player = KalturaPlayer.setup(config);
player.loadMedia(mediaInfo).then(function () { player.play();
});
```

3 . Connect everything together and you'll get:

```
<!DOCTYPE html>
<html lang="en"> <head> <meta charset="UTF-8" /> <title>MyPlugin</title> <script src=""></script>
<script src=""></script> </head> <body> <script> KalturaPlayer.core.registerPlugin('myPlugin', MyPlugin);
  var config = { log: { level: 'DEBUG' }, targetId: 'player-div', provider: { partnerId:
'YOUR_PARTNER_ID' }, plugins: { myPlugin: { myValue: 10 } } };
  var mediaInfo = {entryId: 'YOUR_ENTRY_ID'}; var player = KalturaPlayer.setup(config);
player.loadMedia(mediaInfo).then(function () { player.play(); }); </script> </body>
</html>
```

That's it! the player starts to play and your plugin is activated.

Plugin Boilerplate - and fully working example

You can find [here](#) a prepared and pre-configured plugin template

The template includes a full environment, written in [ECMAScript6](#) and [TypeScript](#), and transpired in ECMAScript5 using [webpack](#) and the [TypeScript compiler](#), includes all scripts needed for the CI/CD cycle, bundling, and version handling and includes ([Mocha](#) / [Karma](#)) based test environment and pre-configured [ESLint](#) static code analysis.

For Detailed instructions for using the template - follow the steps [here](#)