[Advanced users] How to write a plugin

Last Modified on 07/15/2026 11:51 am IDT

> **This article is designated for advanced users.**

About

You can extend the Kaltura Player by creating custom plugins. Plugins can add functionality, respond to player events, modify the user interface, or integrate with third-party services.

> The examples in this article use ES6 syntax.

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.

Plugin development workflow

To create a plugin:

1. Import the player library.
2. Create a plugin class that extends `BasePlugin`.
3. Implement the required properties and methods.
4. Register the plugin with the player.
5. Configure the plugin in the player configuration.

Base plugin API

> `constructor(name, player, config)`
>
> **Parameters**

Name	Type	Description
`name`	`string`	Plugin name
`player`	`playkit.core.Player`	Player instance

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

Name	Type	Description
config	Object	Plugin config

Available 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.

Base methods

> getConfig(attr)

Parameter	Type	Description
attr	string	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)

Parameter	Type	Description
update	Object	New full or partial config

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

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(uiCompoments: KPUIComponent[] )`
>
> 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.

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

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.

Register plugin API

> `registerPlugin(pluginName, pluginClass)`

Parameters

Name	Type	Description
`pluginName`	`string`	The plugin name
`pluginClass`	`playkit.core.BasePlugin`	The plugin class

Create 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).

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

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

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

```
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.

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.

```
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>
```

Plugin Boilerplate - and fully working example

A ready-to-use plugin template is available to help you get started.

The template includes:

- ECMAScript6 and TypeScript support
- webpack configuration
- ECMAScript 5 transpilation
- CI/CD scripts
- Version management
- Mocha and Karma test environments
- ESLint configuration

For setup instructions, see the plugin template documentation.

Copyright © 2025 Kaltura Inc. All Rights Reserved. Designated trademarks and brands are the property of their respective owners. Use of this document constitutes acceptance of the Kaltura Terms of Use and Privacy Policy.