Welcome, developers, to the Logi Actions SDK! This powerful toolkit allows you to create custom plugins that enhance the functionality of devices like the Logitech MX Creative Console and Loupedeck CT, Live, and Live S. Whether you're extending your workflow, adding new creative tools, or building innovative integrations, the Logi Actions SDK provides the flexibility and support you need to bring your ideas to life.

Built on the foundation of the Loupedeck SDK, the Logi Actions SDK leverages the power of C# to allow developers to create robust and customizable plugins. With our SDK, you can unlock the potential of these versatile devices, adding new features and controls tailored to a variety of professional and creative environments.

From this documentation, you will find everything you need to:

- **Create your own plugin project:** Set up your development environment and learn the essential building blocks for plugin development.
- **Develop and test your plugin:** Dive into the key components, APIs, and workflows that make up a Logi Actions plugin.
- **Package and distribute your plugin:** Learn how to package your plugin for distribution, ensuring it works smoothly for others.
- **Submit to the Logitech Marketplace:** Share your plugin with a growing community of users through the Logitech Marketplace, where it can help others elevate their creativity and productivity.

We’re excited to see what you’ll build with the Logi Actions SDK! Explore the documentation to get started on your journey toward creating incredible custom plugins.


VA

The Logi Actions SDK includes a command-line tool (`LogiPluginTool`) that allows developers to create plugin projects, package plugins, and verify them.

- Basic knowledge of .NET and C# development.
- A .NET IDE (e.g., Visual Studio Code, Visual Studio 2022 Community Edition or higher, or JetBrains Rider).
- A device for testing the plugin (e.g., Logitech MX Creative Console, Loupedeck CT, Loupedeck Live, Loupedeck Live S, or Razer Stream Controller).

1. Ensure you have the latest Logitech Options+ or Loupedeck software installed:
   - Logitech Options+: https://www.logitech.com/software/options.html
   - Loupedeck Software: https://loupedeck.com/downloads/
2. Install the latest .NET 8 SDK:
   - You can download it from https://dotnet.microsoft.com/download/dotnet/8.0
3. Install the LogiPluginTool as a .NET tool. Open a terminal and run the following command:

   ```
   dotnet tool install --global LogiPluginTool
   ```

4. Generate a template plugin project using the LogiPluginTool:

   ```
   logiplugintool generate Example
   ```

   where "Example" is the name of the plugin. The command creates a folder named `ExamplePlugin` in the current directory.
5. Navigate to the generated folder and build the template plugin solution:

   ```
   cd ExamplePlugin
   dotnet build
   ```

6. Confirm that the build produces a `.link` file in the Logi Plugin Service's Plugins directory:

Windows

```
C:\Users\USERNAME\AppData\Local\Logi\LogiPluginService\Plugins\ExamplePlugin.link
```

macOS

```
/Users/USERNAME/Library/Application
Support/Logi/LogiPluginService/Plugins/ExamplePlugin.link
```

> **Note**
>
> The `.link` file simply tells the Logi Plugin Service where to load your plugin from. If the installed plugin folder is present, the plugin from .link address is loaded first.

7. Launch Logitech Options+ or Loupedeck software and wait for the configuration UI to appear.

   In the Logitech Options+ configuration view, navigate to 'All Actions' and verify that 'ExamplePlugin' appears under the 'Installed Plugins' section. If the plugin is not shown on the list, go the Options+ settings and select 'Restart Logi Plugin Service'.

   In Loupedeck Software, Unhide the "Example" plugin on the "Hide and show plugins" tab of the Action panel. The plugin should be now shown in the UI.

You can use the .NET Hot Reload feature to automatically rebuild the plugin project and reload the plugin in the host software whenever a source code file is saved.

To start hot reloading, first navigate to the plugin project's `src` directory, then run the watch command:

Windows

```
cd ExamplePlugin\src\
dotnet watch build
```

macOS

```
cd ExamplePlugin/src/
dotnet watch build
```

More information about .NET Hot Reload: https://devblogs.microsoft.com/dotnet/introducing-net-hot-reload/

Plugins can be distributed via the Logitech Marketplace and Loupedeck Marketplace to all other users.

- Please ensure you have tested the plugin properly with the supported hardware and software.
- Ensure that your plugin complies with the Marketplace Approval Guidelines.
- Ensure that the plugin icon is in the metadata/ -subfolder under the plugin folder.
- Pack the plugin to the .lplug4 file. The instructions can be found below.
- Deliver the plugin using the submission form at https://marketplace.logitech.com/contribute.

A .lplug4 file is essentially a zip file with a specific format and a yaml-configuration file. The file format is registered with Logi Plugin Service and can be installed by double-clicking the file.

To create a plugin package, use the Logi Plugin Tool with the pack command:

```
logiplugintool pack ./bin/Release/ ./Example.lplug4
```

To validate a package use Logi Plugin Tool with the verify command:

```
logiplugintool verify ./Example.lplug4
```

.lplug4 packaging information:

- Please check that the metadata file matches the claimed operating system support.
- Logi Plugin Service includes a package installer that does all the needed work to install the plugin to the Service Plugin directory and run all needed installation methods.
- Recommended name for the .lplug4 package: pluginName_version.lplug4 example: SpotifyPremium_1_0.lplug4. ZIP archive must contain in the /metadata folder a loupedeckPackage.yaml file with plugin manifest in YAML format.

```
metadata/loupedeckPackage.yaml   -- Plugin manifest.
metadata/icon256x256.png         -- Plugin icon. This filename is searched by
default.
win/       -- Binaries for Windows version of plugin, add only if Windows is
supported.
mac/       -- Binaries for Mac version of plugin, add only if Mac is
supported.
```

- The YAML and the icon file are searched from the metadata/ folder of the .lplug4 package
- The YAML manifest has the following format (the user-modifiable fields are in <> brackets)

```
type: plugin4
name: <Name of the plugin>
displayName: <Display name of the plugin>
version: <version string>
author: <author id>
copyright: <copyright>

supportedDevices: <Note if you support only one , remove another>
  - LoupedeckCt
  - LoupedeckLive

pluginFileName: <Plugin file name>
pluginFolderWin: <Folder for Windows binaries, add only if Windows is
supported>
pluginFolderMac: <Folder for Mac binaries, add only if Mac is supported>
```

Mandatory fields:

- **type**: use “plugin4” for plugins.
- **name**: This is the unique ID for the plugin and cannot be changed after it's published in the marketplace. The field is limited to Latin small and capital letters, digits, underscore, and dash (regex: "[a-zA-Z0-9_-]+"). Note! the name cannot contain "Plugin" at the end.
- **displayName**: Name that is shown in the Marketplace and in Options+ or Loupedeck software.
- **version**: is major.minor[.build] Every part must be a decimal number. Examples: "1.0", "1.0.0", If you're delivering an updated version of the plugin, please ensure the version number is increased accordingly.
- **author**: Name of the author that will be displayed in Marketplace.
- **supportPageUrl**: Could be an URL of a support page or a "mailto:" link of an email address. For example, GitHub issues can be used here for getting feedback on the plugin. Examples: https://support.mycompany.com/f-a-q-support or mailto:foo@bar.com

- **license**: Select a license under which you want to share the plugin. Please ensure that the selected license is compatible with Marketplace Developer License Agreement. One compatible option with the Marketplace is the MIT license: The MIT License | Open Source Initiative. **Note:** GPL licenses are not compatible with the Marketplace.
- **licenseUrl**: URL to license.

Optional fields:

- **copyright**: Author copyright.
- **supportedDevices**: use “- LoupedeckCt” for Loupedeck CT (and/or) “- LoupedeckLive” for Loupedeck Live (and/or) "- RazerStreamControllerX" for Razer Stream Controller X. If you support only one, remove another.
- **homePageUrl**: A link to a webpage, which has more information about the plugin.
- **icon256x256**: optional custom path and name to an icon file in the package. By default, the icon is searched from the 'metadata/Icon256x256.png' file.
- **minimumLoupedeckVersion**: Minimum Logi Plugin Service version that is required to run the plugin. The version is major.minor[.build] Every part must be a decimal number. Examples: "4.0", "4.0.0".

Here is an example LoupedeckPackage.yaml file for Spotify Premium plugin, which supports both Windows and Mac:

```yaml
type: plugin4
name: SpotifyPremium
displayName: Spotify Premium
version: 1.0
author: Logitech
copyright: Logitech

supportedDevices:
  - LoupedeckCt
  - LoupedeckLive

pluginFileName: SpotifyPremiumPlugin.dll
pluginFolderWin: bin/win/
pluginFolderMac: bin/mac/

license: MIT
licenseUrl: https://opensource.org/licenses/MIT
homePageUrl: https://logitech.com
supportPageUrl: https://support.logitech.com/f-a-q-support
```

The capitalized terms used in this document have the same meaning as those defined in the Logitech Marketplace Developer Agreement.

Every new Digital Product and all new Digital Product versions (also known as updates), are subject to a verification and approval process performed by and at the sole discretion of the Logitech Marketplace team.

All new Digital Products and their subsequent updates are reviewed according to Logitech's availability at the time the Digital Product or its update is uploaded using this form, but Logitech doesn't guarantee the time frame for the review as it can take longer due to various factors.

Along with automated checks, we manually review each Digital Product and Digital Product update one-by-one, before it becomes publicly available on Logitech Marketplace or Loupedeck Martketplace. The Digital Product Developer will receive a notification as soon as the status of the review changes, or if there are any questions regarding the Digital Product or Digital Product update.

If you haven't heard from us within the next ten (10) working days after the Digital Product upload, please reach out to us at marketplace@logitech.com

Logitech reserves the right to repeat the review of Digital Products and Digital Product updates from time to time and withdraw approval for a Digital Product or Digital Product update if new information on their conformity with the approval criteria comes to Logitech's attention after such a review.

All Digital Products for Logitech Marketplace must meet these general approval criteria:

- Please ensure you have tested the plugin properly with the supported hardware and software.
- Ensure that the plugin icon is in the metadata/ -subfolder under the plugin folder.

- Pack the plugin to the .lplug4 file. The instructions can be found here.
- Deliver the plugin using the submission form at https://marketplace.logitech.com/contribute.

- Metadata file matches the claimed operating system support.
- Logi Plugin Service software includes a specific package installer that does all the needed work to install the plugin to the Logitech Plugin directory and run all needed installation methods. The input for Logi Plugin Package Installer is a ZIP archive with a .lplug4 extension. To build it, use the Logi Plugin Tool. With the Plugin Tool you can both package and validate the plugin.
- Recommended name for the .lplug4 package: pluginName_version.lplug4 example: SpotifyPremium_1_0.lplug4. ZIP archive must contain in the /metadata folder a LoupedeckPackage.yaml file with plugin manifest in YAML format.
- Please check your Digital Product complies with all the other requirements stated in the SDK Documentation around prerequisites, testing, debugging and distribution.
- All of the external links on your Digital Product page are valid, reachable from the internet, and relate to the Digital Product or Digital Product author.
- The Digital Product is compatible with Logitech Plugin Service and can be installed.
- During each upload of a Digital Product (or new version of such Digital Product), compatibility must be verified.

- The Digital Product Developer must accept the Logitech Marketplace Developer Agreement before submitting the Digital Product.
- The Digital Product Developer must provide their own end user license agreement (known as a Developer EULA) with any Digital Product.
- In case your Developer Created Content contains any open source software, this open source software is licensed under one of the following open-source licenses: (i) Apache 2.0 or (ii) the MIT License and duly complies at all times with their terms. The Developer Created Content should not be GPL2 or GPL3 licensed. (the “Open Source Requirements”).
- The Digital Product Developer must have in place adequate privacy agreements if the Digital Products collect any personal data, must comply with them and must have an adequate Privacy Policy in place in accordance with applicable legislation.

Logitech may establish additional criteria on a case-by-case basis. Logitech reserves the right to remove any Digital Product from Logitech Marketplace or Loupedeck Marketplace at any

time and at its sole discretion.

Features implemented by the Digital Product are subject to review and approval by Logitech, and must conform to the following criteria:

- The Digital Product does not implement features which are not related to its major functionality.
- The Digital Product does not implement any malicious features or features for additional promotion that would give it an unfair advantage.
- Logitech may establish additional criteria on a case-by-case basis.
- If you have any questions about the approval process, please email us at marketplace@logitech.com.

Logitech's demo plugin contains the following parts:

The plugin must implement both of these classes are the parent classes are abstract:

- `DemoPlugin` class (inherited from the `Plugin` abstract class) contains the plugin-level logic.
- `DemoApplication` class (inherited from the `ClientApplication` abstract class) contains the logic related to the client application.
- `metadata/LoupedeckPackage.yaml` file contains the plugin configuration.

To create a simple command, add to your plugin project a class inherited from the `PluginDynamicCommand` class. To alter the command appearance and behavior, change the properties and overwrite the virtual methods of this class.

As an example, let's add a simple command to the Demo plugin that mutes and unmutes the system sound. You can assign this command to a touch or a physical button on a console.

To toggle between mute and unmute, we send the `VK_VOLUME_MUTE` virtual-key code using one of the native methods provided by the Logi Actions Plugin API that works on both Windows and macOS.

You can find the `ToggleMuteCommand` class here: ToggleMuteCommand.cs

1. Open the Demo plugin solution in Visual Studio.
2. In the Solution Explorer, right-click on the DemoPlugin project and select Add > Class.
3. Enter ToggleMuteCommand.cs as the file name and click Add. The `ToggleMuteCommand` class opens for editing.
4. Inherit the `ToggleMuteCommand` class from the `PluginDynamicCommand` class:

```
class ToggleMuteCommand : PluginDynamicCommand
```

5. Create an empty, parameterless constructor and set the command display name, description, and group name in the parent constructor parameters:

```
public ToggleMuteCommand()
    : base(displayName: "Toggle Mute", description: "Toggles audio mute
state", groupName: "Audio")
{
}
```

6. Overwrite the `RunCommand` method that is called every time a user presses the touch or the physical button to which this command is assigned:

```

```

```
protected override void RunCommand(String actionParameter)
{

this.Plugin.ClientApplication.SendKeyboardShortcut(VirtualKeyCode.VolumeMut
e);
}
```

7. Start debugging and wait until the software is loaded.
8. Open the configuration UI.
9. Switch the Adapt to App to OFF Options+ or Dynamic mode to OFF in Loupedeck Software.
10. In the applications dropdown list, select Demo.
11. On the left pane, under Press Actions, expand the Demo node, then expand the Audio group and ensure that the Toggle Mute command is there.
12. Drag and drop the Toggle Mute command to any touch button.
13. Connect a console to your computer.
14. Check that the console shows the Toggle Mute command on the touch screen.
15. Press this button to mute and unmute your computer's audio.

To create a simple adjustment, add to the plugin project a class inherited from the `PluginDynamicAdjustment` class. To alter the command appearance and behavior, change the properties and overwrite the virtual methods of this class.

As an example, let's add a simple adjustment to the Demo plugin that increases or decreases the counter based on the number of ticks with which the encoder is rotated.

You can find the `CounterAdjustment` class here: CounterAdjustment.cs

1. Open the Demo plugin solution in Visual Studio.
2. In the Solution Explorer, right-click on the DemoPlugin project and select Add > Class.
3. Enter CounterAdjustment.cs as the file name and click Add. The `CounterAdjustment` class opens for editing.
4. Inherit the `CounterAdjustment` class from the `PluginDynamicAdjustment` class:

```
public class CounterAdjustment : PluginDynamicAdjustment
```

5. Add a private counter field and set it to `0`:

```
private Int32 _counter = 0;
```

6. Create an empty, parameterless constructor and set the command display name, description, and group name in the parent constructor parameters. To indicate that the adjustment has a reset functionality, set the parameter `hasReset` to `true`:

```
public CounterAdjustment()
    : base(displayName: "Counter", description: "Counts rotation ticks",
groupName: "Adjustments", hasReset: true)
{
}
```

7. Overwrite the `ApplyAdjustment` method that is called every time a user rotates the encoder to which this adjustment is assigned:

```
protected override void ApplyAdjustment(String actionParameter, Int32 diff)
{
    this._counter += diff; // Increase or decrease the counter by the
number of ticks.
}
```

8. Overwrite the `RunCommand` method that is called every time a user presses the encoder to which this command is assigned:

```
protected override void RunCommand(String actionParameter)
{
    this._counter = 0; // Reset the counter.
}
```

9. Plugin Service can draw the current adjustment value near the encoder. To enable that functionality, overwrite the `GetAdjustmentValue` method:

```
protected override String GetAdjustmentValue(String actionParameter) =>
this._counter.ToString();
```

10. To inform Plugin Service that the adjustment value has changed, call the `AdjustmentValueChanged` method:

```
protected override void ApplyAdjustment(String actionParameter, Int32 diff)
{
    this._counter += diff; // Increase or decrease the counter by the
number of ticks.
    this.AdjustmentValueChanged(); // Notify the Plugin service that the
adjustment value has changed.
}

protected override void RunCommand(String actionParameter)
{
    this._counter = 0; // Reset the counter.
    this.AdjustmentValueChanged(); // Notify the Plugin service that the
adjustment value has changed.
}
```

11. Start debugging and wait until the Software is loaded.
12. Open the configuration UI.
13. Switch the Adapt to App to OFF Options+ or Dynamic mode to OFF in Loupedeck Software.
14. In the applications dropdown list, select Demo.
15. On the left pane, under Rotation Adjustments, expand the Demo node, then expand the `Adjustments` group and ensure that the Counter adjustment is there.
16. Drag and drop the Counter command to any encoder.
17. Connect a console to your computer. The console shows the Counter command on the encoder screen.

18. Rotate the encoder to change the counter value.
19. Press the encoder to reset the counter value to `0`.

You can link your Logi Actions plugin to an application so that the plugin is activated when the application comes to the foreground.

You can find the `DemoApplication` class here: DemoApplication.cs

1. Open the Demo plugin solution in Visual Studio.
2. In the Solution Explorer, double-click the DemoApplication.cs file.
3. Modify the `GetProcessName` method that returns the process name of the supported application. Replace `DemoApplication` with the name of the application you want to link to the plugin:

```
protected override String GetProcessName() => "DemoApplication";
```

4. Start debugging and wait until the Options+ or Loupedeck software is loaded.
5. Open the configuration UI.
6. Switch the Adapt to App to ON in Options+ or Dynamic mode to ON in Loupedeck Software.
7. Connect a console to your computer. The console shows the default profile on the screen. This is the default profile when no supported application is in the foreground.
8. Start the application you linked with the plugin.
9. The device shows the Demo profile on the screen.
10. Change the active applications with `Alt+Tab` to see how the device switches between them.

1. It is possible to define several process names for the supported applications. In this case, override the `GetProcessNames` method instead of `GetProcessName`.

```
```

```
protected override String[] GetProcessNames() => new[] { "Ableton Live 10
Lite", "Ableton Live 10 Standard" };
```

2. You can set a process name filter instead of fixed application names. In this case, override the `IsProcessNameSupported` method instead of `GetProcessName`:

```
protected override Boolean IsProcessNameSupported(String processName) =>
processName.ContainsNoCase("CaptureOne");
```

Commands and adjustments can contain parameters. As an example, the "apply develop profile" command in the Lightroom plugin takes the preset file name as a parameter.

Parameters are especially useful when you cannot determine the number of similar commands or adjustments at the development stage. Consider Windows 10/11 Volume Mixer - you cannot predict how many channels it will have on different PCs. However, a plugin can implement a single "toggle mute" command (or "change volume" adjustment) that takes the channel name as a parameter - and serve them all.

The plugin needs to indicate that the action has a parameter, and provide a list of available parameters.

The list of parameters can change at any moment (for example if a user started Spotify that added a channel to Volume Mixer), and there is a way to notify the console about the change.

A command or adjustment can have only one string parameter. If the plugin needs to store more data associated with a parameter, it should treat the parameter as an ID and keep an internal dictionary that links this ID to any related data.

Plugin service treats a parameter as a random string. It is the plugin's responsibility to keep these parameters unique for every action.

To create a command with a parameter, add to the plugin project a class inherited from the `PluginDynamicCommand` class (same as for a simple command). However, commands with parameters use a different base constructor.

As an example, let's add a simple command to the Demo plugin that toggles 4 switches.

You can find the `ButtonSwitchesCommand` class here: ButtonSwitchesCommand.cs

1. Open the Demo plugin solution in Visual Studio.
2. In the Solution Explorer, right-click on the DemoPlugin project and select Add > Class.
3. Enter ButtonSwitchesCommand.cs as the file name and click Add. The `ButtonSwitchesCommand` class opens for editing.

4. Inherit the `ButtonSwitchesCommand` class from the `PluginDynamicCommand` class:

```
class ButtonSwitchesCommand : PluginDynamicCommand
```

5. Create an empty, parameterless constructor that calls the parameterless constructor of the base class. You need to define the display name, description, and group name separately for each parameter.

```
public ButtonSwitchesCommand() : base()
{
}
```

6. Add 4 parameters in the constructor using the `AddParameter` method:

```
public ButtonSwitchesCommand() : base()
{
    for (var i = 0; i < 4; i++)
    {
        // parameter is the switch index
        var actionParameter = i.ToString();

        // add parameter
        this.AddParameter(actionParameter, $"Switch {i}", "Switches");
    }
}
```

7. Add `_switches` Boolean array that keeps the current state of switches:

```
private readonly Boolean[] _switches = new Boolean[4];
```

8. Overwrite the `RunCommand` method that is called every time a user presses the touch or the physical button to which this command is assigned:

```
protected override void RunCommand(String actionParameter)
{
    if (Int32.TryParse(actionParameter, out var i))
    {
        // turn the switch
        this._switches[i] = !this._switches[i];

        // inform service that command display name and/or image has
changed
        this.ActionImageChanged(actionParameter);
    }
}
```

9. Overwrite the `GetCommandDisplayName` method that is called every time Plugin Service needs to show a command on the console or the configuration UI.

Note that if your command does not change the display name during runtime, you don't need to override this method. Plugin Service uses display names that the plugin specifies with the `ActionImageChanged` method in the class constructor.

```
protected override String GetCommandDisplayName(String actionParameter,
PluginImageSize imageSize)
{
    if (Int32.TryParse(actionParameter, out var i))
    {
        return $"Switch {i}: {this._switches[i]}";
    }
    else
    {
        return null;
    }
}
```

10. Start debugging and wait until the software is loaded.
11. Open the configuration UI.
12. Switch the Adapt to App to OFF Options+ or Dynamic mode to OFF in Loupedeck Software.
13. From the applications dropdown list, select Demo.
14. On the right pane, under Press Actions, expand the Demo node, then expand the Switches group and ensure that it contains 4 Switch commands.
15. Drag and drop more than one Switch command to any touch button.
16. Connect a console to your computer.
17. Check that the console shows the Switch commands on the touch screen.
18. Press the buttons and check how their text changes.

By default, when the Logi Plugin Service needs to draw a button image, it uses the display name of the command that is assigned to the button.

However, the plugin can change this behavior so that an image is shown instead of the command name. To inform Logi Plugin Service that a custom image should be used, the `GetCommandImage` method of the `PluginDynamicCommand` class must be overridden.

Moreover, if the plugin wants to change a button image at runtime when the command state changes, the plugin can call the `ActionImageChanged` method to inform Logi Plugin Service that the image should be redrawn.

In the example below, we will create a simple dynamic command that changes its state when the user presses the button. When the command state changes, the command requests redrawing the button image.

You can find the `ThumbUpDownCommand` class here: ThumbUpDownCommand.cs

1. First, create a `ThumbUpDownCommand` dynamic command that toggles the internal boolean `_isThumbDown` variable on every button press. See Add a simple command for more information about creating dynamic commands.

```
namespace Loupedeck.DemoPlugin
{
    using System;

    public class ThumbUpDownCommand : PluginDynamicCommand
    {
        private Boolean _isThumbDown = false;

        public ThumbUpDownCommand() : base(displayName: "Thumb up/down",
description: null, groupName: "Switches")
        {
        }

        protected override void RunCommand(String actionParameter)
        {
            this._isThumbDown = !this._isThumbDown;
        }
```

```
    }
}
```

2. Add two images for "Thumb up" and "Thumb down" states to the plugin project. Build action for these files must be set to "Embedded Resource". The images "ThumbUp.png" and "ThumbDown.png" can be fetched from: DemoPlugin/images

   Note that the buttonimages must be in PNG format and have a size of 80x80 pixels.

3. Add two string class members that will hold the image resource paths. In the constructor, set these members to the full path of these files. Note that call to `PluginResources.FindFile()` method eliminates the need to know the exact path to these files (see Accessing plugin resource files).

```
        private readonly String _imageResourcePathThumbUp;
        private readonly String _imageResourcePathThumbDown;

        public ThumbUpDownCommand() : base(displayName: "Thumb up/down",
description: null, groupName: "Switches")
        {
            this._imageResourcePathThumbUp =
PluginResources.FindFile("ThumbUp.png");
            this._imageResourcePathThumbDown =
PluginResources.FindFile("ThumbDown.png");
        }
```

4. Override the `GetCommandImage` method to return the right image based on the command state:

```
        protected override BitmapImage GetCommandImage(String
actionParameter, PluginImageSize imageSize)
        {
            var resourcePath = this._isThumbDown ?
this._imageResourcePathThumbDown : this._imageResourcePathThumbUp;
            return PluginResources.ReadImage(resourcePath);
        }
```

5. Call the `ActionImageChanged` method when the command state changes:

```
        protected override void RunCommand(String actionParameter)
        {
            this._isThumbDown = !this._isThumbDown;
            this.ActionImageChanged();
        }
```

   Note that calling `this.ActionImageChanged(null)` will redraw all the buttons currently shown on the device.

Here is an example.

Note: Don't use explicit fontsize parameters when drawing text. Logi Plugin Service will define the best font size per device.

Note that `MyImage.png` file must be added to the plugin project as an embedded resource.

```
protected override BitmapImage GetCommandImage(String actionParameter,
PluginImageSize imageSize)
{
    using (var bitmapBuilder = new BitmapBuilder(imageSize))
    {

bitmapBuilder.SetBackgroundImage(PluginResources.ReadImage("MyPlugin.EmbeddedR
esources.MyImage.png"));
        bitmapBuilder.DrawText("My text");

        return bitmapBuilder.ToImage();
    }
}
```

The `PluginResources` class provides helper methods to get plugin resources easily everywhere in the plugin code.

You can find the source code here: PluginResources.cs

For instance, the following methods can be used for finding and getting images:

```
public static String FindFile(String fileName) =>
PluginResources._assembly.FindFileOrThrow(fileName);

public static BitmapImage ReadImage(String resourceName) =>
PluginResources._assembly.ReadImage(PluginResources.FindFile(resourceName));
```

For a new plugin, the easiest way to take the plugin resources into use is to generate the plugin project with the Logi Plugin Tool (see Getting Started). The generated skeleton project contains the enabler code and an example of how to get plugin resources from the plugin code.

For an existing plugin, you can take the getting images into use as follows:

1. Download the PluginResources.cs file and include it in your plugin project.

2. In the `PluginResources.cs` file, change the namespace to the same one that your plugin project uses:

```
namespace Loupedeck.DemoPlugin
```

3. Initialize the `PluginResources` class in the constructor of your plugin class (replace the plugin class name `DemoPlugin` with your plugin class):

```
public DemoPlugin() => PluginResources.Init(this.Assembly);
```

After this, you can get image resources in your plugin code:

```
String resourceName = PluginResources.FindFile("MyImage.png");
BitmapImage myImage = PluginResources.ReadImage(resourceName);
```

A plugin icon is a graphical image that represents your plugin in user interfaces. The plugin icon is displayed in the Options+ and Loupedeck applications, as well as in the Logitech Marketplace.

Options+ supports both black and white backgrounds for icons. To ensure that the plugin icon displays correctly on different backgrounds, please follow these guidelines:

- Image resolution must be 256x256 px in PNG format.
- The actual icon graphic must fit within a 192x192 px area centered in the icon.
- The remaining area around the icon graphic must be transparent.

To add an icon for the plugin, add it to the plugin lplug4 package under the folder named "metadata":

- `metadata/Icon256x256.png`

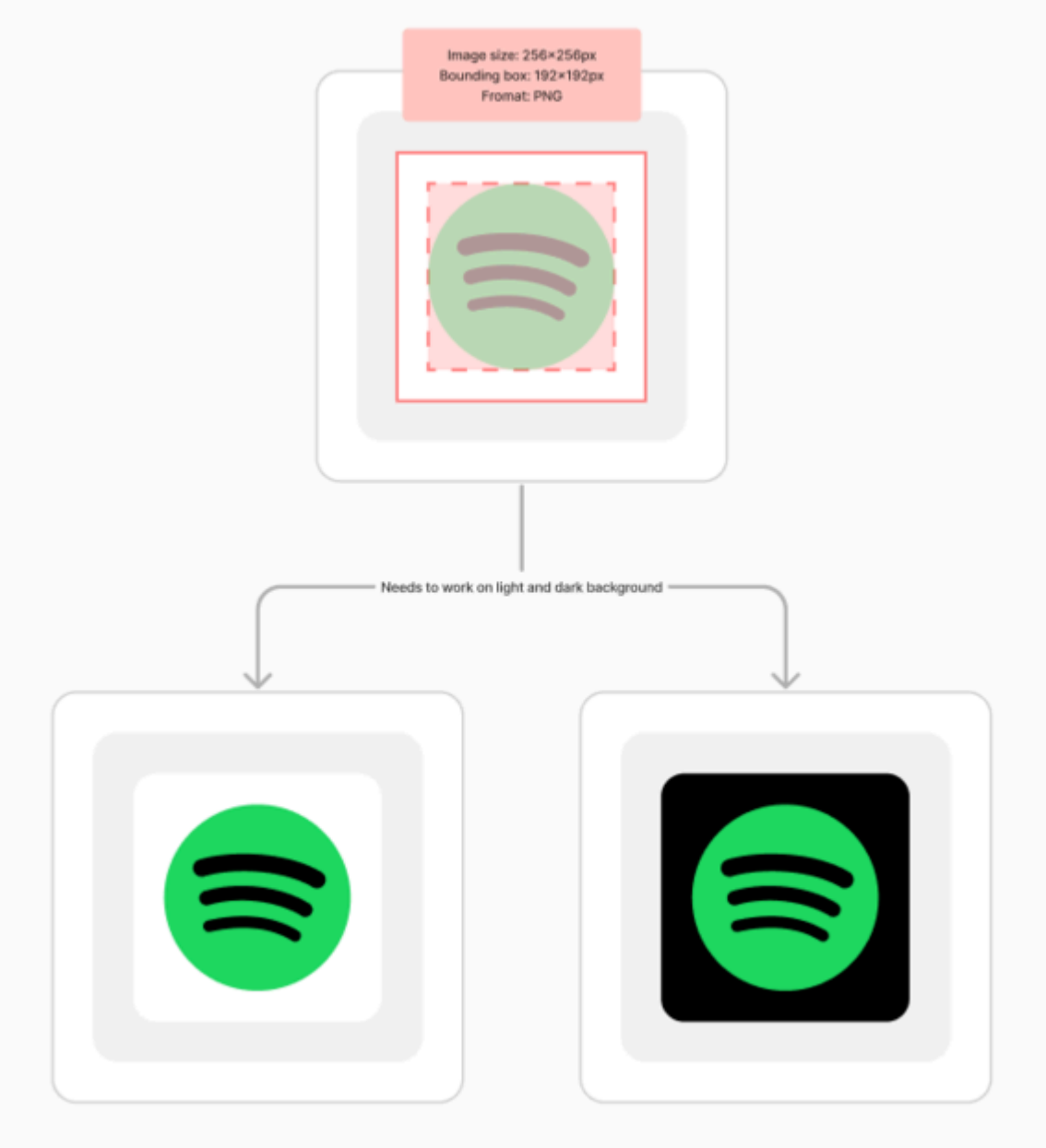


Image size: 256×256px
Bounding box: 192×192px
Fromat: PNG
Needs to work on light and dark background

Logi Plugin Service can use vector images instead of raster images when drawing action icons on device.

Currently only SVG format is supported as a vector format.

- Images returned by `PluginDynamicCommand.GetCommandImage()` and `PluginDynamicAdjustment.GetAdjustmentImage()` methods can either have raster (PNG) or vector (SVG) format.

Below is an example of a plugin command that reads an SVG file from plugin assembly embedded resources and returns it as command image.

```
namespace Loupedeck.TestPlugin
{
    using System;

    internal class VectorGraphicsDynamicCommand : PluginDynamicCommand
    {
        public VectorGraphicsDynamicCommand()
            : base("Vector graphics", "Command that has an SVG image", "Test
Group")
        {
        }

        protected override BitmapImage GetCommandImage(String actionParameter,
PluginImageSize imageSize)
            => BitmapImage.FromResource(this.Plugin.Assembly,
"Loupedeck.TestPlugin.VectorGraphicsDynamicCommand.svg");
    }
}
```

- If .LPLUG4 package has an `actionicons` folder in its root, then Logi Plugin Service first searches this folder for plugin action images.

- In this case no changes are required in plugin code.
- Images can either have raster (PNG) or vector (SVG) format.
- Image file name should consist of action class full name as its name and corresponding file extension, e.g. in the above example it should be "Loupedeck.TestPlugin.VectorGraphicsDynamicCommand.svg" ("Loupedeck.TestPlugin" from namespace name and `VectorGraphicsDynamicCommand` from class name).
- This is the preferred method.

Icon Templates define button appearance by specifying its image and text layout. These templates improve customization and maintain clarity when handling button designs. Icon Template files have the `.ict` extension and can be used across various precedence levels.

Icon Templates operate on several levels of precedence (from high to low), which guide their application in different contexts:

1. User Level:
   - Scope: Can be updated directly by users in the Icon Editor. Reset icon to default in the Icon Editor to remove the updates.
   - Location: Stored in the `ActionIcons` folder within the user profile directory.
   - Purpose: Allows personalized updates for icon designs unique to user-specific workflows.
2. Plugin Action Level (see ToggleMuteCommand.cs and its Icon Template as an example):
   - Scope: Template configurations for an individual action. Icon Template can be created and exported using the Icon Editor developer mode.
   - Location: Stored in the `icontemplates` folder of plugin packages. Action class full name should be used as the Icon Template file name.
   - Purpose: Provides plugin-specific customization and greater control over button appearance.
3. Plugin Level:
   - Scope: Default template configurations for individual plugins.
   - Location: `DefaultIconTemplate.ict` stored in the `metadata` folder of plugin packages.
   - Purpose: Ensures consistent plugin branding where specific configurations aren't defined.
4. Global Level:
   - Scope: The global default settings.

- Location: Built into Logi Plugin Service and not editable by plugin developers.
- Purpose: Acts as a fallback configuration to standardize appearance globally across plugins.

Here's a fully annotated example of an Icon Template:

```
{
  "backgroundColor": 4278869247, // Background color in ARGB format
  "items": [
    {
      "$type": "Loupedeck.Service.ActionIconImageItem, LoupedeckShared", // Is
being used for proper deserialization and is optional
      "image": "", // Encoded image string if applicable
      "imageFileName": null, // Reference to image file, if available
      "imageColor": 4294967295, // Tint color for the image in ARGB format
      "imageRotation": "None", // Image rotation
      "isVisible": true, // Indicates whether the item is visible
      "itemType": "Image", // Specifies that this item is an image
      "area": {
        "x": 15,
        "y": 0,
        "width": 70,
        "height": 70,
        "isFullScreen": true
      } // Defines placement and size of the image
    },
    {
      "$type": "Loupedeck.Service.ActionIconTextItem, LoupedeckShared", // Is
being used for proper deserialization and is optional
      "text": "Some text", // Default text display
      "textColor": 4294967295, // Text color in ARGB format
      "fontSize": 5, // Font size
      "fontName": "Brown Logitech Pan Light", // Font type
      "isVisible": true, // Indicates whether the item is visible
      "itemType": "Text", // Specifies that this item is a text component
      "area": {
        "x": 0,
        "y": 70,
        "width": 100,
        "height": 30,
        "isFullScreen": false
      } // Defines placement and dimensions for the text
    }
  ]
}
```

Key Notes

1. Icon Templates contain two optional items:

- `"ItemType": "Image"` for visual buttons.
- `"ItemType": "Text"` for text overlays on buttons.

2. `image` property is optional and can include encoded image data.
3. Visibility and placement are controlled with:
   - `area` properties (`x`, `y`, `width`, `height`) for positioning and sizing.
   - `isVisible` flags to determine whether an item is displayed.
4. Other properties such as `fontSize`, `imageRotation`, and `imageColor` enable additional customization.

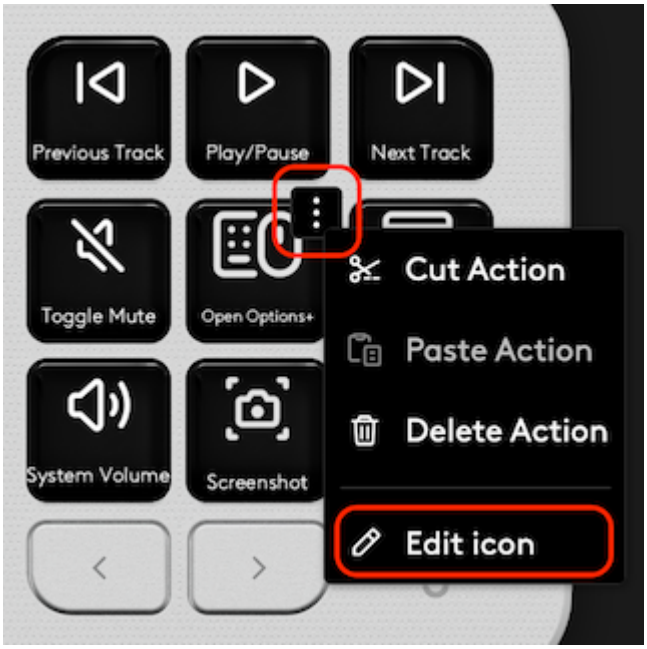
Icon Editor is a part of the Options+ and Loupedeck user interfaces that lets users customize action icons. It provides navigation controls and tools, allowing you to select an icon and modify its appearance or associated text.

For additional capability, developers can export Icon Templates via Icon Editor in developer mode:

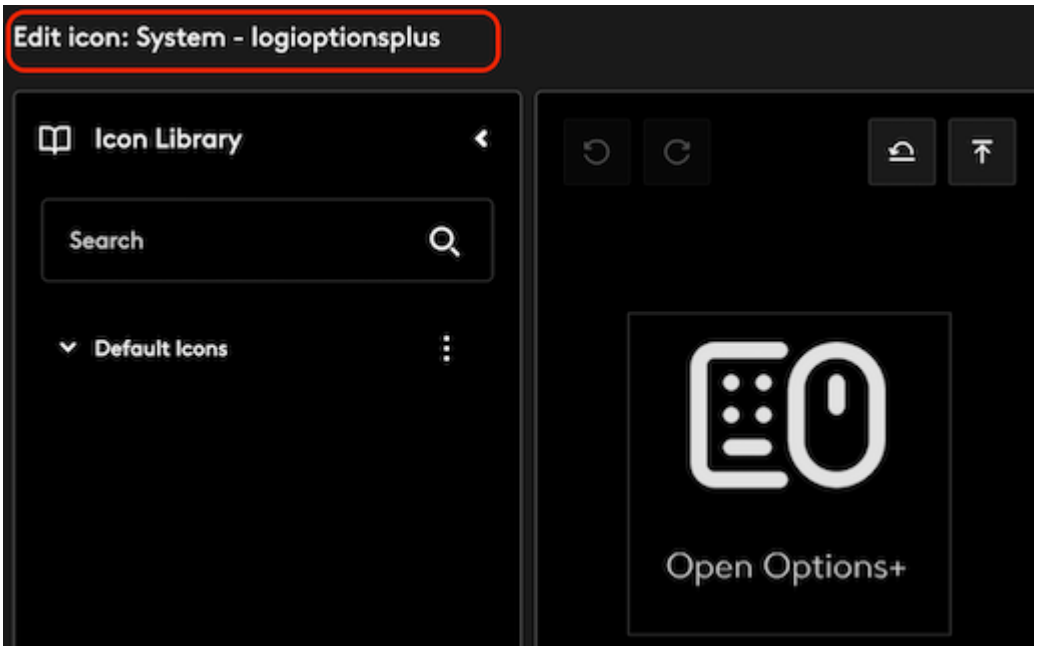
1. Enable developer mode:
    - Stop Logi Plugin Service.
    - Open the `LoupedeckSettings.ini` configuration file located in the Logi Plugin Service directory:
        - Path (Windows): `C:\Users\<user_name>\AppData\Local\Logi\LogiPluginService`
        - Path (macOS): `~/Library/Application Support/Logi/LogiPluginService`
    - Add the following line:

      ```
      Loupedeck/DeveloperMode=True
      ```

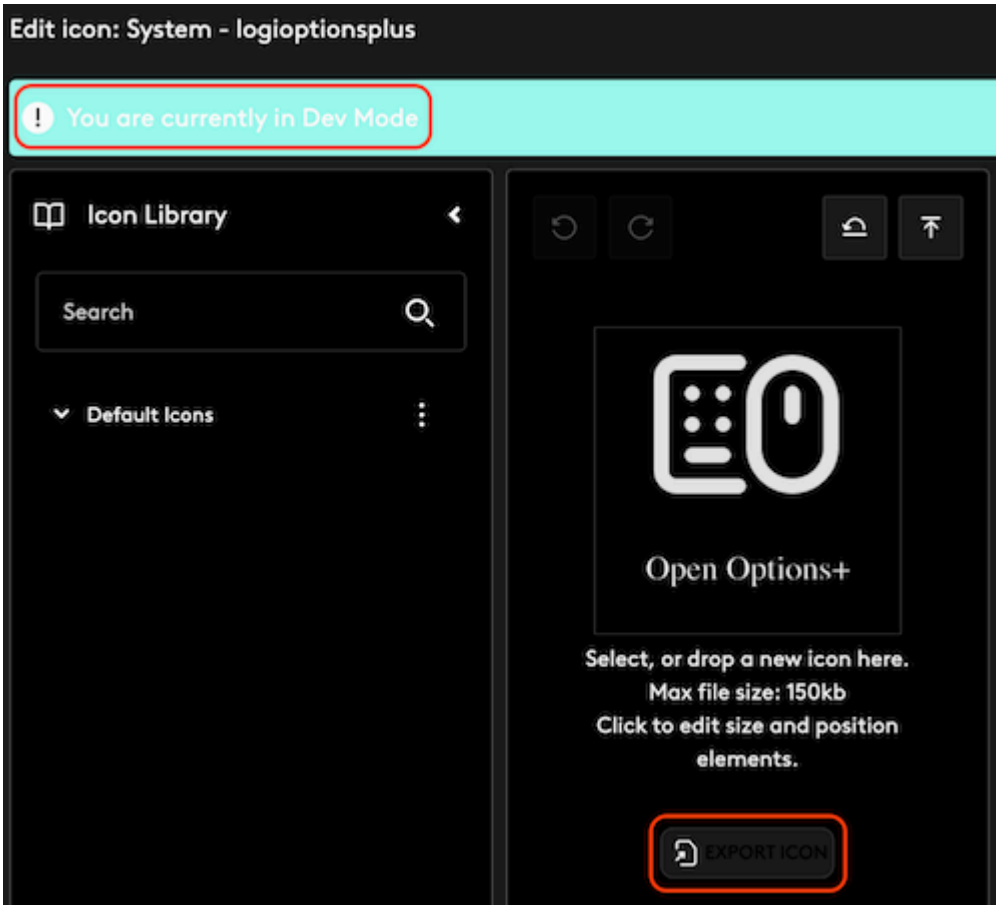
    - Start Logi Plugin Service.
2. Open Icon Editor.



3. Switch to developer mode by clicking on the "Edit Icon: ..." title in the Icon Editor interface.



4. Perform needed updates and use "Export Icon" button to export the Icon Template.


Edit icon: System - logioptionsplus
You are currently in Dev Mode
Icon Library
Search
Default Icons
Open Options+
Select, or drop a new icon here.
Max file size: 150kb
Click to edit size and position elements.
EXPORT ICON

For each plugin, Logi Plugin Service stores a collection of setting names and values. Here are some characteristics of plugin settings:

- Both setting names and values are of `String` type.
- The setting names are case-insensitive.
- Plugin settings are persistent and are stored encrypted.
- Any setting can be marked to be backed up in the cloud for the logged-in Logi user.

The following methods are available in the `Plugin` class.

```
protected Boolean TryGetPluginSetting(String settingName, out String
settingValue);
```

Returns a plugin setting.

Returns `true` if the setting exists and `false` otherwise.

If the setting does not exist, then `settingValue` is set to `null`.

```
protected void SetPluginSetting(String settingName, String settingValue,
Boolean backupOnline);
```

Saves a plugin setting.

Set `backupOnline` to `true` to backup this setting in the cloud and `false` to keep it only locally.

```
protected void DeletePluginSetting(String settingName);
```

Deletes a plugin setting.

```
protected String[] ListPluginSettings();
```

Returns a list of plugin setting names.

```
private String GetUserId()
{
    const String SettingName = "UserId";

    // first try to get existing user ID

    if (this.TryGetPluginSetting(SettingName, out var existingUserId))
    {
        return existingUserId;
    }

    // if it does not exist, generate a new one and save it

    var newUserId = Guid.NewGuid().ToString("N");

    this.SetPluginSetting(SettingName, newUserId, false);

    return newUserId;
}
```

- Storing Plugin Data

A dynamic folder (also known as "Control center") is a dynamic workspace that is fully controlled by a plugin.

- Like a normal workspace, a dynamic folder contains touch pages, encoder pages and wheel tools.
- Unlike a normal workspace, users cannot add any items to it, the whole content is defined by a plugin.

A dynamic folder is represented in the UI by a command that can be assigned to any touch or physical button. A dynamic folder can be opened by pressing this button on the device.

A dynamic folder can be closed

- by pressing a special "Back" button,
- by pressing the device Home button,
- by executing any change workspace or change page command, or
- when the active application is changed.

To add a command folder to a plugin, the developer needs to:

1. Add a class inherited from `PluginDynamicFolder` class to the plugin project;

```
public class TaskSwitcherDynamicFolder : PluginDynamicFolder
```

1. Set folder parameters in the constructor:

```
public TaskSwitcherDynamicFolder()
{
    this.DisplayName = "Alt+Tab";
    this.GroupName = "System";
    this.Navigation = PluginDynamicFolderNavigation.EncoderArea;
}
```

1. Define commands, adjustments and wheel tools that this workspace will contain;
2. (optionally) Define actions display names, images and action behavior.

- To execute some code at folder loading, override `Load` method (is called during plugin load).
- To execute some code at folder unloading, override `Unload` method (is called during plugin unload).

Do not execute any code in the dynamic folder constructor, except setting folder parameters.

- To execute some code at folder activation, override `Activate` method (the method is called when the first instance of the folder is open on the connected devices).
- To execute some code at folder deactivation, override `Deactivate` method (the method is called when the last instance of the folder is closed on the connected devices).

As an example, in `Activate` method you may want to subscribe to application events, and in `Deactivate` method to unsubscribe from those. The plugin does not need to process application events if the dynamic folder is not visible on the device.

By default, the button that opens a dynamic folder shows the dynamic folder display name defined in the constructor:

```
public NumpadDynamicFolder()
{
    this.DisplayName = "Numeric Pad";
    this.Navigation = PluginDynamicFolderNavigation.None;
}
```

However, it is possible to change the display name runtime by overriding `GetButtonDisplayName` method:

```
public override String GetButtonDisplayName(PluginImageSize imageSize) => $"
```

```
{this.ChannelCount} Channels";
```

It is also possible to use an image instead of text in this button by overriding `GetButtonImage` method, for example:

```
protected override BitmapImage GetButtonImage(PluginImageSize imageSize)
{
    var bitmapImage =
PluginResources.ReadImage("Loupedeck.DemoPlugin.Images.ButtonImage.png");
    return bitmapImage;
}
```

These methods should return `null` if the display name or the image is not available.

See Accessing plugin resource files for more information about using `PluginResources` class.

The following modes are available:

- `None` - navigation is fully done by the plugin developer.
- `ButtonArea` - "Back" button is automatically inserted on every touch page in the left top corner. This is the default mode.
- `EncodeArea` - "Back" button is automatically inserted at the top of the left encoder page area. This is possible only if the dynamic folder does not define any encoder actions (neither rotation nor reset ones).

Navigation mode can be changed by setting the `Navigation` property in the constructor.

If the plugin developer sets the navigation mode to `None` then she can insert navigation buttons in the code.

The following action names should be used:

- `PluginDynamicFolder.NavigateUpActionName` - for "Back" button;
- `PluginDynamicFolder.NavigateLeftActionName` - for "Previous Touch Page" button;
- `PluginDynamicFolder.NavigateRightActionName` - for "Next Touch Page" button;

Use the following methods:

- `GetButtonPressActionNames` - to define touch button commands;

- `GetEncoderRotateActionNames` - to define encoder adjustments;
- `GetEncoderPressActionNames` - to define encoder "reset" commands;
- `GetWheelToolNames` - to define wheel tools.

The plugin developer should use these methods to create action names:

- `CreateCommandName` - for commands;
- `CreateAdjustmentName` - for adjustments.

A dynamic folder will automatically create more pages if the actions do not fit on one page.

A dynamic folder will automatically add navigation actions depending on the selected navigation mode.

```
public override IEnumerable<String> GetButtonPressActionNames()
{
    return new[]
    {
        PluginDynamicFolder.NavigateUpActionName,
        this.CreateCommandName("7"),
        this.CreateCommandName("8"),
        this.CreateCommandName("9"),
        this.CreateCommandName("."),
        this.CreateCommandName("4"),
        this.CreateCommandName("5"),
        this.CreateCommandName("6"),
        this.CreateCommandName("0"),
        this.CreateCommandName("1"),
        this.CreateCommandName("2"),
        this.CreateCommandName("3")
    };
}
```

A dynamic page can inform the Loupedeck service that the list of actions or wheel tools has changed by calling these methods:

- `ButtonActionNamesChanged` - for commands (including "reset" commands of encoders)
- `EncoderActionNamesChanged` - for adjustments

A dynamic folder can define display names and images for commands and adjustments by overriding the following methods:

- `GetCommandDisplayName`
- `GetCommandImage`
- `GetAdjustmentDisplayName`

- `GetAdjustmentImage`

These methods should return `null` if the display name or the image is not available.

The Loupedeck service first tries to get the image and then, if no image is available, uses the display name as the button image.

```
public override String GetCommandDisplayName(String actionParameter,
PluginImageSize imageSize) =>
    this.TryGetNativeApplication(actionParameter, out var app) ?
app.DisplayName : "Unknown";

public override BitmapImage GetCommandImage(String actionParameter,
PluginImageSize imageSize) =>
    this.TryGetNativeApplication(actionParameter, out var app) ?
app.Icon?.ToImage() : null;
```

To return the adjustment value, override the `GetAdjustmentValue` method:

```
public override String GetAdjustmentValue(String actionParameter) =>
    this.TryGetVolume(actionParameter, out var volume) ? volume.ToString("D")
: null;
```

To inform the Loupedeck service that the adjustment value has changed, call the `AdjustmentValueChanged` method:

```
private void OnVolumeChanged(Object sender, VolumeMixerItemEventArgs e) =>
    this.AdjustmentValueChanged(e.Id);
```

- `RunCommand` - override this method to execute commands
- `ApplyAdjustment` - override this method to apply adjustments

```
public override void RunCommand(String actionParameter)
{
    if (Int32.TryParse(actionParameter, out var processId))
    {
        this._nativeMethods.ActivateProcess(processId);
    }
}
```

The folder can also be closed programmatically with `this.Close();`.

This might be useful when the dynamic folder is used to switch between several options. But please be consistent with the use. The recommendation is to either close the folder after each command or don't close it at all and let the user navigate out of it.

```
public override void RunCommand(String actionParameter)
{
    # execute the selected command here

    # close the folder after the command
    this.Close();
}
```

There's no programmatic way to open the folder but the Logi Plugin Service controls this.

As an alternative to reacting to actions, the plugin developer might choose to react to low-level commands:

- `ProcessButtonEvent` - override to process physical button presses;
- `ProcessEncoderEvent` - override to process encoder rotations;
- `ProcessTouchEvent` - override to process touch events.
    - TouchDown
    - TouchUp
    - LongPress
    - LongRelease
    - Tap
    - DoubleTap
    - Move
    - HorizontalSwipe
    - VerticalSwipe
    - TwoFingerTap

If the dynamic folder handles a low-level event, it should return `true`.

```
public override Boolean ProcessButtonEvent(String actionParameter,
DeviceButtonEvent buttonEvent)
{
    if (buttonEvent.IsPressed)
    {
```

```
            this.SendKeyboardShortcut(actionParameter);
            return true;
        }

        return false;
    }
```

Profile action is a special type of actions with parameters, but there are some key differences.

With profile actions:

- UI does not show all possible parameter values in the actions list;
- in most cases it is not possible to predict the parameter list;
- parameters are entered by users in UI;
- profile actions with actual parameters are stored in application profiles.

They are called *profile* actions because the actual profile actions are stored in application *profiles*.

Existing actions with parameters can be easily converted to profile actions by calling the `MakeProfileAction()` method in the dynamic action constructor.

- `"text"` - any text, is represented in UI as a label and a text box.
- `"execute"` - path to the executable file to run.
- `"list"` - an item from the list, is represented in the UI as a label and a combo box.
- `"tree"` - multi-level selection; currently only 2-level combo boxes are supported.

The plugin provides data for "list" and "tree" actions by overriding the `GetProfileActionData()` method of dynamic action.

The profile action type can be complemented with a label to show in UI, e.g.:

- `"text;Enter chat message to send:"`
- `"list;Select album to play:"`

As an example, we will implement a profile action that sends a user-defined chat message.

1. In the plugin project, create a class based on either `PluginDynamicCommand` or `PluginDynamicAdjustment` base class.
2. Call the `MakeProfileAction` method in the class constructor:

```
this.MakeProfileAction("text;Enter chat message to send:");
```

1. Add code to execute the command. In this example, it is as simple as starting the application URI:

```
protected override void RunCommand(String actionParameter) =>
Chat.SendMessage(actionParameter);
```

As an example, we will implement a profile action that starts Windows Settings applications.

Windows Settings applications are grouped in categories, so UI should show two levels of combo boxes: for categories and for applications within a selected category.

1. In the plugin project, create a class based on either `PluginDynamicCommand` or `PluginDynamicAdjustment` base class.
2. Call the `MakeProfileAction` method in the class constructor:

```
this.MakeProfileAction("tree");
```

1. Override `GetProfileActionData` method to return tree data.

```
protected override PluginProfileActionData GetProfileActionData()
{
    // create tree data

    var tree = new PluginProfileActionTree("Select Windows Settings
Application");

    // describe levels

    tree.AddLevel("Category");
    tree.AddLevel("Application");

    // add data tree

    var categoryNames = this._applications.Values.Select(a =>
```

```
a.CategoryName).Distinct();

    foreach (var categoryName in categoryNames)
    {
        var node = tree.Root.AddNode(categoryName);

        var items = this._applications.Values.Where(a =>
a.CategoryName.EqualsNoCase(categoryName));

        foreach (var item in items)
        {
            node.AddItem(item.ApplicationUri, item.ApplicationName, null);
        }
    }

    // return tree data

    return tree;
}
```

1. Define display names for each parameter:

```
protected override String GetCommandDisplayName(String actionParameter,
PluginImageSize imageSize) =>
    this._applications.TryGetValue(actionParameter, out var application) ?
application.ApplicationName : null;
```

1. Add code to execute the command. In this example, it is as simple as starting the application URI:

```
protected override void RunCommand(String actionParameter) =>
Process.Start(actionParameter);
```

- Actions with parameters

Logi Plugin Service provides a possibility for plugins to store data locally.

Use `Plugin.GetPluginDataDirectory()` method to get plugin data folder path.

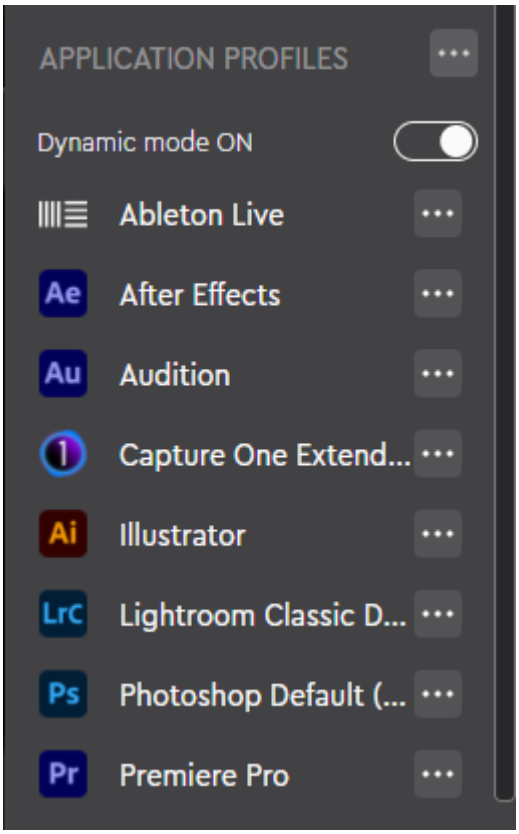
Call `IoHelpers.EnsureDirectoryExists(String path)` method to ensure the given directory exists.

```
var pluginDataDirectory = this.GetPluginDataDirectory();
if (IoHelpers.EnsureDirectoryExists(pluginDataDirectory))
{
    var filePath = Path.Combine(pluginDataDirectory, "MyData.bin");
    using (var streamWriter = new StreamWriter(filePath))
    {
        // Write data
    }
}
```

- Managing Plugin Settings

In terms of using the applications plugins are divided into two classes:

- The first class is the plugins for applications. These plugins are visible in the application section; they generally require an application to be in the foreground to execute commands.
- The other type of plugins do not require an application to be in the foreground or any application to be running locally at all. For example, Twitch and Philips Hue plugins are using remote services directly.



You can specify whether your plugin requires an associated application by changing the following flag in your Plugin class:

```
public override Boolean HasNoApplication => true;
```

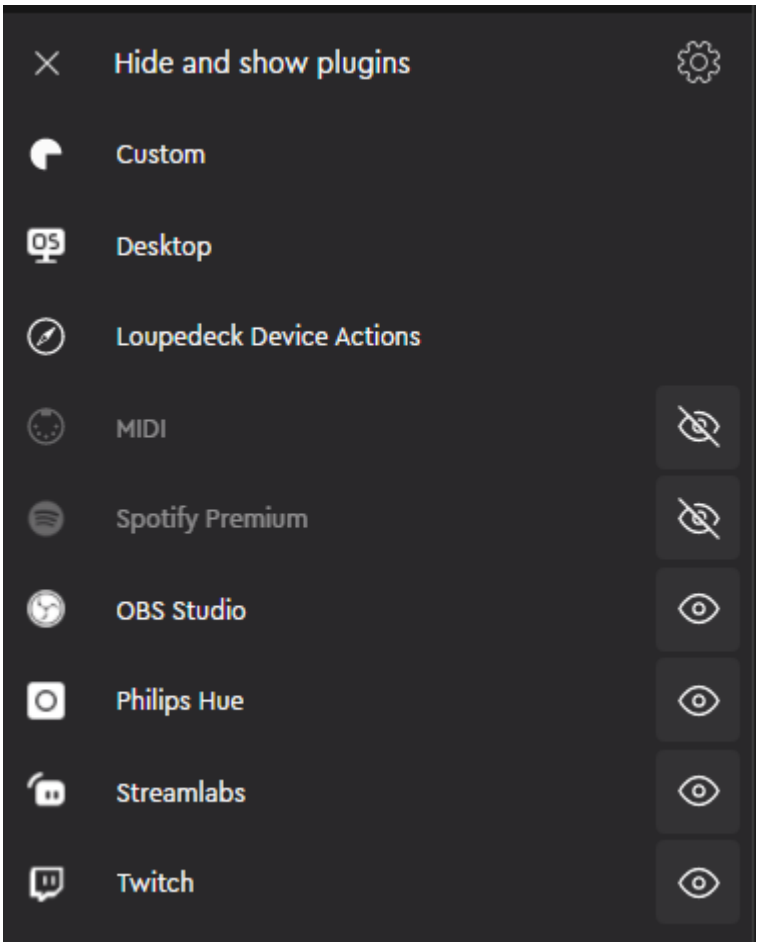
There are two distinct types of actions:

- Shortcuts, which essentially are key combinations that Logi Plugin Service sends on behalf of the user and
- API-based actions, those that are controlling target application/service using dedicated API (for example, OBS can be controlled via WebSocket using obs-websocket plugin)

to indicate if a plugin is having API-only actions, set the following flag in the Plugin class:

```
public override Boolean UsesApplicationApiOnly => true;
```

Plugins with this flag set to true can be connected to any profile and are accessible from the Action Panel.



Each plugin can be in one of the following states:

- "Normal" - plugin is working properly;
- "Warning" - plugin is partially working;
- "Error" - plugin is not working:
  - application is not installed;
  - cannot connect to application plugin;
  - cannot connect to cloud service;
  - login or authentication required;
  - etc.

By default plugin is in "Normal" state.

Plugin developer should call `OnPluginStatusChanged` method to change plugin state.

The following methods are available.

```
public void OnPluginStatusChanged(PluginStatus status, String message, String supportUrl);
```

- To set "normal" state:

```
this.OnPluginStatusChanged(PluginStatus.Normal, null, null);
```

- To set "error" state:

```
this.OnPluginStatusChanged(PluginStatus.Error,
    "Cannot connect to Twitch application",
    "https://support.loupedeck.com/knowledgebase/article12345.html");
```

- Working example:

```
protected override void RunCommand(String actionParameter)
{
    if (actionParameter.TryGetEnumValue<PluginStatus>(out var
pluginStatus))
    {
        this.Plugin.OnPluginStatusChanged(pluginStatus, $"Plugin status
changed to {pluginStatus}", "https://support.loupedeck.com/my-test-plugin-
error");
    }
}
```

In addition plugin can override `Plugin.Install()` and `Plugin.Uninstall()` methods to customize installation and uninstallation process:

- `Install()` is called immediately after plugin is installed (copied to the Logi Plugin Service directory).
- `Uninstall()` is called just before plugin is uninstalled (deleted from the Logi Plugin Service directory).

What `Install()` method can do:

- Copy photo editing application plugin file(s) to specific directory.
- Copy scripts, presets, etc. to use by photo editing application plugin file(s) to specific directory.
- Allow specific TCP/IP ports in firewall.

What `Uninstall()` method can do:

- Delete files installed by Install method.
- Delete caching files.
- Install additional files

If plugin needs to install additional files (photo editing application plugin, scripts, presets, etc.) it can do that in `Install()` method:

- Additional files are added to plugin as "Embedded Resource".
- In `Install()` method plugin can call the following helper methods to extract these file to required directory:
  - `Assembly.ExtractFile(String resourceFileName, String pathName)` - extracts embedded resource file to specified location on local drive.
  - `Assembly.FindFile(String resourceFileName)` - return full path to embedded resource by file name.
  - `Assembly.GetFilesInFolder(String resourceFolderName)` - returns array of full paths to embedded resources located in given folder.

- `Assembly.ReadTextFile(String resourceName)` - reads text from embedded resource text file.

Plugin has access to its Assembly instance via `this.Assembly` field:

```
var pluginFileName = Path.Combine(gimpDirectory, "plug-ins",
"logi_plugin.py");
this.Assembly.ExtractFile("Loupedeck.Payload.plugin.logi_plugin.py",
pluginFileName);
```

Logi Actions plugin should have a "copy" installation performed by copying plugin DLL to a plugin folder under Logi Plugin Service Plugins directory.

If your plugin requires additional class libraries, there are 2 ways to install these additional dependencies:

- Use ILMerge tool (recommended).
- Copy dependency DLLs together with plugin DLL to the plugin folder and restart the service.

Starting with version 5.0 Logi Plugin Service supplies specific package installer that does all needed work to install the plugin to LogiPluginService Plugin directory and run all needed installation methods. The same tool LoupedeckPluginPackageInstaller.exe can uninstall the previously installed plugin.

The input for Logi Plugin Service Installer is a ZIP archive with `.lplug4` extension. To build it you can use the Plugin Tool available at Developer page.

See details here: Distributing the plugin

Default application profiles should be used only with application plugins.

There are two cases where default application profiles are used:

- When a user installs the application plugin with the default profile for the first time
- When a user wants to create a new profile for the application

When Logi Plugin Service creates the application profile:

- First the service tries to use the default profile from the Logi Plugin Service application plugin
- If a default application profile is not available, Logi Plugin Service creates an empty profile

If the application plugin is updated and has a new version of the default profile:

- It's not automatically updated to the existing application profile
- The updated default profile acts as a template for new application profiles
- When the user creates a new application profile after the update, a new default application profile is used

To create `DefaultProfileXX.lp5` files, create the regular profile in the UI and use the export feature to save it as a DefaultProfile file.

- Create a new application profile in Loupedeck software
- From the profile dropdown, select the application and click the three dots next to it, then click "Add profile". Add actions from the plugin to the profile and create the layout.
- From the profile dropdown in the Loupedeck software, select the three dots next to the selected profile and select the profile you want to export. Then select the three dots next to it to select "Export profile".

Note: It is not recommended to edit the zip profile folders manually, as personal information, such as your account name, may remain visible.

Default application profiles are stored in the native plugin binaries as embedded resources with the following file names:

In most cases, we recommend using the default profile that extends to all devices, which is:

- `DefaultProfile20.lp5`

If customization is wanted per device, you can use the following names:

- `DefaultProfile20.lp5` - for Loupedeck CT
- `DefaultProfile30.lp5` - for Loupedeck Live
- `DefaultProfile40.lp5` - for Razer Stream Controller
- `DefaultProfile50.lp5` - for Loupedeck Live S
- `DefaultProfile60.lp5` - for Razer Stream Controller X
- `DefaultProfile70.lp5` - for Logitech MX Creative Keypad
- `DefaultProfile71.lp5` - for Logitech MX Creative Dialpad

To make different default profiles for Windows and Mac, use the `win` and `mac` postfixes after the profile name. Example:

- `DefaultProfile20win.lp5` - for Loupedeck CT on Windows
- `DefaultProfile20mac.lp5` - for Loupedeck CT on Mac

The default application profiles are added as an embedded resource in Visual Studio or another IDE. We recommend adding the default profiles under `DefaultProfile` folder, but it's not required.

More information about embedding a resource to a project.

The language ID is a string in the format `languagecode-countrycode`, where `languagecode` is a lowercase 2-letter language code derived from ISO 639-1 and `countrycode` is derived from ISO 3166-1 alpha-2 and usually consists of two uppercase letters.

Examples: `en-US`, `en-GB` and `fi-FI`.

More information: System.Globalization.CultureInfo.Name Property

By default, Loupedeck's current language is the same as your OS language, if supported. Otherwise it is English.

You can overwrite the current language with the `Language` property of the `LoupedeckSettings.ini` file. For example:

```
Loupedeck/Language=de-DE
```

All the current strings (in the English language) are used as string IDs and cannot be changed.

As a result, if you want to change the English language strings, you need to add an English-to-English translation.

The plugin language is defined by the language of the client application.

If it is not possible to get the language of the client application, the plugin language is the same as the Loupedeck Software language.

The plugin must set the current language as early as possible (for example, after establishing a connection with an application via the application API) using

`Plugin.Localization.SetCurrentLanguage()` method:

```
var applicationLanguageId = this._applicationApi.GetLanguage();

if (!this.Localization.SetCurrentLanguage(applicationLanguageId))
{
    this.Localization.SetCurrentLanguage(LocalizationEngine.DefaultLanguage);
}
```

Note: Enabling logging might slow down the Logi Plugin Service considerably. Remember to turn off logging when you don't need it.

Log files are located in the "Logs" subdirectory of the Logi Plugin Service data directory:

- On Windows it is `C:\Users\<user_name>\AppData\Local\Logi\LogiPluginService\Logs`
- On macOS, it is `~/Users/<USERNAME>/Library/Application Support/Logi/LogiPluginService/Logs`

Enabling traces and logs are done by creating empty files with specific names, without the file extension in the Logi Plugin Service data directory (same place where the LoupedeckSettings.ini file is located at):

- `enablelogs` - enables writing traces to log file

Note: Filename should not include a file extension. For example, enablelogs.txt will not work.

New in version 5.6.

Logi Plugin Service provides logging possibilities also for plugins. The log messages from a plugin are written both to the Logi Plugin Service log file (when enabled) and to a plugin-specific log file. The plugin logging is always enabled, even if the Service logging is disabled. The plugin log file is located in the "Logs\plugin_logs" subdirectory of the Logi Plugin Service data directory.

For a new plugin, the easiest way to take the plugin logging into use is to generate the plugin project with the Logi Plugin Tool (see Getting Started). The Plugin Tool version must be 5.6 or newer. The generated skeleton project contains the enabler code for plugin logging and an example of how to log messages from the plugin code.

The demo plugin contains an example of plugin logging: DemoPlugin.

The `PluginLog` class provides helper methods to log messages easily everywhere in the plugin code. You can find the source code here: PluginLog.cs

The following log levels are supported by the plugin logs: Verbose, Info, Warning, and Error. For each log level, the `PluginLog` class has a method for logging

- a message only (a text string), and
- a message and an exception.

For instance, the following methods can be used for logging with the Info log level:

```
public static void Info(String text) => PluginLog._pluginLogFile?.Info(text);

public static void Info(Exception ex, String text) =>
PluginLog._pluginLogFile?.Info(ex, text);
```

For an existing plugin, you can take the plugin logging into use as follows:

1. Download the PluginLog.cs file and include it in your plugin project.
2. In the `PluginLog.cs` file, change the namespace to the same one that your plugin project uses:

   ```
   namespace Loupedeck.DemoPlugin
   ```

3. Initialize the `PluginLog` class in the constructor of your plugin class (replace the plugin class name `DemoPlugin` with your plugin class):

   ```
   public DemoPlugin() => PluginLog.Init(this.Log);
   ```

After this, you can log messages in your plugin code:

```
PluginLog.Info("Counter was reset");
```

By default, plugin actions have one state.

The plugin can define more than one state for any dynamic action.

States are identified by a 0-based index.

Once set, the number of states cannot be changed.

Each state has its own:

- display name;
- description;
- button image;
- LED color.

Create a class inherited from `PluginMultistateDynamicCommand` abstract class.

```
protected Int32 AddState(String displayName, String description)
```

In the dynamic action class constructor, the plugin calls `AddState()` method for every action state.

E.g. if the command has "on" and "off" states:

```
public LampSwitchDynamicCommand()
{
    this.AddState("On", "Lamp is turned on");
    this.AddState("Off", "Lamp is turned off");
}
```

```
public IReadOnlyList<PluginMultistateDynamicCommandState> States { get; }
```

Is `null` by default and is created during the first call of the `AddState()` method.

All methods that work with the current state have two overloads: without and with the `actionParameter` parameter.

```
public Boolean TryGetCurrentState(out Int32 currentState);
public Boolean TryGetCurrentState(String actionParameter, out Int32
currentState);
```

Plugin calls `SetCurrentState()` method to change the current state.

```
protected void SetCurrentState(Int32 newStateIndex);
protected void SetCurrentState(String actionParameter, Int32 newStateIndex);
```

```
protected void IncrementCurrentState(String actionParameter);
protected void IncrementCurrentState();

protected void DecrementCurrentState(String actionParameter);
protected void DecrementCurrentState();
```

The plugin calls these methods to increment and decrement the current state.

Incrementing the last state activates the first state. Decrementing the first state activates the last state.

```
protected void ToggleCurrentState(String actionParameter);
protected void ToggleCurrentState();
```

Plugin calls `ToggleCurrentState()` method to toggle the current state.

Works only actions with 2 states. In other cases throws `InvalidOperationException` exception.

E.g. if the command has "on" and "off" states:

```
protected override void RunCommand(String actionParameter) =>
this.ToggleCurrentState(actionParameter);
```

```
protected virtual String GetCommandDisplayName(String actionParameter, Int32
deviceState, PluginImageSize imageSize);
```

Overload `GetCommandDisplayName` method with an additional `deviceState` parameter is called for actions with more than one state.

```
protected virtual BitmapImage GetCommandImage(String actionParameter, Int32
deviceState, PluginImageSize imageSize);
```

Overload `GetCommandImage` method with an additional `deviceState` parameter is called for actions with more than one state.

```
namespace Loupedeck.Test4Plugin
{
    using System;

    public class ToggleMultistateDynamicCommand :
PluginMultistateDynamicCommand
    {
        public ToggleMultistateDynamicCommand()
            : base("Toggle Multistate", null, "Test")
        {
            this.AddState("On", "Turn me on");
            this.AddState("Off", "Turn me off");
        }

        protected override void RunCommand(String actionParameter) =>
this.ToggleCurrentState();
    }
}
```