ENFOCUS

SWITCH
25.11

Node.js Scripting

11 - 2025

Contents

1. About scripting in Switch

Switch offers a comprehensive scripting environment. With limited scripting experience, a user can substantially extend Switch's capabilities and provide for interaction with third-party systems in new and powerful ways.

Scripting applications

SwitchScriptTool is a command line tool that is part of the Switch installation. It offers the functionality to create Switch scripts and also to convert script representations from script folder to script package and back.

SwitchScripter is a Switch application component that offers a scripting development environment to allow you develop your own scripts.

To able to use the SwitchScripter, you need an active license for the Scripting Module.

Note: An active license will give you complete access to the scripting API. Therefore, information in the internal (XMP) and external datasets is accessible through the scripts, even though the Switch Metadata module is not licensed. You can also define script expressions for certain properties of some of the flow elements (for example: the Job priority property for the Hold job flow element).

SwitchScripter is installed as a separate application. It offers the following functionality:

- Edit the Switch script declaration
- Emulate the Switch scripting API for testing purposes (only for legacy scripting languages)
- Set up specific input/output conditions for testing purposes (only for legacy scripting languages)

Scripting language

As of Switch 2020 Spring, **Node.js JavaScript** is the preferred scripting language for Switch.

Some of the advantages of Node.js JavaScript are:

- It's a modern JavaScript standard supported by Node.js
- It gives script writers the ability to use the full power of the Node.js ecosystem, for example, third-party node modules.
- It offers easy in-flow debugging.

As of Switch 2020 Fall, it is also possible to use **TypeScript** to create scripts based on Node.js. TypeScript extends JavaScript by adding the concept of static type declarations, which makes it possible to find common bugs in the script before running the code. This leads to less bugs in production and can save script writers a lot of time by not having to investigate and debug these problems at run time. We also offer a TypeScript Declarations file for script writers *to enable auto-completion for the classes and methods described in this Scripting reference*.

Legacy JavaScript and VBScript are currently still supported but not documented in this new scripting guide (however, the legacy documentation will remain available through our website). In this document we focus on the use of Node.js.

Note:

- If you're new to Node.js, take a look at the chapter on *Node.js scripting* for more information about this scripting language.

- If you're switching from the legacy scripting API to the new Node.js scripting API, take a look at *Differences between the legacy and the Node.js based API* on page 64 for a better understanding.

Compatibility

Scripts created in a particular Switch/Node.js version can be used in a Switch of the same version or higher. Script writers who want their script to run in multiple versions of Switch, should use the SwitchScripter and SwitchScriptTool of the oldest Switch version that they want to support.

Switch version used to create Node.js scripts	Supported Node.js version	Node.js scripts created in this Switch version can be used in...
Switch 2019 and lower	No support for Node.js	No support for Node.js
Switch 2020 Spring	12	Switch 2020 Spring & Fall Switch 2021 Spring & Fall Switch 2022 Spring & Fall
Switch 2020 Fall	12	Switch 2020 Fall Switch 2021 Spring & Fall Switch 2022 Spring & Fall
Switch 2021 Spring	12 and 14	Switch 2021 Spring & Fall Switch 2022 Spring & Fall
Switch 2021 Fall	12, 14 and 16	Switch 2021 Fall Switch 2022 Spring & Fall
Switch 2022 Spring	12, 14 and 16	Switch 2022 Spring & Fall
Switch 2022 Fall	12, 14 and 16	Switch 2022 Fall
Switch 2023 Fall	12, 14, 16 and 18	Switch 2023 Fall
Switch 2024 Spring	12, 14, 16, 18	Switch 2024 Spring
Switch 2024 Fall	12, 14, 16, 18 and 20	Switch 2024 Fall

Remark for Mac users

For *scripts and Apps created in versions older than Switch 2021 Fall*, Switch will use older versions of Node.js which are already Intel only, so that existing scripts and Apps will remain working on more recent Switch versions running on ARM. However, we recommend script writers who create scripts that contain binaries, to foresee new versions of their scripts to not keep depending on Rosetta as Rosetta will get deprecated over time and is also slower.

For *scripts and Apps created in Switch 2021 Fall or higher*, Switch will start the Intel version of Node.js on Intel and the ARM version of Node.js on ARM and will be using Node.js 20. Therefore script writers should ship binaries for Intel and ARM and do the necessary testing for their scripts to support both platforms.

2. Scripting concepts

2.1. Node.js scripting

2.1.1. Node.js

Node.js is an open-source JavaScript runtime environment that allows executing JavaScript code on any platform. This is basically the executable included in the Switch installation.

Node.js JavaScript (also referred to as 'Node.js scripting') is the JavaScript language syntax and the API (so the functions and objects) supported by the Node.js runtime environment.

Check the following links for detailed information about Node.js and JavaScript:

- The Node.js project website where you can find all the information about the runtime environment and the supported API: *https://nodejs.org*
- Direct link to the Node.js API: *https://nodejs.org/download/release/v20.12.2/docs/api/*
- This is a link to a good JavaScript tutorial: *https://javascript.info/*

In the Switch Scripting API the functions with callbacks are not supported, as they are difficult to use and the code is difficult to maintain. Instead, promises and the async/await syntax sugar on top of them can be used. More info on async/await can be found here: *https://javascript.info/async-await*.

2.1.2. TypeScript

TypeScript extends JavaScript by adding the concept of types, which makes it possible for the TypeScript compiler to find common bugs in the script before running the code. See *https://www.typescriptlang.org/* for more info.

2.1.3. Visual Studio Code

Node.js scripts cannot be edited directly in the SwitchScripter development environment. Therefore it is recommended to use Visual Studio Code to edit Node.js scripts to take full advantage of the many features and extensions it supports.

For more information about Visual Studio Code, refer to: *https://code.visualstudio.com/docs/setup/setup-overview*.

However, you can use any code editor you want; there is no limitation.

2.2. Script element and script expression

A **script element** is a flow element that encapsulates a script written in any of the supported scripting languages. The script implements one or more "entry point" functions called by Switch automatically in appropriate moments. For example, the jobArrived entry point is called for each job being processed through the flow element.

The script can be represented by a *script folder* or a *script package*.

A **script expression** is a text string associated with a flow element property. To determine the value of the property, Switch evaluates the contents of the string as a JavaScript expression in the context of the job being processed. Legacy JavaScript and Node.js TypeScript are supported in script expressions.

The **Switch scripting API** (application programming interface) provides script elements with access to information about the job being processed, as well as its metadata. Script expressions are limited to read-only access, while script elements can update the job and its metadata information.

The scripting API objects and functions that are not available for use in Node.js script expressions are marked with this label: NOT FOR SCRIPT EXPRESSIONS. Note that using third-party node modules is also not supported in Node.js script expressions.

2.3. Script folder

A script folder is one of the two ways to represent a Switch script. It's a folder containing a number of unencrypted files that describe different parts of a Switch script as explained below.

Script folders are better suited for inclusion in version control systems for professional script development. Also, they are more convenient for developing and debugging the script code as there is no need to re-save the complete script in SwitchScripter after doing changes in script code only.

When scripts are stored in the form of script folders in version control system, it's also a lot easier to share common library code between them. It's possible to set up a fully automated build process to convert script folders into regular script packages, so the packages get a latest copy of library code during the build.

A placeholder ".sscript" file in the script folder is used to select the script folder in SwitchScripter and Switch Designer. Note that this placeholder file is empty and not a real *script package*.

Script folders are meant only for local development and therefore aren't copied into exported flows. Keep this in mind when exporting flows for use on another machine or when setting the preference for automatic flow backup that is also based on flow export. Regular script packages are therefore better suited for distribution and deployment while script folders are more convenient for script development.

Creating script folders and converting script folders into regular script packages can only be done using the *SwitchScriptTool* command line tool that is located in the Switch installation folder.

Once the script folder is created, it can be opened in SwitchScripter to edit and save changes to the script declaration.

Script folders provide similar functionality to *script packages*, however there are some differences:

- Script folders are available **for Node.js scripts only**.
- Since script folders are saved unencrypted, password protection won't work.

  Important: SwitchScriptTool will discard the password when converting a script package to a script folder.
- For script folders containing TypeScript, code transpilation has to be done using *SwitchScriptTool*.
- Script folders can only have 'Script' as type (not 'App').

A script folder cannot be used as a *scripted plug-in* without converting it to a script package. Thus to be able to create apps, you should first pack the script using SwitchScriptTool, and then set the expected script type in SwitchScripter before creating the enfpack.

2.3.1. Script folder structure

A script folder contains the complete information needed to load and execute a particular script by a script element in Switch. This includes the script program but also other information like the script ID, the list of properties and the icon for the script.

A script folder contains the following files:

- **manifest.xml**: For Switch internal use only.
- **<ScriptID>.xml**: *Script declaration* created by SwitchScripter. This file should not be edited manually.
- **<IconFileName>.png**: Optional script icon.
- **<ScriptID>.pdesc**: Optional enfpack description file created by SwitchScripter. It is present only if extra files were specified. This file should not be edited manually.
- **<ScriptID>.sscript**: Zero-size placeholder script file that can be used in SwitchScripter and Switch Designer to select the script.
- **main.ts**: Main script code file for TypeScript scripts.
- **main.js**: Either the main script code file for non-TypeScript scripts or the result of transpilation from TypeScript; executed by Switch.
- **main.js.map**: Optional source map file generated from the original *main.ts* file during transpiling. It is only present if TypeScript is used.
- **node_modules**: Optional node modules folder.
- **Resources** folder: Folder where extra resource files that are used in the script can be placed. This folder gets packed along with its content when the script is packed using SwitchScriptTool.
- **A set of <LanguageCode>.ts files**: Optional translation files (for more details, refer to the Switch App SDK).
- **.vscode/launch.json**: Configuration file for debugging a Node.js script (see *debugging script code*). This file is created when using *SwitchScriptTool* in *create mode*.
- **.vscode/switch.code-snippets**: Snippets file for all the currently available entry points. This file is created when using *SwitchScriptTool* in *create mode* (only for TypeScript).

- **package.json**: NPM configuration file for installing type declarations for Node.js and Switch Scripting (see *auto-completion using index.d.ts*). This file is created when using *SwitchScriptTool* in *create mode* (only for TypeScript).
- **tsconfig.json**: This file specifies the transpilation options required to transpile TypeScript to JavaScript (see *SwitchScriptTool: Transpile mode*). It is created when using *SwitchScriptTool* in *create mode* (only for TypeScript).
- **node_modules\@types\node**: Type declarations for Node.js (see *auto-completion using index.d.ts*). This folder is created when using *SwitchScriptTool* in *create mode* (only for TypeScript).
- **node_modules\@types\switch-scripting**: Type declarations for Switch scripting (see *auto-completion using index.d.ts*). This folder is created when using *SwitchScriptTool* in *create mode* (only for TypeScript).

Note:

- If TypeScript *main.ts* is used, in order to be able to use the script folder in a flow, the script writer must first transpile the script using *SwitchScriptTool*. This will generate *main.js*, which is the only script code that can be executed by Switch.
- The XML files inside script folders should not be edited manually. You should always use SwitchScripter!
- When packing a script folder in SwitchScriptTool, its content is adjusted to match the layout described above. Any extra files/folders are removed.

2.3.2. Script declaration

A script declaration is an XML file that specifies information about a script program, such as the script ID and the data type of any properties (arguments) used by the script. The contents of a script declaration can be edited with SwitchScripter.

Purpose

When a script folder or package is associated with a script element, the script declaration is used to:

- Produce the appropriate behavior in Switch Designer for the associated script element with regards to supported connections and properties.
- Provide the appropriate information to the run-time environment and the scripting API, for example, to access flow element properties and to allow moving jobs in accordance with the semantics for various outgoing connection types.

Main script properties

A script declaration contains the following information:

- The ID of the script.
- The number and type of supported incoming and outgoing connections.
- The script's execution mode (as in concurrent or serialized).
- The list of custom properties that were defined for the script element.

- The list of custom properties that were defined for the outgoing connections of the script element.

See *Main script properties* on page 22.

Property definition properties

Each property definition in the script declaration includes:

- A tag used to uniquely identify the property to the script and its human-readable name.
- The type of property editors used to enter a value for the property.
- Specifications for validating the property value.

See *Defining custom properties* on page 28 and *Property editors* on page 29.

2.3.3. Script program

A script program is a plain text file that contains the Node.js script program that will be executed by the Node.js executors. When a third-party "node_modules" folder is found next to the selected Node.js script, it will also be included in the script folder or package when saving it in SwitchScripter.

The script program also relies on the *Switch scripting API*. This means that the script normally uses the scripting objects Job, FlowElement and Switch, which are automatically made available for the script and allow performing Switch-specific processing tasks.

2.3.4. Icon

The optional script icon is a PNG image file (not interlaced) with a size of exactly 32x32 pixels in the RGB color space and with support for transparency. When a script folder or package is associated with a script element, and the folder or package contains an icon, Switch Designer uses the icon to display the script element in the canvas. Otherwise Switch Designer uses the default script element icon.

Note: The icon file extension must be 'png'.

2.4. Script package

A script package is another way to represent a Switch script. A script package is a file with extension .sscript. It's in fact a ZIP archive created by SwitchScripter or SwitchScriptTool that contains the same files as described for script folders (see *Script folder structure* on page 9).

Note: Some files that exist in a script folder are not present in a script package. This is the case for files that are not really required when executing a script in production, for example, Visual Studio Code configuration files. To list the script package content, use the *List mode*.

A script package can be used to deploy scripts on different machines or to exchange scripts between users as a single entity.

While script folders are better suited to develop and debug scripts, script packages are intended to be used in a production environment. Normally, once script development is finished, a script folder is packed to a script package using SwitchScriptTool in *pack mode*.

From Switch 2020 Spring onwards, it's possible and recommended to embed Node.js scripts in your script package. JavaScript and VBScript remain valid as well, but they are considered 'legacy'.

2.4.1. Password protection

A script package can be saved with a password. This means the script package can no longer be opened in SwitchScripter (for viewing or editing) without providing the same password. However the script package can still be executed by Switch. There is no way to block a script package from being executed.

For more information, refer to *Protecting a script*.

2.5. Scripted plug-in

A scripted plug-in is a regular script package saved with the value "App" as Type property and is placed in the appropriate folder, so that it can be located and loaded by Switch. Once successfully loaded, the scripted plug-in's icon appears in the Elements pane and it can be used just like any other built-in tool.

The scripting API offers several entry points, properties and functions to support optional features that are specific to scripted plug-ins. Since none of these features are required, a regular script package can be turned into a scripted plug-in in a matter of seconds.

Furthermore, all scripted plug-in features can be developed and tested in exactly the same way as regular scripts.

The scripted plug-in of the type 'App' can be installed and loaded in the Switch Elements pane on the same system where it has been created, but it cannot be distributed outside of the Enfocus Appstore.

2.6. SwitchScriptTool

Switch includes a command line tool to work with script folders called SwitchScriptTool, which is located inside the Switch installation folder.

On Windows the SwitchScriptTool.exe file is located in the SwitchScriptTool subfolder of the Switch installation folder. It is typically C:\Program Files\Enfocus\Enfocus Switch \SwitchScriptTool, but it could of course be installed elsewhere.

In order to avoid having to use the complete path you can add that folder to the PATH environment variable.

On macOS the tool is typically installed here: /Applications/Enfocus/Enfocus Switch/ SwitchScripter.app/Contents/MacOS/SwitchScriptTool/SwitchScriptTool

The symbolic link to the tool is automatically added by the installer into the installation folder root and also in /usr/local/bin folder so the tool becomes available in Terminal.

SwitchScriptTool supports the following modes of operation:

- Creating a script folder
- Packing the script folder into a script package
- Unpacking the script package into a script folder
- Transpiling the code in a script folder from TypeScript to plain JavaScript
- Generating translations (for script folders that represent apps)
- Listing all files/folders included in a script package

2.6.1. Create mode

In create mode, you have to define the script ID and the path to the target folder for the resulting script folder.

If the '--JavaScript' option is specified, the JavaScript language is used in the main script code file and a *main.js* file will be created. If this option is not specified, the TypeScript language will be used and a *main.ts* file will be created.

```
SwitchScriptTool --create <ScriptID> <PathToResultFolder> [--JavaScript]
```

SwitchScriptTool will create a script folder named <ScriptID> in the specified path. When folders in the path do not exist, they are created.

The following files/folders are always created regardless of the use of JavaScript or TypeScript:

- manifest.xml
- <ScriptID>.xml
- <ScriptID>.sscript
- main.ts or main.js (contains a jobArrived entry point with empty body)
- .vscode/launch.json
- an empty 'Resources' folder where resource files used in the script can be placed

When using TypeScript, some additional files are created:

- package.json
- tsconfig.json
- .vscode/switch.code-snippets
- node_modules\@types\node
- node_modules\@types\switch-scripting

For more details, refer to *Script folder* on page 8.

Examples

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --create
 JavaScript C:\Users\me\git\scripts --JavaScript
```

SwitchScriptTool will create a "JavaScript" script folder containing the *main.js* file.

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --create
 TypeScript C:\Users\me\git\scripts
```

SwitchScriptTool will create a "TypeScript" script folder containing the *main.ts* file.

2.6.2. Pack mode

In pack mode, you have to define the source script folder, the target folder for the resulting script package and an optional password:

```
SwitchScriptTool --pack <PathToScriptFolder> <PathToResultFolder> [--password
<Password>] [--verbose]
```

If the password was provided, the resulting package will be protected with it. For more information, refer to *Protecting a script*.

Example:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --pack C:
\Users\me\git\scripts\xsltproc C:\Users\me\sscripts --password 12345
```

Note: The 'package.json', '.vscode/launch.json' and '.vscode/switch.code-snippets' files, which were created in *create mode*, will not be present in the resulting script package.

Package contents

When packing a script folder using SwitchScriptTool, all files that are present in the node_modules folder are added to the package. To reduce the size of the package, you may want to remove modules that are not necessary before packing the script folder. You can do this by running the npm-prune command (*https://docs.npmjs.com/cli/v6/commands/npm-prune*).

Example:

The following command will remove all development dependencies, so they are not included in the package.

```
npm prune --production
```

To list all the files/folders that are added to the script package use the '--verbose' option.

Example:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --pack C:
\Users\me\git\scripts\xsltproc C:\Users\me\sscripts --verbose
```

2.6.3. Unpack mode

In unpack mode, you have to define the source script package file, the path to the resulting folder and, if the source file was password-protected, the password.

```
SwitchScriptTool --unpack <PathToSscript> <PathToResultFolder> [--password <Password>]
```

The tool won't be able to unpack the password-protected script package, if the provided password is wrong or missing.

SwitchScriptTool removes password protection, so the resulting script folder will have no password.

Example

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --unpack
 C:\Users\me\sscripts\xsltproc.sscript C:\Users\me\Documents\xsltproc --password 12345
```

2.6.4. Transpile mode

The transpile mode is **only** necessary **for TypeScript scripts**, as TypeScript scripts need to be transpiled into regular JavaScript before they can run in Switch. Therefore, once you finish editing a TypeScript script folder, it's necessary to transpile it using SwitchScriptTool.

Note: For script packages, transpilation is performed automatically in SwitchScripter when saving the script package.

Using SwitchScriptTool (instead of the regular TypeScript compiler) to do the transpiling of script folders, ensures that the same transpile options are used as in SwitchScripter, and ensures consistent behavior when packing the script folder later on by SwitchScriptTool for deployment.

The transpilation can be customized using the '**tsconfig.json**' file. If the file exists, SwitchScriptTool will use the options that are defined in this file, otherwise the default values will be used. For more info, refer to *https://www.typescriptlang.org/docs/handbook/tsconfig-json.html*.

Note: It is not possible to customize 'files', 'rootDir', 'outDir' and 'typeRoots', because these options are script folder bound and set by the default.

When running SwitchScriptTool in transpile mode, the source script folder path has to be provided:

```
SwitchScriptTool --transpile <PathToScriptFolder>
```

SwitchScriptTool will transpile *main.ts* (located in the script folder) into *main.js*.

Example

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --
transpile C:\Users\me\git\scripts\xsltproc
```

2.6.5. GenerateTranslations mode

The GenerateTranslations mode allows you to export strings for translation into the languages enabled in SwitchScripter.

```
SwitchScriptTool --generate-translations <ScriptFolder> <ResultFolder>
```

The strings are stored in .ts files in the <ResultFolder>. If the corresponding .ts files are already present in the <ResultFolder>, the tool will preserve the existing translations and add new untranslated strings. Note that if there are no languages enabled in the script folder, no .ts files will be saved.

Currently the only way to enable languages is via SwitchScripter. To do so, you should select **Edit** > **Localization** and add the languages of your choice.

More information about the localization of apps can be found in the Switch App SDK documentation.

Examples

Example on Windows:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --
generate-translations C:\Users\me\git\scripts\xsltproc C:\Users\me\git\scripts
\xsltproc
```

Example on Mac:

```
"/Applications/Enfocus/Enfocus Switch/SwitchScriptTool/SwitchScriptTool" --generate-
translations /Users/me/git/scripts/xsltproc /Users/me/git/scripts/xsltproc
```

2.6.6. List mode

The List mode allows you to list all the files/folders that are included in the script package.

You only need to provide the existing script package path. No need to provide the password as SwitchScriptTool doesn't decrypt any files and the user can do the same by unzipping the .sscript file using some zip tool.

Example

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool" --list
C:\Users\me\sscripts\xsltproc.sscript
```

2.7. SwitchScripter

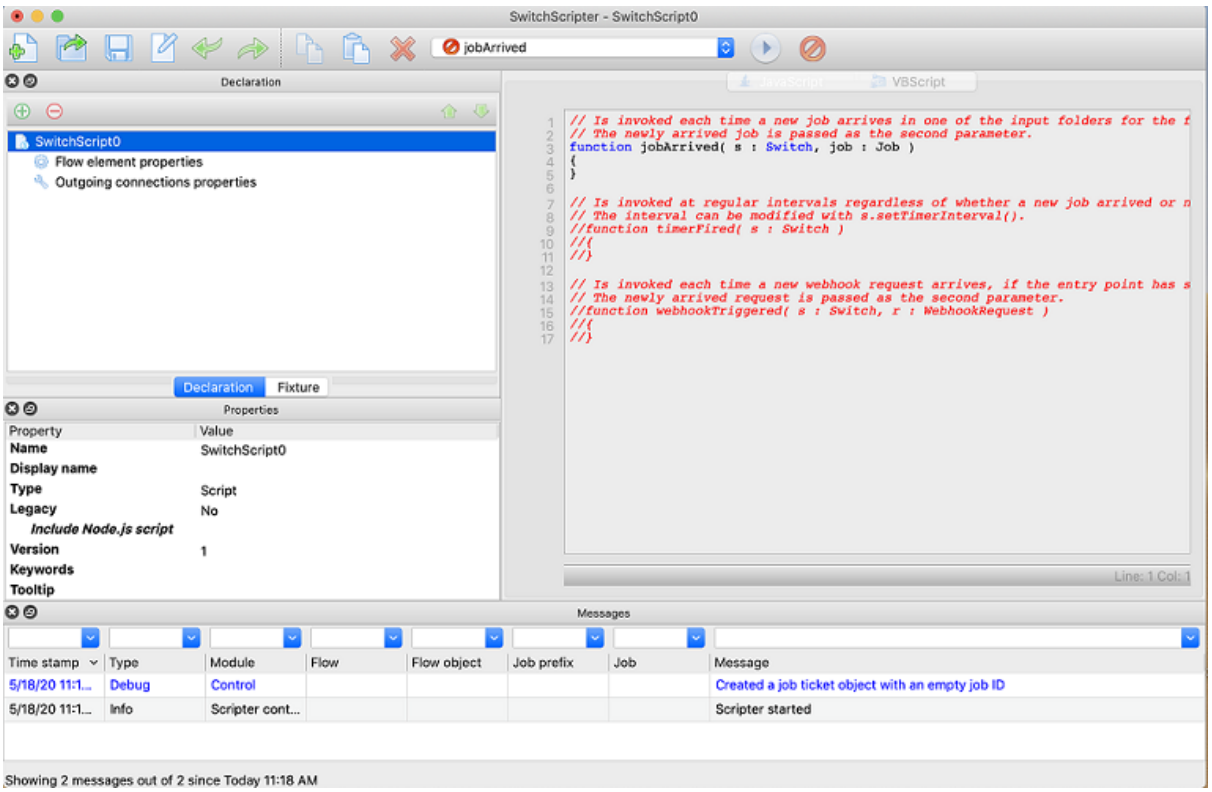
2.7.1. Launching SwitchScripter

To launch SwitchScripter:

- Double-click the SwitchScripter application icon, or
- Double-click a Switch script package (a file with the ".sscript" filename extension), or
- Double-click a placeholder .sscript file from a script folder, or
- Select a script element in the canvas in Switch Designer and choose the "Edit in Scripter" menu item in its context menu.

SwitchScripter offers a workspace window to edit a single script. SwitchScripter can display multiple instances of the workspace window at the same time.

2.7.2. Workspace window



The SwitchScripter window offers a workspace with the following important areas (each of which is discussed in a separate topic):

Workspace area	Description
Toolbar	Contains tool buttons for the most frequently used functions
Declaration pane	Allows editing the *script declaration*, defining connections, custom properties and so on
Fixture pane	Allows setting up an emulated test environment so that you can test-run a script without attaching it to a flow. *(Only for legacy scripts. Refer to the legacy Switch scripting documentation on the Enfocus website.)*
Properties pane	Serves to edit the properties of the item currently selected in the declaration or fixture pane

Workspace area	Description
Program pane	Allows editing your script program in any of the legacy scripting languages. *(Only for legacy scripts. Refer to the legacy Switch scripting documentation on the Enfocus website.)*
Message pane	Displays log messages issued by your script or by the emulated run-time environment during testing

The various panes can be shown or hidden by choosing the corresponding item in the View menu. Panes can be resized by dragging the separators between them, they can rearranged next to one another or overlaid as tabs by dragging their title bar and they can be undocked as a floating pane. All configuration settings are persistent across sessions.

2.7.3. Toolbar

The toolbar offers a number of tool buttons for frequently used operations (all of which can be accessed via menu items as well).

File

Tool	Description
New	Create a new empty script package Also see *Creating a script* on page 41.
Open	Open an existing script folder or script package
Save script	Save the script package or folder being edited

View

Tool	Description
Declaration/Fixture pane	Show the Declaration pane or the Fixture pane; toggle between these panes
Properties pane	Show or hide the Properties pane
Messages pane	Show or hide the Messages pane

Edit

The Edit buttons (Undo, Redo, Copy, Paste and Delete) are relevant for editing script programs for legacy scripts only. For more information, refer to the *legacy scripting documentation*.

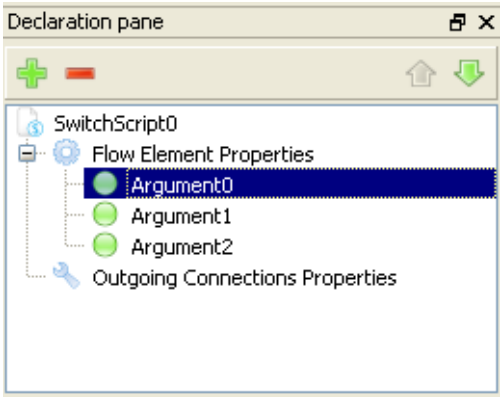
For the script package written in legacy scripting languages, SwitchScripter provides a simulated run-time environment for testing purposes.

For more information, refer to the legacy Switch scripting documentation on the *Enfocus website*.

Menu bar

Alternatively, the tools in the toolbar can be accessed through the menu bar. Note however that a number of menu options are only available for scripted plug-ins (apps), for example **File** > **Create pack**, **Edit** > **Localization** and **Edit** > **Extra files**. For more details, refer to the App SDK documentation.

2.7.4. Declaration pane



The Declaration pane allows viewing and editing the information in the script declaration for the script. When you select an item in the Declaration pane, the Properties pane displays the properties for that item.

Terminology

In this topic the term "property" is used to mean two different things:

- A "custom" property defined by the script writer that will be shown in the Properties pane in Switch Designer when the script is loaded as flow element in a Switch flow.
- One of the many "settings" in the SwitchScripter Declaration pane for the item currently selected in the SwitchScripter user interface. This may be a setting of one of the custom properties defined by the script writer for example.

This is confusing but hopefully the context of the term's use will resolve the ambiguities.

Main script properties

The properties displayed in the properties pane when you select the main script item in the Declaration pane (that is, "SwitchScript0" in the example above) are described in main script properties.

Property definitions

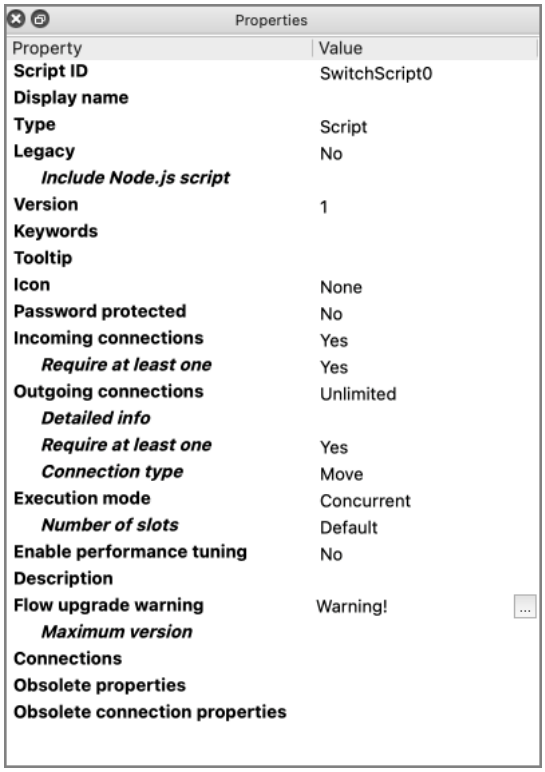
The "Flow element properties" and "Outgoing connections properties" sections of the script declaration define properties to be added to the property of flow element to which this script will be attached or to the outgoing connections of that flow element. Each section can have any number of property definitions (including zero).

Use the small tool buttons at the top of the Declaration pane to add, remove and reorder property definitions. In the above example, there are three flow element property definitions ("Argument 0", "Argument 1" and "Argument 2") and there is not yet an outgoing connection property definition.

The values of these custom properties are entered in Switch Designer's Properties pane while designing a flow, and can be retrieved by the script at run-time.

The properties displayed in the Properties pane when you select one of the property definitions in the Declaration pane (that is, "Argument 0", "Argument 1" and "Argument 2" in the above example) are described in the property definition properties.

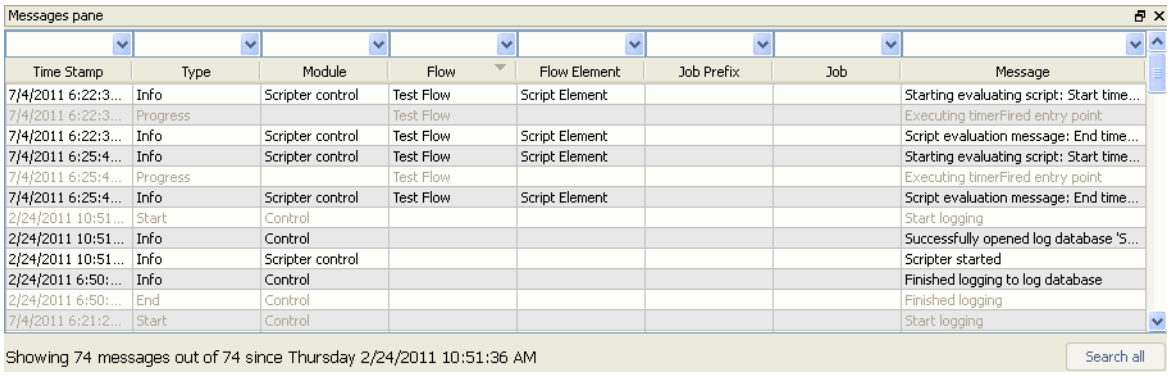
2.7.5. Properties pane



The Properties pane allows viewing and editing the properties for the currently selected item in the Declaration pane or in the Fixture pane (for legacy scripts only). The properties for each item are described in the topic on the *Declaration pane* or in the topic on the Fixture pane (details can be found in the legacy scripting documentation).

This pane is very similar to the Properties pane in Switch Designer.

2.7.6. Message pane



The Messages pane displays a list of log messages issued by your script or by the emulated run-time environment during testing. You can filter and sort the messages using the controls at the top of the pane.

3. Developing scripts

3.1. Script declaration

3.1.1. Main script properties

The following table describes the main script properties contained in the script declaration. The script declaration can be edited using the Declaration pane in SwitchScripter.

Property	Description
Script ID	The ID of this script for identification to a user For scripted plug-ins, a tilde should separate the company name from the tool name (for example, "Adobe~Acrobat Distiller"); Switch will derive appropriate long and short versions.
Display Name	Name that will be displayed to the end user. If the Display name is empty, the value of the Script ID field will be used instead.
Type	Element type. For script folders, the type is always 'Script' and this cannot be changed. For script packages, choices are: • App • Script (= regular script, for example for use with the Script element in Switch)
Legacy	If set to No, the script can only include Node.js scripts. If set to Yes, the script can include legacy JavaScript and VBScript code (for more information, refer to the legacy Switch scripting documentation on the Enfocus website). **Note:** For script folders this property must always be set No.
Include JavaScript	*This property becomes available when Legacy is set to Yes.* If set to Yes, the script package includes a JavaScript program and the corresponding tab in the program pane is enabled.
Include VBScript	*This property becomes available when Legacy is set to Yes.* If set to Yes, the script package includes a VBScript program and the corresponding tab in the program pane is enabled.

Property	Description
Include Node.js script	*This property becomes available when Legacy is set to No.* Node.js script to include in the Switch script when saving. If a 'node_modules' folder exists next to the script, it will also be included in the script. For more information, refer to *Developing script code* on page 44.
Version	The version of the script or scripted plug-in. This should be a simple integer, starting at version 1 and incremented every time the script or scripted plug-in is updated. In case of apps, it's also possible to use minor versions (like 1.1), for example for hotfix releases.
Keywords	For scripted plug-ins, defines a list of zero or more keywords, separated by a space, a comma and/or a semicolon. When filtering elements in the Elements pane with the "Search keywords" menu item turned on, this element is displayed if the filter text matches the beginning of at least one of the keywords in this list.
Tooltip	For scripted plug-ins, determines the tooltip shown for the icon in the elements pane
Icon	The icon to be displayed for a flow element associated with this script; this must be a PNG image file (not interlaced) with a size of exactly 32x32 pixels in the RGB color space (with support for transparency). **Note:** The icon file extension must be 'png'. See also *Icon* on page 11.
Password protected	**Note:** This property is only available for script packages. If set to Yes, the script package is protected by a password; the password is asked when the script package is opened for viewing and editing in SwitchScripter; a script package can always be opened for execution by Switch, even if it is password protected See also *Password protection* on page 12.
Password *Verify password*	Defines the password for the script package if it is password protected
Incoming connections	If set to Yes, the script supports incoming connections
Require at least one	If set to Yes, the script requires at least one incoming connection

Property	Description
Outgoing connections	Defines to what extent the script supports outgoing connections: "No", "One", or "Unlimited"
Detailed info	Extra information about the outgoing connections
Require at least one	If set to Yes, the script requires at least one outgoing connection (for traffic light connections: at least one outgoing connection that carries data or data with logs)
Connection type	The type of outgoing connections supported by the script, if there are any; choices are: • Move • Traffic light
Include/exclude folder	For outgoing filter connections: set to Yes to let Switch add standard "include/exclude folder" properties to the outgoing connections
Data success *Data warning* *Data error*	For outgoing traffic light connections: set to Yes to let Switch add standard properties and property values to allow sending data (output jobs) over the connection in case of success, warning or error
Log success *Log warning* *Log error*	For outgoing traffic light connections: set to Yes to let Switch add standard properties and property values to allow sending logs over the connection in case of success, warning or error
Execution mode	The execution mode for this script; one of these choices: • Serialized: entry points for all script instances in the same execution group are never invoked concurrently • Concurrent: different instances of the script may be executed concurrently
Execution group	The name of the execution group in which instances of this script reside; the precise interpretation depends on the selected execution mode: • Serialized: determines the scope of serialization • Concurrent: not used
Enable performance tuning	If set to Yes, additional performance tuning is enabled for the script. When using the script in Switch, an extra property "Idle after job (secs)" will be available, which can be used to fine-tune the flow's performance. If script execution mode is also set to "Concurrent", another property, "Number of slots", will become available.

Property	Description
Idle after job (secs)	When performance tuning is enabled, this property contains the default number of seconds the script should remain idle after processing a job. The user can override this value when designing a flow.
Number of slots	The number of slots that can be used concurrently by the specific flow element. Possible values are: "0" (unlimited), "Default" (value from Preferences is used) or any number larger than 0. Note that this property can be overridden by the value specified in Preferences > "Concurrent external processes".
Position in element pane	For scripted plug-ins, determines the position of the icon in the elements pane; the value of the property has the following format: • <section>:<sort-key> Where: • <section> is the name of the elements pane section in which the scripted package should appear: Basics, Tools, Communication, Processing, Metadata, or Custom. If it is not one of these, the section defaults to "Custom". • <sort-key> is a text string that determines the order in which elements are listed within each section; they are sorted alphabetically on the sort key.
Flow upgrade warning	Defines a warning message to be shown during a flow upgrade. This message can be added to inform the Switch user that the behavior of the new version of the plug-in has changed compared to the previous one.
Maximum version	When a flow upgrade warning message is specified, this property defines the previous plug-in maximum version up to which the flow upgrade warning should be shown in Switch. This property doesn't have any effect for flows created before Switch 2021 Fall (in this case the warning message will be shown always).
Connections	Info about the connections.
Obsolete properties	The list of property tags which were present in the previous versions of the app and were removed in the current version. For these tags, Switch does not display the warning messages during flow import or update. For information about the available relevant entry points, refer to the legacy Switch scripting documentation ("Flow element update").
Obsolete connection properties	The list of connection property tags which were present in the previous versions of the app and were removed in the current version. For these tags, Switch does not display the warning messages during flow import or update.

Property	Description
	For information about the available relevant entry points, refer to the legacy Switch scripting documentation ("Flow element update")

3.1.2. Execution mode

Switch execution is inherently multi-threaded, so in principle all tasks can run in parallel. To help reduce implementation complexities (both in the Switch framework and in a script), concurrency is limited in a number of ways as explained in this topic.

For more information on execution mode for legacy scripting languages, refer to the legacy Switch scripting documentation on the Enfocus website.

Terminology

In the following discussion it is important to differentiate between a script folder/package (that is, the definition of a script) and the script instances created by associating a script folder/package with a flow element (through a script element or a scripted plug-in). There may be several script instances for a particular script folder/package.

3.1.2.1. Execution modes

The script declaration defines the execution mode for all instances of the script as one of these choices:

- Serialized: entry points for all script instances in the same execution group are never invoked concurrently.
- Concurrent: entry points for the same or different script instances of the script may be executed concurrently.

Concurrent

In concurrent execution mode, the entry points for the same or different script instances of the script may be called concurrently. In other words, two or more script instances of the same type may run in parallel and even two or more entry points for a particular script instance may run in parallel (for example, the several concurrent executions of the jobArrived entry point).

For example, a flow containing a single flow element with a concurrent script (in addition to input/output folders) will pick up and process multiple input jobs at the same time – within the overall processing limits configured through user preferences.

Script implementations with concurrent execution mode must take care to synchronize access to resources that are shared between script instances or between different entry points.

Serialized

In serialized execution mode, entry points for script instances in the same execution group are never called concurrently. In other words, script instances in the same execution group are executed one after another. Still, the entry points in script instances outside of the execution group may be executing in parallel with those in the group.

Script implementations that need to be serialized between each other (perhaps because they use a common external resource) should refer to the same execution group. Usually however a script

implementation needs serialization only with itself. In that case, the execution group name should be unique to the script implementation.

3.1.2.2. Execution group

The script declaration defines the execution group for all instances of the script. The execution group is a string that should be structured as a reverse domain name (similar to Java packages) to avoid collision with other developer's group names; for example: "com.enfocus.myProject.myExecutionGroup".

If the value of this property is empty, the execution group name defaults to the Script ID (the first property in SwitchScripter's Declaration pane). This causes the script to be in its own execution group unless another script declares the same script ID (that is, there is no strong protection against ID collisions).

The precise interpretation of the execution group depends on the selected execution mode.

Concurrent

With concurrent execution mode, the execution group is not used.

Serialized

With serialized execution mode, a name collision between execution groups is mostly harmless in the sense that nothing breaks, but there is a potential performance issue due to the fact that the inadvertently colliding script instances can't run concurrently. Thus, it is strongly recommended that scripts intended for wide deployment provide a unique execution group name structured as described earlier.

3.1.2.3. Selecting the appropriate execution mode

The following table provides an overview of the execution modes and their typical uses.

Execution mode	Instances in same group are serialized	Information can be preserved across invocations	Example usage
Concurrent	No	No	Gather metadata of a job and produce an XML file (all done in scripting)
Serialized	Yes	No	Control a command-line application that is launched and terminated within each invocation of jobArrived

3.1.2.4. Switch executor cleanup

When an executor...

- Has been idle for 5 minutes
- Reaches 5000 processed tasks
- Consumes 150MB of memory

- Reaches 1024 open file handles

... the created files and folders are removed automatically.

Tip: To avoid files building up between executor refreshes, make sure to remove any created files and folders at the end of your script after routing them and ensure that resources (e.g. open file handles, database connections) are properly closed before the end of the script.

3.1.3. Defining custom properties

The following table describes all settings in the Declaration pane in SwitchScripter to define custom properties for the script or its outgoing connections.

Property	Description
Tag	The internal name of the property as it will be referred to from the script
Name	The name of the property as it will be displayed in Switch Designer's Properties pane
Tooltip	A brief description of the property which will be shown in Switch Designer's Properties pane as a tool tip
Inline editor	The inline property editor used to edit this property in Switch Designer's Properties pane, if any; see *inline property editors* for a list of choices and further information
Editor 1	Extra property editors shown for this property in Switch Designer's Properties pane, if any; see *other editors* for a list of choices and further information. This allows for a maximum of five property editors plus the inline editor. There must be at least one editor (if all editors are set to "None" a single-line text editor is automatically provided). Note that it is meaningless to specify the same editor twice (the second occurrence is ignored), except the external editor; it is possible to use multiple external editors for one script.
Editor 2	
Editor 3	
Editor 4	
Editor 5	
Default	The default value for the property
Depends on master	If set to yes, this property is displayed only if its master property has a particular value; the master property is the nearest preceding property with "Depends on master" set to "No" Only a single level of dependency is supported (a dependent property can never be a master)
Show if master equals	The string value or values of the master for which this property is displayed; multiple values must be separated by a semicolon
Validation	The validation method for the value of this property, which is one of: • None: perform no validation for this property

Property	Description
	• Standard: perform standard validation for this property depending on the property editor used to enter the value • Custom: invoke the script's validateProperties entry point to validate this property; do not perform standard validation • Standard and custom: first perform standard validation for this property, and if successful then also perform custom validation See also *Validating property values* on page 32.

3.1.4. Property editors

Property values

Regardless of which property editor is used to edit a custom property, the property value is always accessed from the script as a string or a string list (using the getPropertyStringValue() function). The tables below describe the values for each supported property editor.

Inline property editors

The "Inline editor" property of a property definition offers the choices listed in the following table. These property editors are contained inside the property value field in Switch Designer's properties pane.

Property editor	Resulting value
Single-line text	The text line entered in the inline editor, as a string
Password	The hidden value entered in the inline editor, as a string (this is the same as single-line text but the value is hidden)
Number	The decimal integer number entered in the inline editor, as a string (this is the same as single-line text but the user can enter only digits)
Hours and minutes	A string formatted as "hh:mm" (including the colon) where h and m stand for a decimal digit; hh and mm include a leading zero if needed (this is the same as single-line text but the user can enter only 4 digits)
No-yes list	One of the strings "No" or "Yes"
Dropdown list	The selected item (that is, one of the provided custom strings); see *Extra properties for certain property editors* on page 31.

Other property editors

The "Editor 1/2/3/4/5" properties of a property definition offers the choices listed in the following table. Most of these property editors are modal dialogs.

Property editor	Resulting value
Literal	The fixed string value corresponding to the name of the property editor, which must be selected from a predefined list; see extra properties below
Choose file	The selected absolute file path, as a string
Choose folder	The selected absolute folder path, as a string
Regular expression	The regular expression entered in the dialog, as a string
File type	The file type selected from a dialog with standard file types, as a string with the corresponding Windows filename pattern(s)
Select from library	The string value selected from a list in a "library" dialog; the contents of the dialog is determined by calling the getLibraryForProperty entry point in the script
Multi-line text	The text entered in the dialog as a single string that may include newlines
Script expression	The result of the script expression evaluated in the context of the job, converted to a string under the JavaScript rules (for more information, refer to the Switch scripting documentation on the Enfocus website).
Single-line text with variables	The text line entered in the dialog, as a string, after replacing any variables by their values in the context of the job
Multi-line text with variables	The text entered in the dialog box as a single string that may include newlines after replacing any variables by their values in the context of the job
Condition with variables	The result of the conditional expression after replacing any variables by their values in the context of the job, as one of the strings "true" or "false"
File patterns	The list of pattern strings as they have been entered in the dialog box
Folder patterns	The list of pattern strings as they have been entered in the dialog box
File types	The list of file types selected from a dialog with standard file types, each as a string with the corresponding Windows filename pattern(s)
String list	The string values entered in a generic string list dialog; each line is represented is a separate string
Select many from library	The string values selected from a list in a "library" dialog; each selected item is represented as a separate string; the contents of the dialog is determined by calling the getLibraryForProperty entry point in the script

Property editor	Resulting value
External editor	The file path to a property set edited by a third-party application; see *External property editor* on page 36.
OAuth 2.0 authorization	A string containing the OAuth 2.0 access token or an empty string if the user hasn't authorized the script yet. Refer to *OAuth 2.0 authorization editor*.

3.1.4.1. Extra properties for certain property editors

The following extra properties are shown just after the "Editor" properties only when a particular property editor is chosen.

Dropdown list

Property	Description
Dropdown items	The list of choices to be offered in the dropdown list

Choose file

Property	Description
Include file for export	If set to yes, a copy of the file indicated by this property's value (a file path) is included in the exported flow archive when a flow with this script is exported

Literal

Property	Description
Literal editor name	The name of the literal property editor, that is, one of: Default, None, Automatic, No jobs, All jobs, All Other jobs, No Folders, All Folders

A literal editor can be used to enable a constant value with a specific meaning for a property. For example, if a property requires some file path, it also may allow specifying the 'Default' value by enabling the corresponding literal editor.

When a Switch user specifies 'Default' in this property, it means some default file will be used by the script. Another common use case is the 'None' literal editor, which is the expected way to define a property that may contain an empty value. The 'Standard' validation accepts this value and there is no need to completely disable validation to allow an empty value for the property.

The table below gives an overview of the values returned by the getPropertyStringValue() for the corresponding literal editors:

Property editor	Returns:
Default	"Default"

Property editor	Returns:
None	"" (empty string)
Automatic	"Automatic"
No jobs	"No Files"
All jobs	"All Files"
All other jobs	"All Other Files"
No folders	"No Folders"
All folders	"All Folders"

Before checking the value of the property in the script, it is necessary to check that the editor type is a literal using getPropertyType().

Select from library, Select many from library, String list

Property	Description
Message	A custom message displayed on the property editor dialog

External editor

See *Property editors* on page 29.

OAuth 2.0 authorization

See *OAuth 2.0 authorization editor* on page 38.

3.1.5. Validating property values

Introduction

The "Validation" property of a property definition specifies the mechanism for validating the values of this property.

By default Switch offers a validation mechanism called 'standard validation', which validates properties at design-time or at run-time depending on the type of property editor being used.

For cases where custom validation logic is required, script writers can opt for 'custom validation' and implement their own custom validation logic in the validateProperties and validateConnectionProperties *entrypoints*.

Validation at design time

In most cases, properties are validated while the flow is being designed or just before it is activated. If there are any invalid properties, Switch refuses to activate the flow. For performance reasons, Switch does assume that the result of design-time validation remains valid for the duration of flow activation. While this assumption is correct in most production environments, it is not strictly true for all data types; for example a file path becomes invalid when the target file is removed.

The following table describes the standard design-time validation for values entered by a particular property editor. Note that all property values are of data type string or string list.

Property editor	Design-time validation requires that the value...
Single-line text	Is non-empty
Password	Is non-empty
Number	Is non-empty and contains only decimal digits
Hours and minutes	Is non-empty and has the format "hh:mm" (including the colon) where h and m stand for a decimal digit, hh and mm include leading zero if needed, hh is in range 00..23 and mm is in range 00..59
No-yes list	Is one of the strings "No" or "Yes" (exact spelling and case)
Dropdown list	Is one of the custom strings provided for the dropdown list
Literal	Is the fixed string value selected from a predefined list
Choose file	Represents a valid absolute path and that a file exists at the path
Choose folder	Represents a valid absolute path and that a folder exists at the path
Regular expression	Is non-empty and has valid regular expression syntax
File type	Is non-empty and has valid filename pattern syntax
Select from library	Is one of the strings returned by the getLibraryForProperty entry point in the script
Multi-line text	Is non-empty
Script expression	N/A
Single-line text with variables	N/A (if the value contains no variables: Is non-empty)
Multi-line text with variables	N/A (if the value contains no variables: Is non-empty)
Condition with variables	N/A
File patterns	Has at least one item and each item has valid filename pattern syntax
Folder patterns	Has at least one item and each item has valid filename pattern syntax
File types	Has at least one item and each item has valid filename pattern syntax
String list	Has at least one item and each item is non-empty

Property editor	Design-time validation requires that the value...
Select many from library	Has at least one item and each item is one of the strings returned by the getLibraryForProperty entry point in the script
External editor	Represents a valid absolute path and that a file exists at the path
OAuth 2.0 authorization	Contains an OAuth 2.0 access token

Validation at run-time

Properties that contain a variable or a script expression are validated while the flow is running, since their value cannot be determined at design time. Just before each invocation of the jobArrived entry point, Switch performs run-time validation for all properties of the flow element that contain a variable or a script expression. If a property value is invalid, Switch fails the job with an appropriate error message.

Note that there is NO run-time validation before calling the timerFired entry point (or any of the other entry points). Using non-static property values from those entry points is risky anyway since there is no job context (and referencing job context from a script expression or variable in those circumstances has undefined results).

In this case the property editor used to enter the source value (example: the script expression or text with variables) is no indication for the appropriate validation scheme. Therefore, as a general principle, standard run-time validation allows the computed property value to conform to the validation scheme of ANY of the property's property editors.

Specifically, the validation algorithm works as follows:

- Compile a list of validation schemes by adding the validation scheme from the following table for each of the property's property editors ("--" means do not add a scheme for this editor).
- The computed property value must be non-empty AND it must comply with at least one of the validation schemes in the list compiled above (unless the list is empty).

Property editor	Run-time validation scheme
Single-line text	--
Password	--
Number	Same as design-time scheme
Hours and minutes	Same as design-time scheme
No-yes list	Same as design-time scheme, or a Boolean value (*)
Dropdown list	Same as design-time scheme
Literal	Same as design-time scheme
Choose file	Same as design-time scheme
Choose folder	Same as design-time scheme

Property editor	Run-time validation scheme
Regular expression	Same as design-time scheme, or a Boolean value (*)
File type	Same as design-time scheme
Select from library	Same as design-time scheme
Multi-line text	--
Script expression	--
Single-line text with variables	--
Multi-line text with variables	--
Condition with variables	--
File patterns	Same as design-time scheme (for one item), or a Boolean value (*)
Folder patterns	Same as design-time scheme (for one item), or a Boolean value (*)
File types	Same as design-time scheme (for one item), or a Boolean value (*)
String list	--
Select many from library	Same as design-time scheme (for one item)
External editor	Same as design-time scheme
OAuth 2.0 authorization	Same as design-time scheme

(*) a Boolean value is represented by one of the strings "true" or "false"

Validation example

Consider a property that specifies a property set which can be provided as a file or it can be part of an external library. The property is set to standard validation and it has the following property editors:

- Choose file
- Select from library
- Script expression
- Single-line text with variables

For the first two editors, validation is always performed at design time. For the last two editors, validation is performed at run-time (except if the single-line text does not contain any variables). In all cases, the property value is guaranteed to contain one of the following:

- A valid file path that points to an existing file.
- One of the library items returned for the property by the getLibraryForProperty entry point.

For most use cases it can be assumed that these values are mutually exclusive, and that the script code can tell the data types apart from looking at the value. If needed however, the script code can check which editor has been used to enter the value.

3.1.6. External property editor

Introduction

Switch supports a property editor called "External editor" that offers integrated editing capabilities for third-party property sets (such as a Preflight Profile) without including third-party code in Switch. The property set must be stored in a file, and an external editing application must be supplied that behaves according to some specific guidelines.

Designing a flow with an external property editor

When a user invokes an external property editor while designing a flow, Switch launches the associated external application and passes the file path to a copy of the property set to be edited. The external application brings up a dialog for editing the contents of the property set and saves any changes back into the file. When the application exits, Switch recognizes the updated property set.

The value of a property edited with the "External editor" property editor is a file path. The actual file is stored in a temporary place allocated by Switch (similar to property sets created while importing a flow).

The "External editor" property editor works well in conjunction with the "Choose file" property editor, since both operate on a file path. Furthermore, Switch always makes a copy of the property set before editing. This means that:

- A property set that was selected with the "Choose file" property editor is never changed.
- Edited property sets are never shared between multiple flow elements.

Specifying an external property editor in SwitchScripter

The "External editor" property editor supports the following extra properties (shown in SwitchScripter and stored in the script declaration):

Property	Description
Application	The path (absolute or relative) to the application (executable) used to edit a property set. If you want to dynamically specify the path to the External editor, you have to leave the value of this property empty and implement the findExternalEditorPath entry point in your script. If a relative path is specified (that is, the path does not start with a drive letter or a forward slash), Switch looks for the external application relative to the system Applications folder. Typically this is the "Program Files" folder (on Windows) or the "Applications" folder (on Mac). If no path is specified, Switch executes the findExternalEditorPath entry point and the returned path is considered the absolute path to the required editor application.

Property	Description
Arguments for new	The argument string passed to the application when there is no pre-existing property set, after substituting all occurrences of %1 and %2 as described below.
Arguments for edit	The argument string passed to the application when there is a pre-existing property set to be edited, after substituting all occurrences of %1 and %2 as described below.
Editor name	The name your external editor user will see when he will use the editor.
Value overlay	The value of this property defines the string that will be shown after the editor is closed. The default value for this property is 'External properties defined'.

File format	There are two values possible:
	• If you want to use your own file format, choose **Custom**.
	• If you want to use a format that Switch can understand, choose **Switch**. In that case, the XML must have the following format:
	`<SwitchExternalEditor>`
	`  <PathsForExport>`
	`  <Path id="000001>c:/MyAppConfig/MyICCProfile.icc</Path>`
	`  <Path id="000002>c:/MyAppConfig/MyKeyFile</Path>`
	`  <Path id="000003>c:/MyAppConfig/MyOwnConfigurationFile.xml</Path>`
	`  <Path id="000004>c:/AnotherFile</Path>`
	`  </PathsForExport>`
	`</SwitchExternalEditor>`
	The XML includes a list of file paths. The files listed in these <Path> tags will be exported within flows. Before returning the paths list to a script, the list is sorted according to the lexical order of the corresponding **id** attribute values. This means the order of <Path> tags does not matter and may be changed by Switch after exporting the flow.

Note: If the file format is set to 'custom', Switch doesn't read the content of the property set and the files listed in the property set will not be exported within flows.

Substitutions in argument strings

The following substitutions are performed in the specified argument strings:

Placeholder	Replaced by
%1	The file path for the property set: • If it points to an existing file, the external application should load this file and replace it by the updated property set (the "edit" argument string is used)

Placeholder	Replaced by
	• If the path does not point to a file, the external application should place the newly created property set at this path (the "new" argument string is used)
%2	The locale currently in use by Switch specified as a four-letter string consisting of the two-letter ISO 639 language code (lowercase), followed by the two-letter ISO 3166 country code (uppercase); example: "enUS"

External application behavior

The external application must open the property set specified on its command line (or create a new default property set) and display a main window and/or a dialog box. After the user saves or cancels the changes, the application must exit with an appropriate exit code as follows:

Exit code	Meaning	Switch action
Zero	Changes to the property set were successfully saved	Use the newly created or updated property set
Nonzero	The user cancelled the operation or an error occurred	Discard any changes

Note: In case of the *Switch file format*, the external application must save a property set in the format described above (in the last row in the table under *Specifying an external property editor in SwitchScripter*).

Entry point "findExternalEditorPath"

findExternalEditorPath(s: Switch, flowElement: FlowElement, tag: String) : String

This entry point starts when "External Editor" is selected in the properties of a script element in Switch Designer. The third argument specifies the tag of the property involved in the operation.

This entry point returns a value that is used as the path for the external editor.

3.1.7. OAuth 2.0 authorization editor

Important remark

The OAuth 2.0 authorization editor is implemented **for Node.js scripting only**.

The recommended setup of OAuth 2.0 properties in SwitchScripter is as follows:

- Inline editor: None
- Editor 1: OAuth 2.0 authorization
- Editors 2, 3, 4, 5: None

Introduction

The OAuth 2.0 authorization editor stores an OAuth 2.0 access token and provides the user with a convenient way to request such a token from a third-party service directly from the

Switch Designer user interface. This editor is intended for scripts that call third-party REST APIs operating on data of end users. The script writers have to register their script or a group of scripts as an application in those third-party services. If they don't, the flow designer will have to do this later when using the script in a flow.

About OAuth 2.0

OAuth 2.0 is an authorization protocol that allows one side (client service) to request access to user data stored by the other side (resource service). It's commonly used for single sign-on registration on websites. In this case a website requests the user's social network profile information and creates the user account based on that information. However, it's not the only application of OAuth 2.0.

Here are some examples where OAuth 2.0 is useful in Switch context:

- Operations on e-mail boxes of popular services such as Gmail and Office 365
- Download from and upload to Dropbox
- Reporting processing results via Facebook, Twitter, Slack etc.

Authorization via OAuth 2.0 is performed as follows. The client service (in the case of Switch it's conceptually the flow element) opens a webpage of the resource service where the user authenticates and grants necessary permissions to the client service. As a result, the client service receives a temporary authorization code which is exchanged for the access token. Then the client service can query the resource service using this access token via the resource service's REST API.

The access token for some resource services can expire. Switch automatically refreshes the access token so that the users don't have to handle this in their script code.

In order for authorization to work, the resource service typically requires the client service to be registered as an application. The exact procedure to do so varies between resource services. Please refer to the resource service documentation for information and guidance.

Extra properties in SwitchScripter

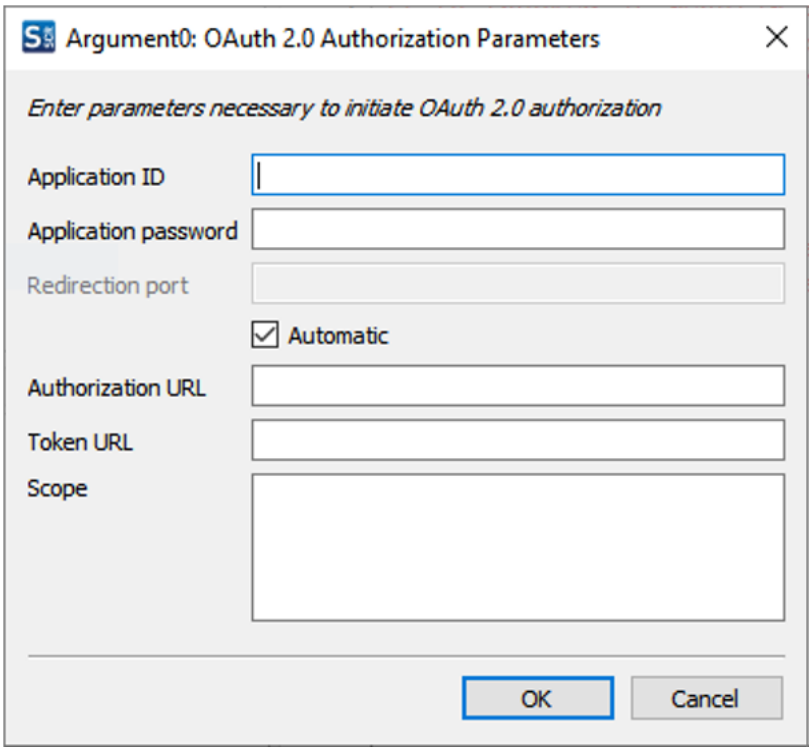
After a successful registration of the application, the resource service provides a number of values that should be entered into the properties of the OAuth 2.0 authorization editor.

Property	Description
Application ID	The identifier of the application that was registered in the resource service for this Switch script.
Application password	A password that was created during the registration of the application.  **Note:** If the resource service doesn't require a password, set Application password to *None*.
Authorization URL	URL of the resource service that is used to first authenticate the end user and then authorize the client service. The URL can usually be found in the resource service documentation.
Token URL	URL of the resource service that is used to convert the authorization code into the access token, and to refresh access tokens. The URL can usually be found in the resource service documentation.

Property	Description
Scope	The authorization scope defines which REST APIs should be made available for the client service. Note that authorization scopes are specific to the resource service. This information can usually be found in the resource service documentation. Scope names can be separated with spaces, or you can put them on separate lines via the string list editor. **Note:** If the resource service requires no scope, set Scope to *None*.
Redirection port	The OAuth 2.0 protocol requires a free network port on the client service machine, so that the resource service can return the authorization results. This port's number is sent to the resource service when the authorization is initiated. Both the resource service and the user machine may have restrictions on what network ports can be used. Therefore, this property can contain a comma-separated list of ports that Switch should try. The restrictions may be imposed by the firewall policy or the resource service itself: some resource services require a fixed redirection URI (including the port) in application settings. If there are no restrictions on the redirection port, set Redirection port to *Automatic*. In that case the operating system will select a port from the dynamic port range. If the resource service requires to define the redirection URI (one or many), specify it as http://127.0.0.1:<port> where <port> is taken from the list of ports specified in this property.

OAuth 2.0 Authorization Parameters dialog

If an extra property is left blank, its value will be asked in the OAuth 2.0 Authorization Parameters dialog when the user invokes the editor in Switch Designer.



Compared to the extra properties setup in SwitchScripter, only a single redirection port can be chosen, not a list. If the given port is occupied by another process running on this system, the authorization process will fail, and the user will have to try another port value.

3.2. Creating and using a script in Switch

This section briefly describes how to create and use a script folder in Switch.

The section *Recommended ways to work with scripts* contains a detailed step-by-step guide explaining how to create, develop, debug and deploy scripts using a small real script as example. Also the section describes what to do if you start from an existing script.

3.2.1. Creating a script

SwitchScriptTool is intended for creating new scripts for use with Switch and converting script folders to script packages.

SwitchScripter's primary function is edit the script declaration for script folders or packages.

A Switch script consists of a script declaration, a script program and an optional icon for the script.

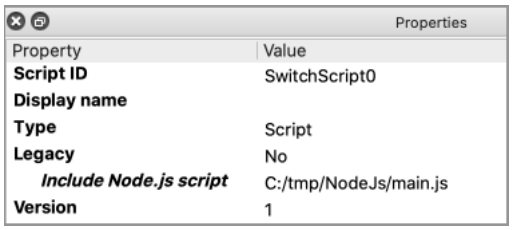
To create a new script folder for Switch from a blank sheet:

1. Run SwitchScriptTool with the 'create' option. See *Create mode*.
2. Launch SwitchScripter or, if it is already running, click **File** > **Open** and select the placeholder sscript file in the created script folder.

3. In the Declaration pane, select the first element, i.e. the root element (representing the script as a whole).

4. Include the required Node.js script file or edit in an external editor the template script file already by default included in the script folder.

   The name of the script must be *main.js* or *main.ts* depending on whether you use plain JavaScript or TypeScript.



5. Configure the other script declaration settings, including supported connections and custom properties.

6. Click **File** > **Save script** to save your script.

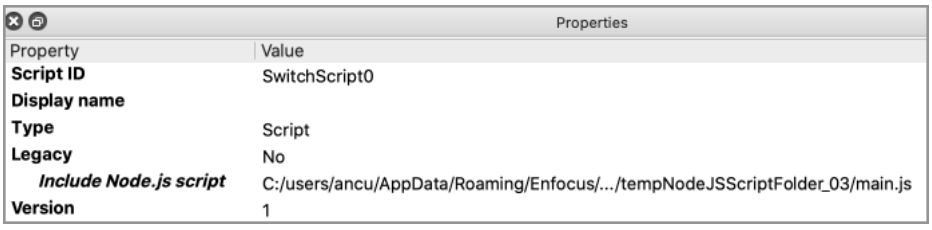
When saving a script folder or package in SwitchScripter, the specified Node.js script is copied to the script folder or embedded in the script package.

In case of a script package, if TypeScript is used (*main.ts*), SwitchScripter first transpiles TypeScript into JavaScript and puts the resulting code in the script package file. The script package still contains the original code, so that it can be accessed by the script writer when opening the script in SwitchScripter.

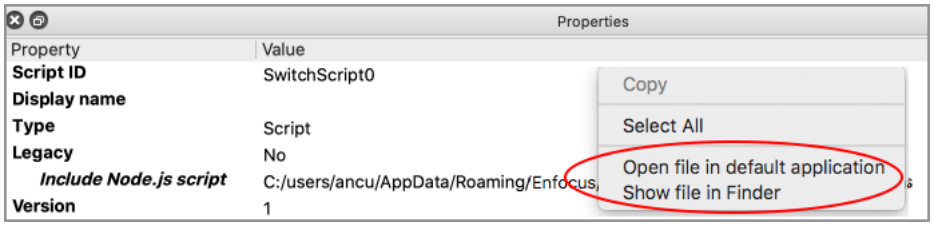
For script folders, the transpilation has to be done using SwitchScriptTool (see *SwitchScriptTool: Transpile mode*).

When reopening a script folder, the *Include Node.js script* property will point to the script file inside the folder. When a "node_modules" folder is found next to the selected Node.js script, it will also be copied to the script folder or included in the script package. This way the required node modules can be used by the Node.js executors (that are part of the Switch Processor component that is installed as part of the Switch installation) when executing the script.

When reopening a script package, the *Include Node.js script* property will point to the temporary location where the embedded version of the Node.js script was unpacked to.



You can view or edit the script file via the context menu: **Open file in default application** or **Show file in Explorer (on Windows) / Finder (on Mac)**.



Note: As of Switch 2024 Spring, both ES modules and CommonJS modules are supported in Node.js scripting. Scripts using these modules must be packed with SwitchScriptTool 2024 Spring or later.

3.2.2. Using a script in a flow

To test and use the created script folder in Switch:

1. Start Switch Designer and create a new flow.
2. Drag a script element onto the canvas in Switch.
3. Set the script element's "Script package" property to the file path of the placeholder .sscript inside the script folder.
4. Add the appropriate connections and configure any relevant properties for the script element and its connections.
5. Activate the flow that contains the script element.
6. Test the flow by adding some jobs to the input folder.

Switch retains just a reference to the script (that is, it remembers the file's path rather than the file itself). Each time the flow containing the script element is deactivated and activated, the script declaration is reloaded afresh. The script code is reloaded automatically after any modification so there is no need to deactive and active the flow. This means that the script can be modified and tried again in a very short cycle.

A script package can be used in the script element similarly.

Note: When a flow is exported, the exported ".sflow" file contains a full copy of script packages that were referenced in the flow. Script folders on the other hand will not be stored as a copy in the exported flow but only as a reference.

3.3. Developing script code

We recommend using Visual Studio Code for developing and debugging the script code.

3.3.1. Configuring Visual Studio Code

Debugging capability

Visual Studio Code enables you to debug scripts: *https://code.visualstudio.com/docs/nodejs/nodejs-debugging*.

For more information about debugging scripts in a Switch environment, refer to *Debugging script code* on page 45.

Static code analyzer

Visual Studio Code comes with a lot of extensions, which you can use to write your scripts. Here's the link to the extensions: *https://code.visualstudio.com/docs/nodejs/extensions*.

It's recommended to use ESLint, which helps you keep your code better maintainable. It also allows you to find and fix bugs earlier. For example, as a lot of Switch Scripting API calls are asynchronous, it is handy to use the ESLint feature require-await, *https://eslint.org/docs/rules/require-await*, to avoid race conditions that are not easy to catch and debug.

If you prefer to write scripts in TypeScript, you can use the @typescript/eslint plugin to lint your code. Also it's helpful to use the "@typescript-eslint/no-floating-promises" rule to detect a forgotten await keyword when calling asynchronous functions. For more info about the rule, refer to *https://github.com/typescript-eslint/typescript-eslint/blob/master/packages/eslint-plugin/docs/rules/no-floating-promises.mdx*

Here is the content of an example .eslintrc config file. Please note that Prettier code formatter (https://prettier.io/) is also used.

```
{
      "root": true,
      "parser": "@typescript-eslint/parser",
      "parserOptions": {
      "project":"./tsconfig.json"
      },
      "plugins": [
      "@typescript-eslint",
      "prettier"
      ],
      "extends": [
      "eslint:recommended",
      "plugin:@typescript-eslint/eslint-recommended",
      "plugin:@typescript-eslint/recommended",
      "prettier"
      ],
      "rules": {
      "no-console": 1, // Means warning
      "prettier/prettier": 2, // Means error
      "require-await": 2,
      "@typescript-eslint/no-floating-promises": 2 // Means error
      }
      }
      Example .prettierrc:
      {
      "semi": true,
```

```
    "trailingComma": "none",
    "singleQuote": true,
    "printWidth": 120
    }
```

Auto-completion

If the script folder was created by running the 'create' command of SwitchScriptTool, then it contains the data required for the auto-completion to work. Otherwise there are three ways to enable auto-completion:

1. Using index.d.ts

A TypeScript declarations file can be downloaded to enable auto-completion for classes and methods described in the Scripting reference.

Link to the TypeScript declarations file: *https://github.com/enfocus-switch/types-switch-scripting/blob/main/index.d.ts*.

Proceed as follows:

1. Download *index.d.ts*.
2. Put index.d.ts in the subfolder node_modules/@types/switch-scripting of the folder containing *main.ts*.

2. Using npm and package.json

Proceed as follows:

1. Create package.json.

   You can do that manually, or using *npm init -y* in the folder containing *main.ts* (if it doesn't exist already). Fix the file if needed.
2. Add "@types/switch-scripting": "https://github.com/enfocus-switch/types-switch-scripting/archive/refs/tags/v24.1.1-final.tar.gz" to dependencies in package.json.
3. Run npm install.

3. Using npm

Proceed as follows:

1. Create package.json.

   You can do that manually, or using *npm init -y* in the folder containing *main.ts* (if it doesn't exist already). Fix the file if needed.
2. Run *npm install --save https://github.com/enfocus-switch/types-switch-scripting/archive/refs/tags/v24.1.1-final.tar.gz*.

Note: Regardless of how you set up auto-completion, it's always recommended to use types in your *main.ts* file when writing your script, so that Visual Studio Code will better understand what it should offer.

3.3.2. Debugging script code

As a script writer, you can debug your Node.js scripts using Chrome or the Visual Studio Code.

Note: Debugging is only possible for script folders and not password-protected script packages!

Before running and testing your script in a Switch environment, you can try it as a pure Node.js script by mocking the calls to the Switch Scripting API. For detailed information on debugging Node.js scripts in Visual Studio Code, refer to *https://code.visualstudio.com/docs/nodejs/nodejs-debugging*.

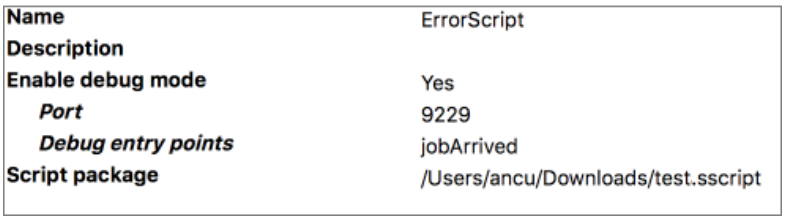
How to debug your scripts in a Switch environment

Proceed as follows:

1. In Switch Designer, create a new flow and drag a Script element to the canvas.
2. In the properties of the Script element:
   a. In the **Script package** property, specify the path to your script folder or script package. The new properties will appear:

Extra property	Meaning
Enable debug mode	If set to Yes, it is possible to debug the included Node.js script using Chrome or Visual Studio Code editor. **Note:** The script execution mode is set to 'Serialized' automatically, once the debug mode is enabled. This prevents the flow element instance that uses the script from executing entry points concurrently. The debugging can be enabled for only one Node.js script at a time.
Port	*This property is only available if Enable debug mode is set to Yes.* Defines the port number a debugger can use to connect to a Node.js process. The default port is 9229.
Debug entry points	*This property is only available if Enable debug mode is set to Yes.* Defines the list of script entry points to debug. The default value is *jobArrived*.

   b. Set the **Enable debug mode** (which appears when the script folder or package includes a Node.js script) to **Yes**.
   c. Enter the **port** number; 9229 is the default. This port will be used to connect the debugger to a Node.js process.
   d. From the list of entry points in the script, select the entry point(s) you would like to debug.



Note: Not all possible entry points can be selected for debugging. You can only select entry points from the list below that are actually used in the script:

- jobArrived

- timerFired
- httpRequestTriggeredAsync
- findExternalEditorPath

3. Build a meaningful flow to test your script and activate it.
4. Debug the script using Chrome or Visual Studio Code (see below). When execution comes to one of the selected script entry points, it will be stopped by the debugger for further investigation. As soon as the debugger stops the execution, you can access your script code automatically.

Important: Do not forget to set the debug option to No when your debug session is finished! Otherwise the flow will hang. In the case of jobArrived the input job will be stuck in the input folder of the script, but in the case of the other entry points, you would not even notice this.

Debugging in Visual Studio Code

1. Do one of the following:
   - *In case of a script folder:* Open the script folder in Visual Studio Code. The required launch.json configuration file should be a part of script folder already.
   - *In case of a script package:*
     1. Open any folder in Visual Studio Code. This can be just an empty folder.
     2. Create a new "launch.json" configuration file (**Run: Add Configuration...** or click **Ctrl+Shift+D**), for example:

```
{ // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information:  https://go.microsoft.com/fwlink/?linkid=830387
     "version": "0.2.0",
     "configurations": [
     { "type": "node",
       "request": "attach",
       "name": "Attach",
       "port": < port specified in the script element properties>
       // 9229 by default
       "skipFiles": [
            "<node_internals>/**"
                    ],
            "address": "remote host IP"
            // optional, only for remote debugging
      }
                        ]
 }
```

2. Start debugging (**Run: Start debugging** or click **F5**). You should immediately see that the debugger stopped in the internal obfuscated Switch code.
3. Click **F5** once again. Your script code will automatically be displayed in Visual Studio Code editor and you will go to the first line of your script.
4. Put a breakpoint anywhere in the entry point you are debugging. Press **F5** to go to it.
5. Here you may find info about arguments and local variables of your entry points and the script itself.

Note: Private variables are not hidden, but all of them start with an underscore. Changing or overriding these variables may lead to unpredictable behavior.

3.4. Deploying a script

Script folders are not intended to be used in production for a number of reasons:

- They don't support password-protection.
- They are not included in flows during export/backup.
- The same script code works slower when used in a script folder compared to a script package.
- Modifications to a script folder have an immediate effect on the version running in production. It is enough to save the script code with an error in it and that will make the script element enter into an error state when it processes a job.
- Script packages based on Node.js are supported in the Switch 2020 Spring release, so it's possible to develop on a newer version and deploy to an older one.

It is therefore strongly recommended to convert a script folder to a (packed) script package before using it in production. To perform the conversion, it is necessary to use SwitchScriptTool (see *SwitchScriptTool: Pack mode*).

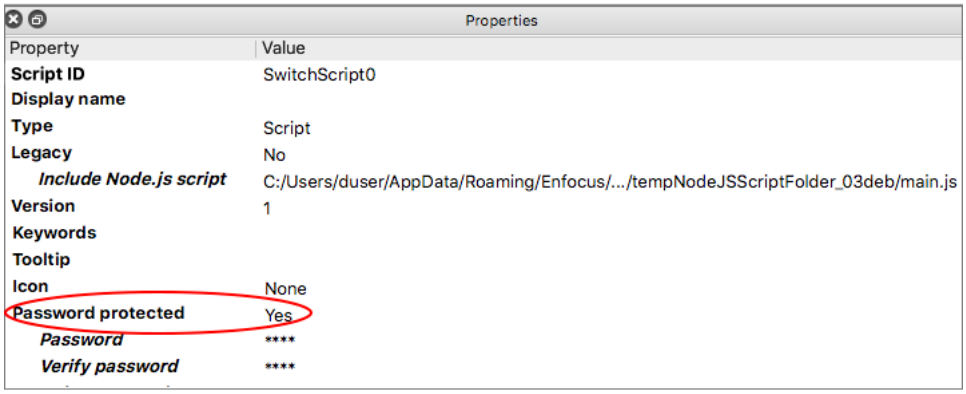
3.4.1. Protecting a script

As a script writer, you can encrypt your Node.js script code with a password.

You can do this in two ways:

- When packing a script folder to a script package, provide the '--password' command line option (see *SwitchScriptTool: Pack mode*), OR
- Open the script package in SwitchScripter, set the "Password protected" property to **Yes** and enter the password twice.

Note: Password protection is not available for *script folders*.



The same password should be used to decrypt the Node.js script file when opening the password-protected script package in SwitchScripter or when unpacking it using SwitchScriptTool. If the password is correct, the "Include Node.js script" property in SwitchScripter will point to the decrypted Node.js script in a temporary location.

Important: For Node.js scripts, even Enfocus will not be able to read the code from the password-protected script package and cannot unpack it properly if the script writer doesn't provide the password. The script writer must make sure to not forget the password, otherwise the script cannot be restored.

Obfuscation

When putting a password on a script package, or when creating a tool or app, the Node.js script code is obfuscated and the obfuscated version of the code is included in the script package in addition to the encrypted original code. The obfuscated version will be used by Switch to execute the implemented entry points.

The goal of using the obfuscated copy of the code is to avoid the need for Switch to decrypt the original code and also prevent stealing the code when the script is unpacked or when attaching a debugger to a Node.js process when the script is being executed. A"node_modules" folder, which locates next to the Node.js script *main.js* or *main.ts* is not obfuscated. This folder is intended for 3rd-party modules that don't need protection.

Note: The obfuscation is applied only to password-protected Node.js script packages.

3.5. Recommended ways to work with scripts

Starting from the Switch 2020 Fall release it is possible to develop and debug Switch scripts in the form of script folders as opposed to developing and debugging script packages. The method using script folders is certainly the recommended way (see *Script folder*).

The Switch 2020 Fall release also introduces support for working with TypeScript. As TypeScript offers better code checking and better autocompletion than working with plain JavaScript does, we also recommend this.

Working with a script package only involves the opening and saving of an .sscript file. Working with script folders, however, requires a couple of steps to be taken, and what steps they are depends on your starting point: you could be developing a brand-new script, you could want to continue the development of a script package made with Node.js, or you could be porting a legacy script package.

The following chapters will take you step by step through a simple example. The example script dismantles a job folder into individual files. It has a property to stop dismantling at a certain folder level.

3.5.1. Developing a brand-new script

If you want to start from scratch, you have to go through the following steps:

1. Create a new pre-configured script folder using *SwitchScriptTool*.
2. Open the empty placeholder .sscript file in that folder using SwitchScripter and edit it as explained further.
3. Edit *main.ts* in Visual Studio Code and transpile it with SwitchScriptTool. If your code requires any packages, install them using npm.
4. Test and debug the script folder in Switch Designer and Visual Studio Code.

5. Deploy the ready script to production by converting it to a packed script using SwitchScriptTool.
6. Debug the deployed script if necessary.

Steps 2, 3, and 4 will of course be repeated until the script does what it has to do.

Let us have a look at each step in more detail.

3.5.1.1. Creating a new script folder

It is best to keep the script folders for development, and the script packages that are used in production well separated from each other.

Let us assume a folder with the name *SwitchScripts* as your starting point and in that folder you create a folder *dev* and a folder *production*.

Proceed as follows:

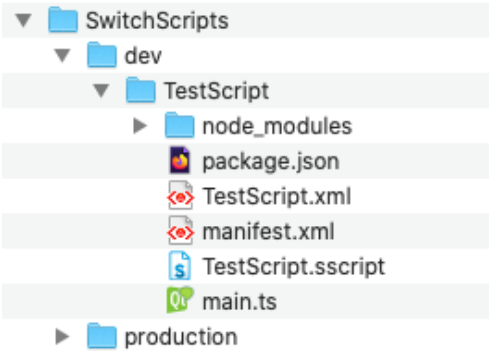
1. Make the *SwitchScripts* folder your current folder in Command prompt.

Note: When you work on macOS, it is the same principle except that you will be working in Terminal.

2. Run the following command:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool.exe" --create TestScript .\dev
```

You now have a subfolder called *TestScript* in the dev folder containing all the files as described in *Script folder structure*.



Note that on Windows, you will see an extra folder (*.vscode*) that is not visible in the above screenshot, because on Mac it's hidden.

Note: The script folder name must match the script ID, so don't change the script folder name.

3.5.1.2. Editing the .sscript file using SwitchScripter

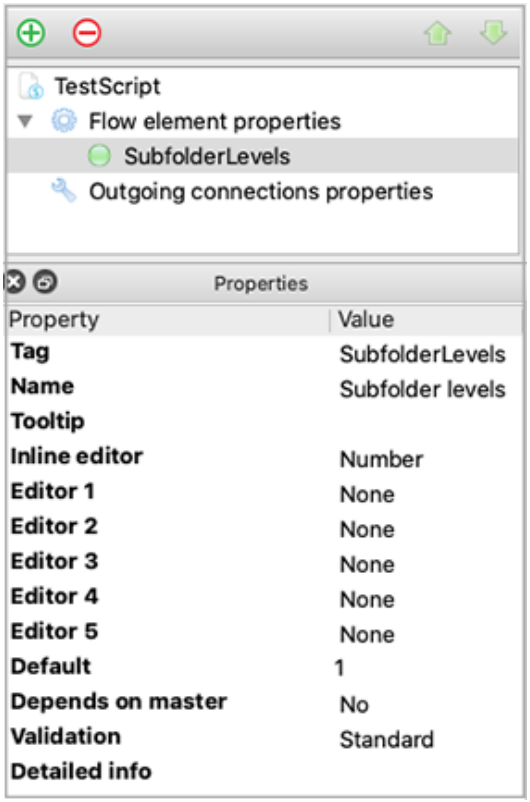
Now you have to edit the script declaration of the empty placeholder file inside the script folder.

Proceed as follows:

1. Open SwitchScripter.
2. From the menu, click **File** > **Open** and select *TestScript.sscript*.
   The script properties are displayed.

   Note that **Include Node.js script** already links to the *main.ts* file.
3. Change the **Execution mode** script declaration property to *Concurrent*.
4. Add a flow element property:
   a. Select **Flow element properties** (just below TestScript).
   b. Right-click to open the context menu and select **Add**.
   c. Select **Argument0**.
   d. Configure the properties as follows:
      - **Tag:** SubfolderLevels
      - **Name:** Subfolder levels
      - **Inline editor:** Number
      - **Default:** 1

This property will be used in the sample script and specifies the limit for the subfolder levels to scan during the dismantling process.



5. From the menu, click **File** > **Save script**.
   This will update the script declaration file, *TestScript.xml*.

Important: Every time you change the properties of a script that has already been loaded in a flow for testing, you must stop the flow that uses the script folder in Switch Designer and click the *Reload script package* context menu item for the script element. This will update the flow element and for example new properties will be shown.

3.5.1.3. Editing the main.ts file using Visual Studio Code

To edit *main.ts*, proceed as follows:

1. Open the script folder in Visual Studio Code.
2. Click on the *main.ts* file to show its content in the code editor.
   The file contains the empty implementation of the jobArrived entry point.
3. Select everything and replace it by this example code:

```
import * as fs from "fs-extra";
import * as path from "path";

async function getFiles(dir: string, levels: number, currentLevel: number, files:
 string[] = []) {
```

```
  const items = await fs.readdir(dir);
  for (const item of items) {
    const itemPath = path.join(dir, item);
    const itemStat = await fs.stat(itemPath);
    if (itemStat.isDirectory() && currentLevel < levels) {
      await getFiles(itemPath, levels, currentLevel + 1, files);
    } else {
      files.push(itemPath);
    }
  }
  return files;
}

async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
  try {
    const jobPath = await job.get(AccessLevel.ReadOnly);
    if (job.isFile()) {
      await job.log(LogLevel.Warning, "The file '%1' cannot be dismantled",
 [job.getName()]);
      await job.sendToSingle();
    } else {
      const levelsValue = await
 flowElement.getPropertyStringValue("SubfolderLevels");
      const levels = parseInt(levelsValue.toString());
      const files = await getFiles(jobPath, levels, 1);
      await Promise.all(
        files.map(async (file) => {
          const childJob = await job.createChild(file);
          await childJob.sendToSingle();
        })
      );
      await job.sendToNull();
    }
  } catch (e) {
    job.fail("Failed to process the job '%1': %2", [job.getName(), e.message]);
  }
}
```

4. Save the script.
This script recursively dismantles the incoming job folder and creates a new job for each item. To get more information about the specific Node.js or Switch scripting module object and functions, please, check the corresponding documentation.

Note:

- If your Terminal does not understand the command *npm*, you will have to install it. The best approach is to install *Node.js*; this will also install npm (Node Package Manager).

- As the above code uses the Node.js package *fs-extra* (first line of the script), you should download this package before you can use the script in Switch. To do so:

  1. Open the Terminal in Visual Studio Code
  2. Make sure the *TestScript* subfolder of the *dev* folder is the current folder.
  3. Execute the following command:

     ```
     npm install fs-extra --save
     ```

     This will download the *fs-extra* package and put it into the *node_modules* folder. Also, it will update the *package.json* file and create the *package-lock.json* file. The latter file can be removed, if you don't need it.

- As the package *path* is a standard component of Node.js, it is not necessary to install it using npm, but it still must be imported, as is being done on line 2 of the script.

5. Next, you must transpile the TypeScript code to JavaScript. To do so, open a command prompt and execute the following command:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool.exe" --transpile .
```

Note: You can customize the transpilation options in the 'tsconfig.json' file (see *SwitchScriptTool: Transpile mode*).

Note the dot referring to the local folder at the end of the command. When it is the first time you run this command, you will see that 2 files are being added to the folder: *main.js* and *main.js.map*. The *main.js* file is the one that will actually be executed by Switch. The *main.js.map* file will be used by Visual Studio Code during debugging (see *Debugging script code*).

Note:

- Every time you make a modification to *main.ts* you have to transpile it again. If you forget to do so and you run the script in a Switch flow, in the Switch Messages you will see: *The file 'main.ts' is newer than 'main.js'. After modifying TypeScript main.ts, the script writer must transpile the script using SwitchScriptTool. This will update the main.js file, which will be executed by Switch.*
- There are Visual Studio Code extensions that transpile TypeScript to JavaScript, but we recommend against using them because it is not certain that the same settings of *SwitchScriptTool* will be used for the transpilation process.

Now the script folder is ready to be used in a flow.

3.5.1.4. Testing and debugging the script in a flow

Proceed as follows:

1. **Test the script:**
   a. Place a *Script element* on the canvas of a new flow.
   b. Choose the placeholder *TestScript.sscript* file from the script folder in the *dev* folder you are working in.
   c. Add an input and an output folder element to the flow and activate the flow.
   d. Place a job folder in the input folder and see the job being dismantled into the output folder.

   Note: When testing and debugging scripts always check the logs in Switch Messages for possible errors and warnings.

2. If the script does not behave as expected, it may be necessary to **debug it:**

a. In the properties of the Script element, set Enable debug mode to Yes.

The port value defaults to 9229. That is also the value that is already defined in the *.vscode/launch.json* file of the script folder. In other words, there is nothing you need to configure in Visual Studio Code. When placing an input job in the input folder, Switch will pause and wait for the debugger to attach.

b. Go to Visual Studio Code and start the debugging process by clicking F5.
The debugger will attach to the Node.js process and the script execution will automatically break at the beginning of the internal obfuscated Switch code.

c. Press F5 again: the execution continues and stops again at the beginning of *main.ts*.

Note that despite the fact that Node.js executes the transpiled JavaScript code, the Visual Studio Code editor allows debugging the TypeScript code directly. This is made possible by the *main.js.map* file in the script folder that is created by the transpilation.

d. Now the script can be executed step by step in the debugger. Set a breakpoint at any line and press F5 – execution will continue and stop at this breakpoint.

You can now investigate the values that variables have at that point in the script to see where the problem could stem from. When there are no more breakpoints and you press F5 again, the script will execute till the end and output jobs will be moved to the output folder. For more information on debugging, see *Debugging script code*.

e. If a problem in the script code is found, the *main.ts* file needs to be fixed, saved and transpiled again. Then the debugging can be continued with the new version of the script code. The cycle “debug-fix-save-transpile” should be repeated as many times as required to ensure the script works as expected.

Note: If you change only the script code, there is no need to re-start the flow or reload the script folder in the flow to apply the changes.

3.5.1.5. Deploying the script

Proceed as follows:

1. Remembering the suggested folder structure at the beginning of this chapter, pack the script folder with the following command in Visual Studio Code's Terminal:

Windows:

```
C:\"Program Files"\Enfocus\"Enfocus Switch"\SwitchScriptTool\SwitchScriptTool.exe
 --pack . ..\..\production
```

Mac:

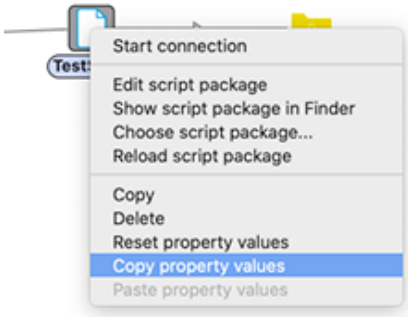
```
SwitchScriptTool --pack . ../../production
```

Note the dot after the *--pack* option to refer to the current folder, followed by a space and two dots that go one level up. You can optionally add a password by adding the *--password* option at the end. For more information, refer to *Protecting a script*.

The result is a file called *TestScript.sscript* in the *production* folder.

2. In Switch, replace the script folder by the script package in the script element:

a. Copy the property values of the script element from the context menu.



b. Choose the script package.

c. Paste the property values from the context menu.

3.5.1.6. Debugging the deployed script

It is recommended to debug Switch scripts while they are script folders, before they are packed to script packages. However debugging script packages might be required in some cases, for example, when a problem reproduces only in production.

Debugging a package is also supported, just like it was in the previous Switch version. The package must have no password.

Also keep in mind the following:

- A script package doesn't have a *launch.json* file with the debug configuration, so you might need to create it as described in the section about *debugging script code*.
- It is not possible to edit the *main.ts* file being debugged, as this is the temporary script from the cache of Switch Processor. To modify the script code you have to open the package in SwitchScripter, edit the temporary *main.ts* file referred by the *Include Node.js script* property and save the package.
- After the package is modified the new copy of the script code is loaded in the cache and executed by Switch Processor so the prevously set breakpoints are lost.

3.5.2. Starting from a script package

If you want to start from a Node.js-based script package that uses JavaScript (not TypeScript) and you want to handle it as a script folder, you should proceed as follows:

1. To unpack the script package to the *dev* folder, use the following command:

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool.exe" --
unpack .\TestScript.sscript .\dev
```

2. Choose the placeholder *TestScript.sscript* file from the resulting script folder in a Script element in the flow, change the *main.js* (or *main.ts*) file, test it, change it, test it again and so on as described higher *(steps 2-5)*.

3. In order to change or add properties of the script, open the placeholder *TestScript.sscript* in SwitchScripter, change what is required, save it and then reload the script package in the flow to activate these changes.

4. Finally pack the script folder for deployment to production.

However, there are a few things missing in the unpacked script folder that you may require. They are explained below.

3.5.2.1. Debugging

When starting from a script package, the file *launch.json* in the hidden subfolder *.vscode* that contains the debug configuration is not created. It is possible to add a debug configuration in Visual Studio Code, but the easiest approach is to simply copy that folder from another script folder that was created with SwitchScriptTool. The content of that folder in the context of Switch scripting is always the same, so it does not matter where you take it from.

3.5.2.2. Packages

When starting from a script package, the script folder does not contain the file *package.json*, a file that contains information about the Node.js modules used by the script.

To generate this file, run the command *npm init* in the Terminal of Visual Studio Code. This will create the *package.json* file and it will add the dependent modules that are in the *node_modules* subfolder to the file.

3.5.2.3. Support for TypeScript

If your script package was JavaScript and you want to change the script folder to work with TypeScript at the same time, you have to take a few additional steps:

1. Open the *TestScript.sscript* placeholder file in SwitchScripter.
2. Rename the *main.js* file to *main.ts*.
   As TypeScript is a superset of JavaScript, any JavaScript file is also valid TypeScript.
3. Replace the link to *main.js* to the link to *main.ts* in the *Include node.js* property of the script.
4. As one of the big advantages of using TypeScript is the much better autocompletion of code, you also have to install the types used by Node.js and by Switch.
   a. Open the *package.json* from another script folder created with SwitchScriptTool.
   b. Copy the below lines from that file, paste it into the *package.json* file of the current script folder, and save it.

```
"devDependencies":
    {
        "@types/node": "12.12.70",
        "@types/switch-scripting": "https://github.com/enfocus-switch/types-
switch-scripting/archive/refs/tags/v24.1.1-final.tar.gz"
    },
```

   c. Run the command *npm install* in the Terminal of Visual Studio Code.
   This will actually create or update the *node_modules\@types* folder.

d. In the code itself you should also make sure that the parameters of the entry points are typed. Change for example the below lines...

```
async function jobArrived(s, flowElement, job)
{

}
```

to:

```
async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {

}
```

e. Before you can now test the new version of the script, enter the following command to transpile the code (see *SwitchScriptTool: Transpile mode*):

```
"C:\Program Files\Enfocus\Enfocus Switch\SwitchScriptTool\SwitchScriptTool.exe"
 --transpile .
```

Note: You can customize the transpilation options in the 'tsconfig.json' file. Copy the file from another script folder created with SwitchScriptTool using TypeScript and customize it like you want (see *SwitchScriptTool: Transpile mode*).

You can now develop, debug and deploy the resulting script folder exactly like described higher *(steps 2-5)*.

3.5.3. Starting from a legacy script

In case you start from a legacy script, it's recommended to convert it to a Node.js script folder.

Note: Make sure you have a backup copy of your legacy script package.

Proceed as follows:

1. *Create a new script folder* with SwitchScriptTool.
2. Open the legacy package in SwitchScripter.
3. Set the **Legacy property** to **No**.
4. Link to the empty *main.ts* or *main.js* in the script folder.
5. From the menu, click **File** > **Save script**.
6. Convert the resulting package to a script folder as described *higher*.
   This ensures that the script folder has the same script declaration.

You can now start developing the script code and go through the same cycles of coding, testing, debugging and finally deploying the script.

3.5.4. Updating older Node.js apps to work with Switch 2024

To make Node.js apps created with older Switch versions compatible with Switch 2024, proceed as follows:

1. Update "devDependencies" -> "@types/switch-scripting" to the new version. You can get the updated link from GitHub.

For example, if an app was compatible with Switch 2022:

BEFORE:

```
json
Copy code
"devDependencies": {
  "@types/node":" 16.11.7",
  "@types/switch-scripting": "https://github.com/enfocus-switch/types-switch-
scripting/archive/22.0.0.tar.gz"
}
```

AFTER:

```
json
Copy code
"devDependencies": {
"@types/node": "20.12.2",
"@types/switch-scripting": "
https://github.com/enfocus-switch/types-switch-scripting/archive/refs/tags/v24.1.1-
final.tar.gz"
}
```

2. Run npm install in the script folder.

3.6. Automating third-party applications

This topic introduces the mechanisms used to develop a script that automates a third-party application and it offers guidelines for adjusting third-party applications so that they can be optimally configured and controlled from within Switch.

3.6.1. Third-party application settings

The third-party application may offer preferences or other settings that are persistent between invocations and cannot be controlled from Switch. This is no problem as long as the user has a way to influence these settings independently from Switch, for example through a user interface that is part of or ships with the third-party application.

3.6.2. Property sets

A property set is a collection of controls, settings or options used to configure a third-party application. A property set usually combines a large number of options and it vastly simplifies the

user interface required in Switch for configuring the application because it can be referred to as a single entity.

Referring to a property set

Depending on how a property set is stored by the third-party application, Switch keeps a different type of reference to the property set – as illustrated in the following table.

Storage model	Reference in Switch	Stored in exported flow
As a regular file anywhere in the file system	Absolute file path	Full copy of the file
In an open repository (folder) controlled by the third-party application but directly accessible to Switch as a regular file	Absolute file path	Full copy of the file
In a private repository (database) controlled by the third-party application and only accessible by Switch by name and through the third-party application	Name	Name

Some applications support multiple storage models for the same property (that is, the property can be specified as a named repository item or as a file path). Switch scripts can easily accommodate this by providing multiple property editors for the property.

Selecting a property set

Switch allows a choice between browsing for a file (the first storage model) or selecting from a list of names (the second and third storage models). Note that there is no way to show a tree structure; only a linear list of names.

The following table describes the implementation mechanism and the cooperation required from the third-party application depending on the storage model.

Storage model	Action in Switch	Implemented through	Cooperation required from application
As a regular file in the file system	Allow the user to browse the file system for a property set	"Choose file" property editor	"Choose file" property editor
In an open repository as a regular file	List all of the appropriate files in the repository folder and show them in a dialog after stripping the filename extension	"Select from library" property editor with getLibraryForProperty entry point that iterates over the repository folder	Document the location of the repository folder
In a private repository, by name	Request the list of names from the third-party application and show them in a dialog box	"Select from library" property editor with getLibraryForProperty entry point that interacts	Provide a run-time mechanism to retrieve the list of names on request

Storage model	Action in Switch	Implemented through	Cooperation required from application
		with the third-party application	

Note: Switch scripting also offers an "External editor" that provides a way to integrate the external GUI application created by a third-party using Switch Designer. The external application is automatically started by Switch Designer when the property editing is required, so the application's GUI looks as a part of Switch Designer. For more details, refer to *External property editor* on page 36.

3.6.3. Communication requirements

Scripts that automate third-party applications need a mechanism to communicate with the third-party application with the following basic requirements:

- Pass on the values of the properties to configure the behavior of an action.
- Start an action on some input file(s).
- Detect termination of an action.
- Locate the output file(s).
- Determine success/warning/error status.

Error handling

The third-party application should clearly document how Switch can discriminate between:

- Successful execution versus failure to execute an action.
- Messages that should be passed on to the Switch execution log versus messages that are irrelevant to the Switch user (or output that can safely be ignored).
- The "level" of messages passed on to the Switch execution log: info/warning/error.

Serialized communication

If required, Switch can serialize all communication with a third-party application, even if many instances of the same script are active simultaneously. For more information, see *Execution modes* on page 26.

Switch can also automatically terminate a third-party application if an action does not complete within a certain amount of time (which is configurable as a user preference).

Platform considerations

If a third-party application is available on both Mac and Windows, it is acceptable (but not necessarily preferable) to use a different communication mechanism on each platform. In other words, a Switch script can include platform-specific code, as long as the user's view of the properties and behavior is the same on both platforms.

3.6.4. Communication mechanisms

Communication with the third-party application is implemented mostly (or solely) in the script's jobArrived entry point.

Command line

From an implementation standpoint this is often the easiest form of communication. The Switch script compiles command line options based on the script properties and if necessary Switch provides console input and/or parses console output. Alternative sources of information may be the application's return code and log files written by the application.

Hotfolder

A hotfolder application could be driven from a Switch script on the condition that it is possible to configure the application's behavior using a single watched folder and a control file. There can be several setups including:

- The control file is placed in a watched folder and points to the path of the input file(s).
- The input file and the control file are both placed in the same watched folder (and they are matched through some filename pattern).

A Switch script can NOT control an application that requires a different folder watched for each configuration setup.

Preferably the third-party application and Switch should be able to settle on a watched folder path without requiring user setup in the third-party application. The best way to achieve this is to provide a mechanism that allows Switch to set the watched folder path at its discretion. Failing that, the third-party application should document a mechanism for the script to retrieve or otherwise determine the watched folder path.

Loadable library

Third-party functionality that is offered as a loadable library (DLL on Windows, dylib on Mac) can be accessed in a Switch script by providing an independent helper application. The helper application calls the library functions and provides a simple interface towards Switch (usually command line and/or console input/output).

Network communication

A script can use one of the networking protocols (e.g. REST) to communicate with the external application or service.

3.6.5. Text encoding issues

On modern operating systems file and folder names may contain any Unicode code point (except for a few platform-specific separator characters). This is true even if the default 8-bit code page is not able to represent these characters. For example, on a regular English Windows XP system it is perfectly possible to have Greek or Cyrillic characters in a filename, although these characters cannot be represented in the default Latin code page.

Switch is fully Unicode enabled and for optimal operation it requires a configured third-party application to be Unicode enabled as well. In other words Switch requires that the third-party application:

- Correctly works with filenames that may contain any Unicode code point.
- Performs all text communication with Switch using an appropriate and well-defined character encoding (as explained in more detail below).

Command line on Mac OS X

Mac OS X uses UTF-8 as its default encoding for representing filenames/paths. In our experience, a command line application can simply take a file path from the command line as an 8-bit string and pass it through to a regular UNIX or Mac OS file system call even if the command line application itself is not Unicode aware.

Command line on Windows

The command line application in turn MUST invoke the Windows-specific Unicode-enabled function calls for retrieving the command line AND for opening the files. For example, in C/C++ the command line application must use the Unicode ("wide") version of the Windows-specific "GetCommandLine" function rather than the argv[] argument in the main function (which only supports the current default code page).

Other text input/output

This includes the console input/output streams and any exchanged control files that may contain plain text (as opposed to XML which has built-in Unicode support).

It is strongly recommended to use UTF-8 for all text input/output because this encoding can represent any Unicode code point and it is upwards compatible with 7-bit ASCII in various ways (for example, with regards to line breaks and null-terminators).

If this is not feasible, at the very least the encoding used must be well-defined and documented, and it should follow these guidelines:

- Text that may contain a filename/path must be able to represent any Unicode code point. In essence UTF-8 or UTF-16 are the only options.
- Text that may contain code points outside 7-bit ASCII (for example, localized messages for human readers) must be in a well-defined encoding that is able to represent the characters in the languages used. Preferably this encoding is the same on all platforms; or at least it is well defined. Again UTF-8 is the best option.
- Text that only contains 7-bit ASCII code points (for example, keywords that are not localized and are not intended for human consumption) is not encoding-sensitive. Still it does not hurt to use UTF-8 since it is compatible with 7-bit ASCII.

4. Scripting reference

4.1. About the Switch scripting API

The Switch scripting API (application programming interface) provides script elements with access to information about the job (file or job folder) being processed, the metadata contents and the job's processing environment.

The API documentation in this document is provided for **Node.js scripting**.

Currently, in Node.js scripting we provide support for:

- Accessing script flow element properties, including outgoing connections
- Moving jobs to these connections
- Accessing information about the current job, including the list of associated metadata datasets

If you need any other functionality that was available in the legacy scripting API, please submit a feature request to Enfocus.

4.2. Differences between the legacy and the Node.js based API

This chapter explains the differences between the legacy JavaScript API and Node.js JavaScript API. It is important for script writers who have already used legacy JavaScript and want to switch to the new scripting.

The advantage of Node.js scripting is the huge set of modules (collections of JavaScript functions and objects) available through Node Package Manager. These modules provide much of the functionality that previously was available in the legacy JavaScript API and a lot more other functionality previously unavailable.

The Node.js scripting API provides the essential set of features for a script to work in a Switch context. The API will be extended in future versions of Switch.

Methods documentation

The documentation of the available methods on the Switch classes (see further) lists some methods just with their names and parameters, and some preceded by the word 'async'. The latter means that the particular method returns a Promise. It is therefore recommended to call these methods with 'await'. The other methods can be called without this.

4.2.1. Entry points

Available entry points

- jobArrived

- timerFired
- getLibraryForProperty
- getLibraryForConnectionProperty
- validateProperties (replaces isPropertyValid)
- validateConnectionProperties (replaces isConnectionPropertyValid)
- findExternalEditorPath
- httpRequestTriggeredSync
- httpRequestTriggeredAsync
- flowStartTriggered
- flowStopTriggered

Refer to *Script entry points* on page 71.

Unavailable entry points

- getUpdatedValueForProperty
- getUpdatedValueForConnectionProperty
- findApplicationPath
- checkApplicationPath
- getApplicationLicensing
- licenseApplication

4.2.2. Property values

Instead of the getPropertyEditor a script user has to use the getPropertyType functions for the FlowElement and the Connection class. An exact mapping between property type and property editor is described in *PropertyType* on page 117.

The getPropertyStringValue function combines the functionality of getPropertyValue and getPropertyValueList from the legacy scripting: when using a single-line editor the function returns a string, when using a multi-line editor the function returns an array of strings. As in the legacy scripting the property values are always returned as strings. In the future, we may provide functions that return other data types, but for the moment casting them to different data types is up to the script developer.

In the legacy scripting the getPropertyValue function returned an empty string when the property in question was a dependent property that was not visible as a result of the setting of the master property. The new scripting, however, throws an error in such a case. In other words you should always use getPropertyStringValue inside an if statement that uses the same logic as the definition of visibility of the dependent property, or place it inside a try-catch construction.

Example:

Master property (yes/no)

Dependent property 1 (visible if "Master property" is yes)

Dependent property 2 (visible if "Master property" is no)

```
if (await flowElement.getPropertyStringValue("Master_property") === "Yes") {
  dependentPropertyValue = await
 flowElement.getPropertyStringValue("Dependent_property_1");
} else {
  dependentPropertyValue = await
 flowElement.getPropertyStringValue("Dependent_property_2");
}
```

OR

```
let dependentPropertyValue;
try {
  dependentPropertyValue = await
 flowElement.getPropertyStringValue("Dependent_property_1");
} catch(err) {
  dependentPropertyValue = "";
}
```

There is a difference in the moment the value of a property is evaluated. In the legacy scripting engine the value of a property is evaluated at the moment of calling s.getPropertyValue inside the script. In the Node.js engine it is evaluated at the moment the script starts. In practically all cases that will not make a difference: evaluating a single-line text with variables that takes a value from an XML dataset will yield the same value whether performed at the start of the script or during the execution of the script. It is, however, theoretically possible that for example a variable takes its value from a database and that it does so several times during the execution of the script in a loop. In such a case it is possible that every call returns a different value in the case of a legacy script, but in a Node.js script it will always have the initial value.

In Node.js scripting, properties using Single-line text with variables, Multi-line text with variables, Condition with variables and script expressions are only available in jobArrived entry points. An attempt to get the value of such properties in timerFired or any entry point other than jobArrived will throw an exception.

4.2.3. Job file and dataset file access

A script can no longer directly modify a job or a dataset file that is located in the element's input folder. The Job::get function and the Job::getDataset function have an accessLevel argument. If read/write access level is requested, the job or dataset file is automatically copied to a temporary location where it can be modified by the script. If the job or dataset is modified during script execution, it is automatically updated in the flow (moved to an output folder) the moment the sendTo function is called.

For the case where a script needs to generate a new job, the old scripting API offered methods (Job::createPathWithName and Job::createPathWithExtension), which returned a temporary path. This path could be used by the script to save the content of the new job and afterwards it had to be passed as parameter in one of the Job sendTo methods. The sendTo method automatically converted the file or folder at that temporary location into a new job and routed it to an output folder. In the new scripting API the script first needs to create a file or folder on disk and pass this path as parameter to FlowElement::createJob, Job::createChild or Job::createDataset to explicitly create a job or dataset in Switch from the file or folder. In the new scripting API, the Job sendTo methods no longer require a path as parameter to be able to route the job afterwards.

Files or folders that are passed to FlowElement::createJob, Job::createChild, or Job::createDataset will not automatically be removed by Switch when sending or failing the job. Therefore it's up to the script to make sure that temporary files or folders passed to these methods are correctly removed after sending or failing the job.

4.2.4. Sending jobs

After a job is sent to an output connection, no other calls for this Job object are allowed except the other sendTo calls and Job::fail. It's a good practice to create a new or child job if you want to route the original job to more than one connection. A child job is a new job that inherits all the properties of the parent job (job ticket, datasets, etc.). There is a new method in the Job class to

create a child job. The creation of a child job was not possible in the legacy scripting. However, it is allowed to call sendTo methods a few times for the same incoming job, if you do not modify it in the entry point.

4.2.5. Scripting API - Mapping table

Legacy scripting module + class	Legacy call	Node.js scripting
Utility module		
		Node.js built-in features + 3rd-party node modules
XML module		
		3rd-party node modules
Network module		
		Node.js built-in features + 3rd-party node modules
Database module		
		3rd-party node modules
Metadata module		
CP2 classes	All	Not available
XMP data model	getString, getNumber, getBoolean	XmpDocument::evaluate
All except the above	All	3rd-party node modules
Flow element module		
Environ ment class	Maintaining global data	Switch::getGlobalData, Switch::setGlobalData, Switch::removeGlobalData
	log	FlowElement::log
	copy, move, isWindows, isMac, findRegisteredApplication, findApplicationOnDisk, find64BitApplicationOnDisk, sleep, compress, uncompress, extract, archive, unarchive, download, createPathWithName, createFileStatistics	Node.js built-in features + 3rd-party node modules
	getApplicationPath, getApplicationLicense, getServerName, getLanguage, getLanguageEnvironment, getLicenseSerial, isInTrialMode,	Not available

Legacy scripting module + class	Legacy call	Node.js scripting
	isLicenseFeatureEnabled, getSecondsLeft, getClientConnection	
	getSpecialFolderPath	Implemented only for plug-in resources - FlowElement::getPluginResourcesPath, FlowElement::getScriptDataPath
Switch class	getElementID, getElementUniqueID, getScriptName, getInConnections, getOutgoingName, getAvailableCodecs, getCounter, isPropertyValueStatic, isPropertyValueActual, getJobsForConnection, getTimerInterval, isDeactivating	Not available
	getElementName	FlowElement::getName
	getFlowName	Flow:Element::getFlowName
	getOutConnections	FlowElement::getOutConnections
	getPropertyValue, getPropertyValueList	FlowElement::getPropertyStringValue
	getPropertyEditor	FlowElement::getPropertyType
	getPropertyLocalizedName	FlowElement::getPropertyDisplayName
	hasProperty	FlowElement::hasProperty
	createNewJob	FlowElement::createJob, Job::createChild
	failProcess	FlowElement::failProcess
	setTimerInterval	FlowElement::setTimerInterval
	Remote processing	Not available
	getJobs	FlowElement::getJobs
	webhookSubscribe	Switch::httpRequestSubscribe
	webhookUnsubscribe	Switch::httpRequestUnsubscribe
Connection class	getName	Connection::getName
	getElementID	Connection::getId
	getElementUniqueID, getConnectionType, allowsSuccess, allowsWarning, allowsError, isOnHold, getFolderName, getByteCount	Not available
	getFileCount	Connection::getFileCount

Legacy scripting module + class	Legacy call	Node.js scripting
	getPropertyLocalizedName	Connection::getPropertyDisplayName
	hasProperty	Connection::hasProperty
	getPropertyValue, getPropertyValueList	Connection::getPropertyStringValue
	getPropertyEditor	Connection::getPropertyType
Job class	getPath	Job::get
	getUniqueNamePrefix	Job::getId
	getName, getNameProper	Job::getName
	getExtension, getMacType, getMacCreator, isType, getFileCount, getByteCount, createPathWithExtension, getEmbeddedDataset, activateFonts, deactivateFonts	Node.js built-in features + 3rd-party node modules
	createPathWithName	flowelement::createPathWithName
	isFile	Job::isFile
	isFolder	Job::isFolder
	sendToNull	Job::sendToNull
	sendToSingle	Job::sendToSingle
	sendToData	Job::sendToData
	sendToLog	Job::sendToLog
	sendTo	Job::sendTo
	getHierarchyPath	Available as private data with the tag 'EnfocusSwitch.hierarchy'
	getEmailAddresses	Available as private data with the tag 'EnfocusSwitch.emailAddresses'
	getEmailBody	Available as private data with the tag 'EnfocusSwitch.emailBody'
	getUserName	Available as private data with the tag 'EnfocusSwitch.userName'
	getUserFullName	Available as private data with the tag 'EnfocusSwitch.userFullName'
	getUserEmail	Available as private data with the tag 'EnfocusSwitch.userEmail'
	setHierarchyPath, addBottomHierarchySegment, addTopHierarchySegment	Available as private data with the tag 'EnfocusSwitch.hierarchy'

Legacy scripting module + class	Legacy call	Node.js scripting
	setEmailAddresses, addEmailAddress	Available as private data with the tag 'EnfocusSwitch.emailAddresses'
	setEmailBody, appendEmailBody	Available as private data with the tag 'EnfocusSwitch.emailBody'
	setUserName	Available as private data with the tag 'EnfocusSwitch.userName'
	setUserFullName	Available as private data with the tag 'EnfocusSwitch.userFullName'
	setUserEmail	Available as private data with the tag 'EnfocusSwitch.userEmail'
	sendToFilter, sendToFolderFilter, failAndRetry, failProcess, getArrivalStamp, refreshArrivalStamp, setAutoComplete	Not available
	getJobState	Available as private data with the tag 'EnfocusSwitch.state'
	setJobState	Available as private data with the tag 'EnfocusSwitch.state'
	getPriority	Job::getPriority
	setPriority	Job::setPriority
	fail	Job::fail
	log	Job::log
	Private data access and manipulation	Job::getPrivateData, Job::setPrivateData, Job::removePrivateData
	Managing external metadata datasets	Job::listDatasets, createDataset, Job::getDataset, Job::removeDataset
	Managing job families	Not available
	Accessing the occurrence trail	Not available
	Evaluating variables	Not available
Occurrence class		Not available
Webhook Request class		HttpRequest class
Webhook Response class		HttpResponse class

Legacy scripting module + class	Legacy call	Node.js scripting
Connection List class		Node.js built-in feature
JobList class		Node.js built-in feature
File Statistics class		PdfDocument, PdfPage, ImageDocument

4.3. Entry points

4.3.1. Script entry points

Entry points form the mechanism for passing control from Switch to a script in a structured manner. Specifically, an entry point is a function defined in a script (according to the rules specified below) and will be called by the Switch application at the appropriate times. All other functions defined in the scripting API work the other way around: they expect to be called from a script.

Switch invokes the entry points defined in a script at certain appropriate times, as explained below.

The first two arguments for each entry point are instances of the Switch and FlowElement classes. These objects provide access to the common Switch functionality and the flow element context.

Remarks

- As it is likely that the entry points will call functions that return Promises, it is also likely that they should be declared as async fuctions. In other words, the documentation specifies that you can use the entry point *jobArrived(s, flowElement, job)*, but in practice you will code it as: *async function jobArrived(s, flowElement, job)*.
- Switch expects at least one of the jobArrived and timerFired entry points to be present in each script.
- The entry points other than jobArrived and timerFired are often invoked while the flow is inactive, which means there is no valid flow run-time context and no valid job context. As a consequence, within these entry points calling some methods (e.g. createJob) of the Switch and FlowElement classes may be not allowed.

Entry points overview

jobArrived(s: *Switch*, flowElement: *FlowElement*, job: *Job*): Promise<void>

This entry point is invoked each time a new job arrives in one of the input folders for the script element. The newly arrived job is passed to the entry point as an argument.

timerFired(s: *Switch*, flowElement: *FlowElement*): Promise<void>

This entry point is invoked for the first time immediately after the flow is activated and afterwards it is invoked at regular intervals regardless of whether a new job arrived or not.

The default value for the interval is 300 seconds (5 minutes) and can be changed using *flowElement.setTimerInterval*.

flowStartTriggered(s : Switch, flowElement: FlowElement): Promise<void>

This entry point is invoked during flow startup.

Note:

- The flowStartTriggered entry point is executed before any jobArrived and timerFired.
- The flowStartTriggered entry point may be executed in parallel for different flow elements of the same type if an element is concurrent. To synchronize such executions, Switch.getGlobalData with lock=true can be used.

This entry point is not available for debugging.

flowStopTriggered(s : Switch, flowElement: FlowElement): Promise<void>

This entry point is invoked during flow stop, after all the other entry points in the flow are stopped and the time-out specified in the 'Release acquired slots after' property is finished (if there were acquired slots at the moment of flow stop).

Note: The flowStopTriggered entry point may be executed in parallel for different flow elements of the same type if an element is concurrent. To synchronize such executions, Switch.getGlobalData with lock=true can be used.

This entry point is not available for debugging.

getLibraryForProperty(s: *Switch*, flowElement: *FlowElement*, tag: string): Promise<string[]>

This entry point is invoked when the "select from library" property editor is chosen in the Switch Designer user interface for one of the properties of the script. The third argument specifies the tag of the property involved in the operation.

This entry point returns the list of strings to display in the property editor dialog.

This entry point is not available for debugging.

getLibraryForConnectionProperty(s: *Switch*, flowElement: *FlowElement*, c: *Connection*, tag: string): Promise<string[]>

This entry point is invoked the "select from library" property editor is chosen in the Switch Designer user interface for one of the properties of the script's outgoing connections. The third and fourth argument specify the connection and the tag of the property involved in the operation.

This entry point returns the list of strings to display in the property editor dialog.

This entry point is not available for debugging.

validateProperties(s: *Switch*, flowElement: *FlowElement*, tag: string[]): Promise<{ tag: string, valid: boolean }[]>

This entry point allows script writers to define their own custom validation logic for regular script properties (see *Validating property values* on page 32). It is invoked at appropriate times to validate the value of a property for the script with "custom" validation, for example when the user changes the property value in Switch Designer, when the user attempts to activate the flow in which the script resides, or just before invoking the jobArrived entry point (while the flow is running).

The third argument specifies the tags of the property that need to be validated.

This entry point returns and array of pair [{tag:"name of tag", valid: true/false}] where true means that the value of the specified property is deemed valid, false otherwise.The returned array must contain all the tags from the input argument. If some tag is missing, the corresponding property is considered invalid.

This entry point is not available for debugging.

Example:

```
async function validateProperties(s, flowElement, tags) {
  let retval = [];
  let tag, value;
  for (let i = 0; i < tags.length; i++) {
    tag = tags[i];
    value = await flowElement.getPropertyStringValue(tag);
    await flowElement.log(LogLevel.Info, "Custom validation of tag " + tag + " with
 value " + value);

    if (tag == "Separator") {
      if (value.length !== 1) {
        await flowElement.log(LogLevel.Error, "The value for the separator must be
 just 1 character long");
        retval.push({ tag: tag, valid: false });
      } else {
        retval.push({ tag: tag, valid: true });
      }

    } else if (tag == "PortNumber") {
      value = parseInt(value);
      if (isNaN(value) == true) {
        await flowElement.log(LogLevel.Error, "The specified value (%1) is not a
 number", [value]);
        retval.push({ tag: tag, valid: false });
      } else {
        if (value >= 1024 && value <= 65353) {
          retval.push({ tag: tag, valid: true });
        } else {
          await flowElement.log(LogLevel.Error, "The value of a port number must lie
 between 1024 and 65353");
          retval.push({ tag: tag, valid: false });
        }
      }
    }
    return retval;
  }
}
```

validateConnectionProperties(s: *Switch*, flowElement: *FlowElement*, c: *Connection*, tag: string[]): Promise<{ tag: string, valid: boolean }[]>

This entry point allows script writers to define their own custom validation logic for properties on outgoing connections (see *Validating property values* on page 32). It is invoked at appropriate times to validate all property values of the script's outgoing connections with "custom" validation, for example when the user changes the property value in Switch Designer, when the user attempts to activate the flow in which the script resides, or just before invoking the jobArrived entry point (while the flow is running).

The third and fourth argument specify the connection and the tag of the property involved in the operation.

This entry point returns and array of pair [{tag: <tag>, valid: <valid>}] where true means that the value of the specified property is deemed valid, false otherwise.The returned array must contain all the tags from the input argument. If some tag is missing, the corresponding property is considered invalid.

This entry point is not available for debugging.

findExternalEditorPath(s: *Switch*, flowElement: *FlowElement*, tag: string): Promise<string>

This entry point is invoked when an "External Editor" property editor is selected for one of the script's properties in the Switch Designer user interface. The third argument specifies the tag of the property involved in the operation.

This entry point returns a value that is used as the path for the external editor.

httpRequestTriggeredSync(request: HttpRequest, args: any[], response: HttpResponse, s: Switch): Promise<void>

The entry point is invoked immediately when the HTTP request is coming to the Switch Web Services if the script is subscribed to a specific HTTP method and URL path.

For example, if the script is subscribed to the method HttpRequest.Method.POST and the URL path '/next', the webhook must be sent to <Switch Web Services protocol>://<Switch address>:<Switch Web Services port>/scripting/next. The HTTP method POST must be used.

Note:

- The protocol and port to be used for receiving webhooks in the Node.js scripting environment can be found in the *Web services* category of the Switch preferences, NOT in the *Webhooks (legacy)* category.
- /scripting part of the URL is hard-coded and enables Switch Web Services to distinguish between webhooks and other API requests.

request is an instance of the HttpRequest class and allows script writers to get all the request parameters like query, headers, body, caller ip address etc. For more details, refer to the *HttpRequest class* description.

args is an array of the extra arguments, which was passed to the httpRequestSubscribe call.

response is an instance of the HttpResponse class, which allows to construct the response for the webhook request. For more details, refer to *HttpResponse* class.

s is an instance of Switch class.

Note:

- If this entry point does not exist, a default response is sent. The HTTP status code is 200, the response body is { "status": true }.
- The request size is limited to 1 Mb. If a bigger request is received, the HTTP status code 413 is sent back. No entry point is called.
- When a webhook is received, it is placed in the queue. A limit for pending requests in the queue is 10000 for a particular element. If the limit for the element is exceeded, the HTTP status code 429 is sent back. No entry point is called.
- The httpRequestTriggeredSync entry point must be executed in less than 1 minute, and will be aborted otherwise. The HTTP status code 524 is sent back if the entry point is aborted.
- The httpRequestTriggeredSync entry point is called in concurrent mode. That means that it can be executed in parallel with any other entry point including an other httpRequestTriggeredSync.

This entry point is not available for debugging.

httpRequestTriggeredAsync(request: HttpRequest, args: any[], s: Switch, flowElement: FlowElement): Promise<void>

The entry point is invoked for the HTTP request if:

- the httpRequestTriggeredSync entry point doesn't exist or
- the httpRequestTriggeredSync entry point succeeded and constructed a response with HTTP status code between 200 and 299.

request is an instance of the HttpRequest class and allows script writers to get all the request parameters like query, headers, body, caller ip address etc. For more details, refer to the *HttpRequest* class description.

args is an array of the extra arguments, which was passed to the httpRequestSubscribe call.

s is an instance of Switch class.

flowElement is an instance of the *FlowElement* class. It is a fully functional FlowElement and allows to create new jobs or call FlowElement.getJobs.

Note:

- This entry point is handled asynchronously by Switch Server.
- It complies with the same rules as the jobs coming to a jobArrived entry point of this same element. It includes concurrency mode (serialized or concurrent), number of concurrent processing tasks and number of slots for the element. For example, if the element is concurrent and the number of slots is set to 4 for the element, then 4 jobArrived + httpRequestTriggeredAsync entry points can be executed in parallel at the maximum.

Example:

```
async function flowStartTriggered(s : Switch, flowElement: FlowElement) {
  await s.httpRequestSubscribe(HttpRequest.Method.POST, '/next', [1, 'a'] /*not used
 later*/);

  await s.setGlobalData(Scope.FlowElement, 'ids', []);
  await s.setGlobalData(Scope.FlowElement, 'ready_ids', []);
}

async function jobArrived(s : Switch, flowElement: FlowElement, job: Job) {
    let ids = await s.getGlobalData(Scope.FlowElement, 'ids');
    ids.push({[job.getId()]: job.getName()});
    await s.setGlobalData(Scope.FlowElement, 'ids', ids);
};

async function httpRequestTriggeredSync(request: HttpRequest, args: any[], response:
 HttpResponse, s: Switch) {
  response.setStatusCode(200);
  let count = 1;
  if (request.query.count) {
    count = Number(request.query.count);
  }
  let ids = await s.getGlobalData(Scope.FlowElement, 'ids');
  let readyIds = await s.getGlobalData(Scope.FlowElement, 'ready_ids');

  for(let i = 0; i < count; ++i) {
    if (ids.length === 0) {
      break;
    }
    readyIds.push(ids.shift());
  }

  await s.setGlobalData(Scope.FlowElement, 'ids', ids);
  await s.setGlobalData(Scope.FlowElement, 'ready_ids', readyIds);

  response.setHeader('Content-Type', 'application/json');
  response.setBody(Buffer.from(JSON.stringify(readyIds)));
}

async function httpRequestTriggeredAsync(request: HttpRequest, args: any[], s:
 Switch, flowElement: FlowElement) {
  let readyIds = await s.getGlobalData(Scope.FlowElement, 'ready_ids');
```

```
  let idsToRequest = [];
  for (let readyId of readyIds) {
    idsToRequest.push(Object.keys(readyId)[0]);
  }
  if (idsToRequest.length > 0) {
    const jobs = await flowElement.getJobs(idsToRequest);
    for(const job of jobs) {
      await job.sendToSingle();
    }

    await s.setGlobalData(Scope.FlowElement, 'ready_ids', []);
  }
}
```

abort(s: Switch, flowElement: FlowElement, job: Job, abortData: any): Promise<void>

This entry point is invoked when the time-out for the other entry point has exceeded. It is executed by the script executor which is executing the main entry point at the same time. Use this entry point to gracefully interrupt some long-executed tasks and/or free resources.

abortData: any custom data the main entry point passed to Switch::setAbortData call.

The list of entry points that can be aborted by a time-out: jobArrived, timerFired, httpRequestTriggeredSync, httpRequestTriggeredAsync, flowStartTriggered, flowStopTriggered.

Note:

- The abort entry point has a maximum of 60 seconds to finish. After this time, the script executor is killed forcibly.
- If the abort entry point is absent, the script executor is killed immediately.
- The abort entry point will not be called if the script hangs on a synchronous operation. Try to avoid the usage of synchronous calls like readFileSync or readdirSync.
- If an entry point is aborted, it will not produce any jobs. If jobArrived is aborted, the job will be moved to the Problem jobs.

Example:

```
import fs from 'fs/promises';

interface ExecutorFunction<T> {
  (resolve: (value?: PromiseLike<T> | T) => void, reject: (reason?: any) => void):
 void;
}

class Abortable {
  private promise: Promise<string | undefined>;
  private abortController: AbortController;

  constructor(fn: ExecutorFunction<string>) {
    this.abortController = new AbortController();
    const abortSignal = this.abortController.signal;

    const wrappedExecutor: ExecutorFunction<string> = (resolve, reject) => {
      abortSignal.addEventListener('abort', () => {
        reject(new Error('Aborted!!!'));
      });
      fn(resolve, reject);
    };

    this.promise = new Promise(wrappedExecutor);
  }

  public async wait() {
    return this.promise;
  }
```

```
  public abort() {
    this.abortController.abort();
  }
}

async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
  let processor = new Abortable(async (resolve, reject) => {
    await sleep(10000000);
    resolve();
  });

  s.setAbortData({processor});

  try {
    await processor.wait();
  } catch (err: any) {
    // This will be called if abort entry point is executed.
    // And the message will be logged.
    await job.log(LogLevel.Error, err.message);
  }

  // This will be called if abort entry point is executed.
  // But it will not have any affect because the job will be moved to the Problem
 jobs.
  await job.sendToSingle();
}

async function sleep(ms: number) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}

async function abort(s: Switch, flowElement: FlowElement, job: Job, abortData: any) {
  const data = (await fs.readFile(await job.get(AccessLevel.ReadOnly))).toString();
  await job.log(LogLevel.Info, data);

  abortData.processor.abort();

  await job.log(LogLevel.Info, "Abort has been finished.");
}
```

4.3.2. Script expressions entry point

calculateScriptExpression(s: *Switch*, flowElement: *FlowElement*, job: Job): Promise<string | number | boolean>

Note: This is not a script entry point, but an entry point that can **only** be used **in script expressions** in the **Define script expression** editor that determine the value of a particular property of a flow element or connection. For more information, refer to the chapter on script expression in the Switch Reference Guide.

This entry point is invoked every time the property value (defined by the script expression in the 'Define script expression' editor) is calculated.

This entry point returns the calculated property value. Supported return types are string, number and boolean. The execution fails if something else is returned by the entry point.

Switch, FlowElement and Job instances passed to this entry point have certain limitations which are explained further in the document, but generally speaking you can use these instances only in read-only mode.

Note that this entry point is not available for debugging.

Example:

```
async function calculateScriptExpression(s: Switch, flowElement: FlowElement, job:
 Job): Promise<string | number | boolean> {
  return PdfDocument.getNumberOfPages(await job.get(AccessLevel.ReadOnly));
}
```

4.4. Switch class

The single instance of the Switch class is passed as an argument to the *script entry points*. It represents common Switch functionality.

4.4.1. Methods

4.4.1.1. Switch Server version

static getServerVersion(): number; NOT FOR SCRIPT EXPRESSIONS

Provides the current switch version number. It returns the version of the Switch server where the script is running.

Example:

```
async function jobArrived(s, flowElement, job)
{
    let result = s.getServerVersion();
    await flowElement.log(LogLevel.Info, 'Result: %1', [result]);
}
```

4.4.1.2. Managing global data

Note:

- **All these methods throw an error if no arguments are provided, or if the scope value is not supported**.
- A global data field should be read/written only once per entry point execution.
- If more than 100 fields must be stored, the script should combine them into one object that can be read/written in one operation.

async getGlobalData(scope: *Scope*, tag: string, lock?: boolean): Promise<any>

Returns the value of a global data variable with the specified scope and tag, or returns an empty string if no global data variable with that scope and tag was set.

Important: Any instance of Date (either tag value OR as part of tag value) will be returned as a string and should be parsed in the script itself.

The lock parameter (false by default) is optional and determines whether or not a lock is set on global data. A lock can be released by calling the setGlobalData function. If the script doesn't update the global data, the data remains locked until the end of the script.

async getGlobalData(scope: *Scope*, tags: string[], lock?: boolean): Promise<{tag: string, value: any}[]>

Returns a list of objects in the format defined above. The result will contain only the data for the provided tags. If non-existing tags are provided, the result will not contain the data for these tags.

Important: Any instance of Date (either tag value OR as part of tag value) will be returned as a string and should be parsed in the script itself.

The lock parameter (false by default) is optional and determines whether or not a lock is set on global data. A lock can be released by calling the setGlobalData function. If the script doesn't update the global data, the data remains locked until the end of the script.

async setGlobalData(scope: *Scope*, tag: string, value: any): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Sets the value of the global data with the specified scope and tag. The value can be any of type: string, number, boolean, object, Array, Date, or null.

async setGlobalData(scope: *Scope*, globalData: {tag: string, value: any}[]): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Note: We strongly recommend that tags created by scripts in Global scope always start with <companyname> for Switch.setGlobalData. This to prevent clashes with tags set by other scripts or by the end user. Scripts should not use EnfocusSwitch as namespace, to avoid clashes with global data tags created by Switch.

Sets the global data values for the specified tags and scope. If global data with the same tag name exists, the value of the global data will be replaced with the new one. The value can be any of type: string, number, boolean, object, Array, Date, or null.

async removeGlobalData(scope: *Scope*, tag: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Removes the global data for the specified scope and tag.

Note: It's recommended to remove global data as soon as it is not needed anymore.

async removeGlobalData(scope: *Scope*, tags: string[]): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Removes the global data for the specified scope and tags.

Note: It's recommended to remove global data as soon as it is not needed anymore.

4.4.1.3. Webhooks

async httpRequestSubscribe(method: HttpRequest.Method, path: string, args: any[]): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Subscribes to incoming webhook requests that use the HTTP *method* to URL *path*. For the allowed values for *method*, see the *HttpRequest.Method* enum. The extra *args* provided here will be passed to the httpRequestTriggeredSync and httpRequestTriggeredAsync entry points.

The size of the incoming webhook request is limited to 1MB. HTTP error 413 will be returned to the webhook caller in case of bigger incoming requests.

When an actual webhook is received by the Switch Web Services, the entry point httpRequestTriggeredSync is invoked if it exists. It allows to provide the HTTP response to the webhook caller. See *httpRequestTriggeredSync* entry point description for more details.

Next to that, the httpRequestTriggeredAsync entry point is invoked if it exists. See the *httpRequestTriggeredAsync* entry point description for more details.

Note that for security reasons we advise using URL paths that are not easy to guess and keeping them private.

Example:

```
async function timerFired(s: Switch, flowElement: FlowElement) {
  try {
    await s.httpRequestSubscribe(HttpRequest.Method.POST, '/next', [1, 1]);
  } catch(error) {
    flowElement.failProcess('Failed to subscribe to the request %1', error.message);
  }
}
```

async httpRequestUnsubscribe(method: HttpRequest.Method, path: string): Promise<void>
NOT FOR SCRIPT EXPRESSIONS

Cancels a subscription. The arguments must have the same values as used for the corresponding subscribe call. If there is no matching active subscription found, the function will throw an exception. Note that all subscriptions for the element instance are cancelled automatically when the flow is stopped.

4.4.1.4. Aborting scripts

setAbortData(abortData: any): void NOT FOR SCRIPT EXPRESSIONS

Stores any data that later can be used in an abort entry point.

4.4.1.5. Translation support

static tr(str: string): string NOT FOR SCRIPT EXPRESSIONS

Marks a string literal for translation. It allows SwitchScriptTool to recognize the strings which must be gathered for translation.

Example:

```
async function jobArrived(s : Switch, flowElement: FlowElement, job: Job) {
  await job.log(LogLevel.Info, Switch.tr("Job name is %1"), [job.getName()]);
```

```
}
```

4.4.1.6. Getting global user preferences

getPreferenceSetting(settingKey: string): Promise<any> NOT FOR SCRIPT EXPRESSIONS

Retrieves the preference setting based on the provided setting key from the database. If only the settingGroup is specified, it returns the entire settingGroup object. If both settingGroup and settingName are specified, it returns the specific setting value.

User preference	Property	settingKey
Mail send	Sender name in emails	"mailSend/senderNameInEmail"
	Sender email address	"mailSend/senderEmailAddress"
HTTP proxy	Use HTTP proxy	"httpProxy/useHttpProxy"
	Proxy server	"httpProxy/httpProxyHost"
	Proxy port	"httpProxy/httpProxyPort"
	Authenticate	"httpProxy/httpProxyAuthRequired"
	Bypass proxy for these servers	"httpProxy/httpProxyBypassHosts"
Processing	Server language	"processing/language"
	Language environment	"processing/languageEnvironment"
Problem alerts	Send problem alerts to	"problemAlerts/sendProblemAlertsTo"
Application data	Application data root	"applicationData/applicationDataRoot"
	Datasets folder	"applicationData/datasetsFolder"
	Job tickets folder	"applicationData/ticketsFolder"
Web services	Port for the Switch Web Service	"webServices/switchWebServicePort"
	Port for the Switch Web Portal	"webServices/switchWebPortalPort"
	Protocol	"webServices/protocol"
	Privacy policy page	"webServices/privacyPolicyPage"
Reporting	Dashboard Service location	"dashboardService/location"
	The complete URL to the Dashboard Service <protocol>://<Address>:<Port> Example: *http://127.0.0.1:55092*	"dashboardService/url"
	Keep historical data for (days)	"dashboardService/keepHistoricalDataforDays"

User preference	Property	settingKey
Remote processing	HTTP port	"SwitchEnvironmentService/httpPort"
	Enable HTTPS	"SwitchEnvironmentService/useHttps"
	Port	"SwitchEnvironmentService/httpsPort"

4.5. FlowElement class

The single instance of the FlowElement class is passed as an argument to the *script entry points* that operate in the context of a particular flow element.

4.5.1. Methods

Note: Methods with the word 'async' return a Promise. It is therefore recommended to call these functions with 'await'. The other functions can be called as they are.

getName(): string

Returns the name of the flow element.

getFlowName(): string

Returns the name of the flow.

getOutConnections(): *Connection*[] NOT FOR SCRIPT EXPRESSIONS

Returns a list of Connection instances representing the outgoing connections for the flow element associated with this script. The list is in arbitrary order. If there are no outgoing connections, the list is empty.

setTimerInterval(seconds: number): void NOT FOR SCRIPT EXPRESSIONS

Sets the interval, in seconds, between subsequent invocations of the timerFired entry point (if it has been declared in the script). The implementation guarantees only that the time between invocations will not be less than the specified number of seconds. Depending on run-time circumstances the actual interval may be (much) longer.

The default value for the timer interval is 300 seconds (5 minutes).

Note: setTimerInterval can only be called from a timerFired entry point!

async getPropertyStringValue(tag: string): Promise<string | string[]> NOT FOR SCRIPT EXPRESSIONS

Returns the value of a custom script property as string. The property for which to return the value is specified by its property tag.

The function throws an "invalid tag" error, if the property with the given tag

- is not defined in the script declaration OR
- is a dependent property which is not visible for the current value of the parent property.

In case of an OAuthToken property type, the function resolves with a valid OAuth2 token. It is refreshed automatically if needed.

Note: Properties containing variables or script expressions are available in the jobArrived entry point only. An attempt to get the value of such properties in timerFired or any entry point other than jobArrived will throw an exception.

getPropertyType(tag: string): *PropertyType* NOT FOR SCRIPT EXPRESSIONS

Returns the property value type. In case a property value can be entered via different editors, the property value type is required to correctly interpret the property value. The function throws an error if the property with the given tag

- is not defined in the script declaration OR
- is a dependent property, which is not visible for the current value of the parent property.

See the *PropertyType* enumeration for the possible return values.

Note: Properties containing variables or script expressions are available in the jobArrived entry point only. An attempt to get the type of such properties in timerFired or any entry point other than jobArrived will throw an exception.

hasProperty(tag: string): boolean NOT FOR SCRIPT EXPRESSIONS

Returns true if a property with the specified tag is available for the flow element, otherwise returns false.

Note: Properties containing variables or script expressions are available in the jobArrived entry point only. Calling hasProperty for such properties in timerFired or any entry point other than jobArrived will return false in this case.

getPropertyDisplayName(tag: string): string NOT FOR SCRIPT EXPRESSIONS

Returns the untranslated name of the property as visible in the user interface in English. This method is mostly intended for including the property name in a log message.

failProcess(message : string, messageParam?: (string | number | boolean)): void NOT FOR SCRIPT EXPRESSIONS

Logs a fatal error for this flow element with the specified message and puts the element instance in the "problem process" state. *messageParam* is used as a substitute for %1 in the message.

Example:

```
flowElement.failProcess('Something went wrong with the %1 flow', 'Superman');
```

Note: When used in entry points other than jobArrived and timerFired, this method doesn't fail the element, it just logs the error message.

async log(level: *LogLevel*, message: string, messageParams?: (number | boolean | string)[]): Promise<void>

Logs a message of the specified level for this flow element.

Returns when the message is successfully logged, otherwise throws an error.

Example:

```
async function() {
  try {
    await flowElement.log(LogLevel.Warning, 'You are reaching the limits: %1 of %2
 attempts remaining.', ['only 1', 10]);
  } catch (error) {
    // do something in case of an error
}
```

async createJob(path: string): Promise<Job> NOT FOR SCRIPT EXPRESSIONS

Creates a new job from a file or folder that already exists at the specified path. Returns a Job instance representing a new job with default values that does not correspond to an incoming job. It should be separately routed using sendTo methods.

If the new job should inherit the job properties of the input job, then use the method Job::createChild instead.

Note:

- This method is valid only in *jobArrived*, *timerFired* and *httpRequestTriggeredAsync* entry points.
- The changes done to the job between creating and routing the job will be lost.

Example:

```
async function timerFired(s, flowElement) {
    try {
        const newJob...;
        await newJob.sendToSingle();
    } catch(e) {
        //catch the error
    }
    //remove ...
}
```

Note: The file or folder that was passed to createJob will not automatically be removed by Switch when sending or failing the job. Therefore it's up to the script to make sure that temporary files or folders passed to createJob are correctly removed after sending or failing the job.

getPluginResourcesPath(): string NOT FOR SCRIPT EXPRESSIONS

For scripted plug-ins, returns the location of the script resources folder.

This function is intended for use by scripted plug-ins and should not be used from regular script packages except for testing; there is no reliable way to transport external resources with regular script packages.

When used in regular script packages for testing, this function returns the path to the folder where the script package is located.

When used in *script folders* for testing, this function returns the path to the script folder.

getScriptDataPath(): string NOT FOR SCRIPT EXPRESSIONS

This function returns the path to the Script Data folder which serves as a common repository for custom "global" Switch script data. It is shared by all scripts and users, so callers must provide unique names for any files stored in this folder.

This is a subfolder of the Switch application data root, which has the advantage that its contents are relocated with all other information relevant to the operation of Switch and can be easily located for diagnostic purposes.

If such a folder does not exist, the function tries to create it.

subscribeToChannel(channelId: string, backingFolderPath: string): <void>; NOT FOR SCRIPT EXPRESSIONS

The methods job.sendToChannel and flowElement.subscribeToChannel allow elements to exchange jobs without requiring any direct connection between them in the flow. The sender and receiver can be located in the same or in different flows.

Each channel should have its own Id that should be unique for the complete Switch environment to prevent jobs being sent to/received from the wrong elements. Once the receiving element has successfully subscribed, jobs that are sent to the subscribed channel will come in through its jobArrived entrypoint. i.e. in the same way as if there would be a direct connection between both elements.

One sender element can send a job to just one receiver element. When multiple receiver elements subscribe to the same channel, only the element that was activated first will receive it. Subscribing to a channel that is already active does not throw an error, so it is up to the script writer to manage this, or up to the flow designer not to mix channel names.

One receiver element can receive jobs from multiple sender elements.

channelId: identifies the channel to receive jobs from.

backingFolderPath: folder path where received files will be stored until they are routed by the receiving element.

Note: subscribeToChannel can only be called from a flowStartTriggered entry point. On flow deactivation, the element will get automatically unsubscribed.

See also *sendToChannel*

Example:

```
async function flowStartTriggered(s, flowElement)
{
 flowElement.subscribeToChannel( 'channelToSubscribe', '/path/to/a/folder' );
}
```

async getJobs(ids: string[]): Promise<Job[]> NOT FOR SCRIPT EXPRESSIONS

Returns a list of job instances representing all the jobs currently waiting in the input folders for the flow element.

The list includes all jobs that have "arrived" in the jobArrived entry point, including the job passed to the current invocation of the jobArrived entry point. If there are no such jobs, the list is empty.

Note: For optimal performance of the script and to reduce the load on the server, this call should only be used to request up to 10.000 jobs that are known to be ready for processing. Job IDs to be passed to this call and data needed to decide when each job will be ready to be processed, can be calculated based on the jobArrived notification of the particular job and stored as global data to retrieve the result later.

This method must not be called more than once per entry point execution.

It is NOT allowed to get private data and metadata for jobs returned by the call or for child jobs created from jobs that were returned by this call in the same entry point execution. We do allow setting private data and setting metadata for jobs returned by get jobs.

It is not possible to use this call in a jobArrived and httpRequestTriggeredAsync entry points within a concurrent script. In that case, an error will be returned.

Example:

```
async function jobArrived(s, flowElement, job) {
  let jobsInfo = await s.getGlobalData(Scope.FlowElement, 'jobsInfo') || {};
  const jobId = job.getId();
  jobsInfo[jobId] = (jobsInfo[jobId] ? jobsInfo[jobId] : new Date().getTime());
  await s.setGlobalData(Scope.FlowElement, 'jobsInfo', jobsInfo); // collect jobs IDs
 to use them afterwards as parameter for the getJobs call
}

async function timerFired(s, flowElement) {
  await flowElement.setTimerInterval(30); // interval for 'holding' jobs
  const jobsInfo = await s.getGlobalData(Scope.FlowElement, 'jobsInfo');
  const currentTimestamp = new Date().getTime();
  let filteredIds = [];
  for (let id in jobsInfo) { // choose only jobs that have been staying in the flow
 element for at least 1 hour
    if ((jobsInfo[id] + (60 * 60 * 1000)) <= currentTimestamp && filteredIds.length
 <= 10000) {
      filteredIds.push(id);
      delete jobsInfo[id];
    }
  }
  if (filteredIds.length > 0) {
    const jobs = await flowElement.getJobs(filteredIds); // note: returns max 10000
 jobs
    for (let job of jobs) {
      await job.sendToSingle(); // move each job to the outgoing connection
    }
    await s.setGlobalData(Scope.FlowElement, 'jobsInfo', jobsInfo);
  }
}
```

When a flow is started, the timerFired entry point is triggered before any jobs arrive. During this initial period, calling FlowElement.getJobs() will return an empty array, even if jobs are already present in the input folder. This will remain the case until a jobArrived entry point has been triggered within the flow.

This behavior can lead to issues when global data is used to track job IDs. For example, if a script attempts to skip already processed jobs in the jobArrived entry point and cleans or processes jobs in the timerFired entry point, valid job IDs may be incorrectly removed from global data. This can in turn cause them to be processed as newly arrived jobs in the jobArrived entry point.

To avoid this issue, the sample script below demonstrates how this can be avoided:

- A flag is stored in global data to indicate whether at least one job has arrived since the flow started.
- The timerFired entry point checks this flag before calling FlowElement.getJobs(), ensuring that it only attempts to retrieve and clean jobs after a jobArrived call has occurred.
- This avoids premature cleanup of waiting job IDs and maintains accurate job tracking across flow restarts.

Example:

```
async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
  // Get array of waiting jobs from global data, locking it to avoid race conditions
 with other concurrent jobs or timerFired
```

```
  const waitingJobIds: string[] = (await s.getGlobalData(Scope.FlowElement,
 "waitingJobIds", true)) || [];

  // Update array of waiting jobs, adding the current job's unique prefix
  await s.setGlobalData(Scope.FlowElement, "waitingJobIds", [...waitingJobIds,
 job.getId()]);

  // Set global data to indicate that a job already arrived so that we know this when
 timerFired is called
  await s.setGlobalData(Scope.FlowElement, "jobArrivedAlready", true);
}

async function timerFired(s: Switch, flowElement: FlowElement): Promise<void> {
  // Set timer interval to 5 minutes
  flowElement.setTimerInterval(5 * 60);

  // Check global data to see if a job arrived since starting the flow
  const jobArrivedAlready: boolean = await s.getGlobalData(Scope.FlowElement,
 "jobArrivedAlready");

  // If not, FlowElement.getJobs() will not return jobs, so we will stop here
  if (!jobArrivedAlready) return;

  // Get array of waiting job IDs from global data, locking it to avoid race
 conditions with arriving jobs
  const waitingJobIds: string[] = (await s.getGlobalData(Scope.FlowElement,
 "waitingJobIds", true)) || [];

  // Get waiting jobs
  const waitingJobs = await flowElement.getJobs(waitingJobIds);

  // Filter out any waiting job IDs which do not have an associated waiting job, i.e.
 a job was not returned by FlowElement.getJobs()
  const cleanedWaitingJobs = waitingJobs.filter((waitingJob) => {
    // For each waiting job, include it in the result only if it's unique prefix is
 found in the array of waiting job IDs
    return waitingJobIds.some((jobId) => jobId === waitingJob.getId());
  });

  // Handle waiting jobs...

  // Clean global data by updating with only the waiting jobs with an associated job,
 accounting for manual deletions
  // IMPORTANT => Without the check for previously arrived jobs since flow start, this
 would lead to removing all job IDs from the array!
  await s.setGlobalData(Scope.FlowElement, "waitingJobIds", cleanedWaitingJobs);
}

async function flowStopTriggered(s: Switch, flowElement: FlowElement): Promise<void> {
  // Mark "jobArrived" global data to indicate that no job has arrived yet, for next
 time the flow is started
  await s.setGlobalData(Scope.FlowElement, "jobArrivedAlready", false);
}
```

createPathWithName(name: string, createFolder: boolean): Promise<any>; NOT FOR SCRIPT EXPRESSIONS

Note: Apps/Scripts using the createPathWithName method are only supported in Switch 2024 Fall and later.

This method serves the purpose of generating a temporary folder inside for example:

```
 C:\Users\<username>\AppData\Local\Temp\SwitchNodeScriptExecutor\<flowID>
\NodeScriptElement\<random_string>\sample
```

Each time this method is invoked in entry points, it generates a unique folder with a random value. Once the Node script executor is discarded, it will clean up those created folders.

The creation of the temporary folder depends on user input. For instance, consider the following invocation:

```
let result = await flowElement.createPathWithName("sample", true);
```

In this case, by specifying the folder name as "sample" and setting the boolean parameter to true, the function automatically generates the "sample" folder inside the path:

```
C:\Users\RAUR\AppData\Local\Temp\SwitchNodeScriptExecutor\24\NodeScriptElement\uuujd
\sample
```

Conversely, if the invocation is as follows:

```
let result = await flowElement.createPathWithName("sample", false);
```

The "sample" folder will not be created inside the "temp" directory. Instead, a folder with the path:

```
C:\Users\RAUR\AppData\Local\Temp\SwitchNodeScriptExecutor\24\NodeScriptElement
\uuujdyu\
```

is created, and the path to the "sample" folder is provided to the user as:

```
C:\Users\RAUR\AppData\Local\Temp\SwitchNodeScriptExecutor\24\NodeScriptElement\uuujd
\sample
```

Example:

```
main.ts

async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
    try {
        let result = await flowElement.createPathWithName("sample", true);
        await flowElement.log(LogLevel.Info, 'Result: %1', [result]);
        await job.sendToSingle(job.getName());
    } catch (error) {
        // Handle any errors that occur during execution
        await flowElement.log(LogLevel.Error, `Error in jobArrived: ${error}`);
    }
}
```

4.6. Job class

An instance of the Job class represents a job (file or job folder) waiting to be processed in one of the input folders for the flow element associated with the script. The third argument passed to the *jobArrived* entry point is the newly arrived job that triggered the entry point's invocation.

New job objects can be created using the createJob function of the *FlowElement class*.

Processing a job in a script usually consists of the following steps:

- Decide on how to process the job based on file type etc.
- Get the path to the incoming job using Job::get() call with the appropriate access level.
- Generate a temporary path for one or more output files or folders.
- Create the output(s) in the temporary location.
- Create the output job(s) using Job::createChild() call.

- In addition to (or instead of) creating new jobs it is possible to modify the incoming job.
- Call one of the sendTo functions for each output.

If the incoming job is passed along without change, the Job::sendTo() functions can be called directly on the incoming job object, skipping all intermediate steps.

A job remains in the input folder until one of the Job::sendTo() or Job::fail() functions has been called for the job. The jobArrived entry point will be invoked only once for each job, so if the entry point does not call a sendTo() or fail() function for the job, the script should do so at a later time (in a timerFired entry point or in a subsequent invocation of the jobArrived entry point for a different job).

Note: After the flow restarts, the jobArrived entry point will be invoked once again for all jobs in the input folder.

4.6.1. Methods

4.6.1.1. Getting job information

Note: To get the path to the input job look under *Accessing job content*.

getName(includeExtension: boolean = true): string

Returns the file or folder name for the job, but excluding the unique filename prefix (job ID). By default the returned name includes the file extension. The user can exclude the file extension by setting the *includeExtension* argument to false.

getId(): string

Returns the unique job ID. This is a filename prefix used for the job, without the underscores. For example, for a job named "_0G63D_myjob.txt", this function would return "0G63D".

Callers should not rely on the syntax of the returned string as this may change between Switch versions.

isFile(): boolean

Returns true if the job is a single file, false otherwise.

isFolder(): boolean

Returns true if the job is a folder, false otherwise.

getVariableAsString(variable: string): Promise<any>

Takes the variable format from the Switch Designer and based on the metadata options (Boolean, Integer, Date, Text, Rational, TextIndexed), it outputs the result as a string.

Example:

```
async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
    try {
        let result = await
 job.getVariableAsString('[Metadata.Rational:Dataset="JSON",Model="JSON",Path="dc:creator/
*[1]"] ');
         await flowElement.log(LogLevel.Info, 'Result: %1', [result]);
    } catch (error) {
    await flowElement.log(LogLevel.Error, `Error retrieving variable: ${error}`);
```

```
    }
  }
```

Note: There are still some limitations to this method:

- Text indexing is not supported in XML, XMP and JDF.
- Embedded HTML is not supported.
- In the Model option, Automatic selection is not supported. You can choose either JDF, XML, XMP or JSON.

This method only works with external types such as XMP, XML, JSON and JDF.

4.6.1.2. Accessing job content

async get(accessLevel: *AccessLevel*): Promise<string>

Returns the path to the job on the file system. It allows the user to read file/folder contents (if called with AccessLevel.ReadOnly) and/or manipulate it (if called with AccessLevel.ReadWrite). Any file/folder modification will be detected and uploaded automatically on the routing stage (see *Routing a job* on page 96).

Note: If job content modification is detected:

- If called with *AccessLevel.ReadOnly*, an error is thrown.
- If called with *AccessLevel.ReadWrite*, the job content is updated with the modified content.

The above applies to ***Job::sendToSingle, Job::sendTo, Job::sendToLog and Job::sendToData.***

Throws an error if:

- Unknown ***AccessLevel*** is provided;
- Any error happens when job content is transferred/accessed.

Remark: This method can be used in script expressions but only with AccessLevel.ReadOnly.

Example 1:

```
const fs = require("fs-extra");

async function jobArrived(s, flowElement, job) {
  // READ-WRITE example
  const tempPath = await job.get(AccessLevel.ReadWrite);
  if (job.isFile()) {
    await fs.copy(sourceFile, tempPath); // i.e. replace job file with another file
  } else {
    // i.e. replace with your own folder structure
    await fs.emptyDir(tempPath);
    await fs.copy(sourceFolder, tempPath);
  }
  await job.sendToSingle(); //at this stage, new job content will be uploaded
 automatically;
}
```

Example 2:

```
const fs = require("fs-extra");

async function jobArrived(s, flowElement, job) {
```

```
  // READ-ONLY example
  const jobPath = await job.get(AccessLevel.ReadOnly);
  if (job.isFile()) {
    const content = await fs.readFileSync(jobPath, { encoding: "utf8" }); // i.e.
 read file
    content;
    await job.log(LogLevel.Info, "Job content is: " + content);
  } else {
    const folderStructure = await fs.readdir(jobPath); //i.e. read folder structure
    await job.log(LogLevel.Info, "Job content structure is: " + folderStructure);
  }
  await job.sendToSingle(); // on this stage, new dataset content will be uploaded
 automatically;
}
```

4.6.1.3. Logging messages

async log(level: *LogLevel*, message: string, messageParams?: (number | boolean | string)[]): Promise<void>

Logs a message of the specified level for this job, automatically including the appropriate job information.

If the message string contains references to non-existing message parameters, an error is thrown. If you want to log a message that contains '%' , pass it in the message parameters:

```
await job.log(LogLevel.Info, '%1', [message]);
```

Example:

```
async function() {
  try {
    await job.log(LogLevel.Warning, 'Something takes longer for job %1 with name "%2"
 ',[job.getId(), job.getName()])
  } catch (error) {
    // do something in case of an error
}
```

4.6.1.4. Manipulating private data

async getPrivateData(tag: string | EnfocusSwitchPrivateDataTag): Promise<any>

Returns the value of the private data with the specified tag, or an empty string if no private data with that tag was set for the job.

Important: Any instance of Date (either tag value OR as part of tag value) will be returned as a string and should be parsed in the script itself.

See examples below.

async getPrivateData(tags?: (string | EnfocusSwitchPrivateDataTag)[]): Promise<{tag: string | EnfocusSwitchPrivateDataTag, value: any}[]>

Returns:

- If tags are provided, a list of objects in the format defined above. The result will contain only data for the provided tags. If non-existing tags are provided, the result will not contain the data for these tags.
- If tags are not provided, a list of objects in the format defined above. The result will contain all available private data for the job.

Important: Any instance of Date (either tag value OR as part of tag value) will be returned as a string and should be parsed in the script itself.

See examples below.

async setPrivateData(tag: string | EnfocusSwitchPrivateDataTag, value: any): Promise<void>
NOT FOR SCRIPT EXPRESSIONS

Sets the value of the private data with the specified tag to the specified value. The value can be any of type: string, number, boolean, object, Array, Date, or null. If private data with the same tag name exists, the value of the private data will be replaced with the new one.

Note: Throws an error if no arguments are provided, or if the **value** argument is missing.

async setPrivateData(privateData: {tag: string | EnfocusSwitchPrivateDataTag, value: any}[]): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Note: We strongly recommend that tags created by scripts always start with <companyname> for job.setPrivateData. This to prevent clashes with tags set by other scripts or by the end user. Scripts should not use EnfocusSwitch as namespace, to avoid clashes with private data tags created by Switch.

Sets the private data values for the specified tags. The value can be any of type: string, number, boolean, object, Array, Date, or null. If private data with the same tag name exists, the value of the private data will be replaced with the new one.

Note: Throws an error if no data is specified or if **privateData** is not an array.

Example for getPrivateData and setPrivateData:

```
const expectedPrivateData = [
        { tag: 'tag1', value: 1 },
        { tag: 'tag2', value: true },
        { tag: 'tag3', value: {key: [2,3,4]} },
        ];
        try {
        await job.setPrivateData('tag4', 'value4');
        await job.setPrivateData(EnfocusSwitchPrivateDataTag.userEmail,
 'john.doe@example.com')
        await job.setPrivateData(expectedPrivateData);
        let actual = await job.getPrivateData();
        console.log('get all private data:', actual);
        actual = await job.getPrivateData('tag1');
        console.log('get one existing:', actual);
        actual = await job.getPrivateData(EnfocusSwitchPrivateDataTag.userEmail);
        console.log('get for predefined:', actual);
        actual = await job.getPrivateData('nonExisting');
        console.log('get one non-existing:', actual);
        actual = await job.getPrivateData(['tag2', 'tag3']);
        console.log('get specific:', actual);
        await job.sendToSingle();
        } catch (e) {
        console.log(e);
        job.fail(`${job.getName()} : ${e.message}`, []);
        }
        /* Output:
        get all private data: [
        { tag: 'BeingProcessedByRemoteProcessElement_7.2', value:'1' },{ tag: 'tag4',
 value: 'value4' },
        { tag: 'tag1', value: 1 },
        { tag: 'tag2', value: true },
```

```
        { tag: 'tag3', value: {key: [2,3,4]} }
        ]
        get one existing:1
        get for predefined:john.doe@example.com
        get one non-existing:
        get specific: [ { tag: 'tag2', value: true }, { tag: 'tag3', value: {key:
 [2,3,4]} } ]
        --- End of output
```

async removePrivateData(tag: string | EnfocusSwitchPrivateDataTag): Promise<void>

NOT FOR SCRIPT EXPRESSIONS

Removes private data with the specified tag from the job.

Note: Throws an error if no **tag** is specified.

async removePrivateData(tags: (string | EnfocusSwitchPrivateDataTag)[]): Promise<void>

NOT FOR SCRIPT EXPRESSIONS

Removes private data with the specified tags from the job.

Note: Throws an error if the **tags** are not specified, or if the array is empty.

Example:

```
try {
  await job.setPrivateData([
    { tag: 'survived1', value: 'value_survived1' },
    { tag: 'survived2', value: 'value_survived2' },
    { tag: 'toBeRemoved1', value: 'value_removed1' },
    { tag: 'toBeRemoved2', value: 'value_removed2' },
    { tag: 'toBeRemoved3', value: 'value_removed3' },
    { tag: EnfocusSwitchPrivateDataTag.userName, value: 'Admin'}
  ]);
  await job.removePrivateData('toBeRemoved1');
  await job.removePrivateData(['toBeRemoved2', 'toBeRemoved3']);
  await job.removePrivateData(EnfocusSwitchPrivateDataTag.userName);
  await job.sendToSingle();
} catch (e) {
  console.log(e);
  job.fail(`${job.getName()} : ${e.message}`, []);
}
```

4.6.1.5. Private data used by built-in elements

Some private data tags are reserved for Switch built-in elements and can be read by the scripts. Except for the EnfocusSwitch.origin these private data fields can be changed by scripts using job.setPrivateData calls.

EnfocusSwitch.hierarchy(string[])

An array with the location path segments in the hierarchy info associated with the job, or an empty array if there is no hierarchy info. The topmost path segment is stored at index 0.

EnfocusSwitch.emailAddresses (string[])

An array with the email addresses in the email info associated with the job, or an empty array if there are none.

EnfocusSwitch.emailBody (string)

The email body text in the email info associated with the job, or an empty string if there is none.

EnfocusSwitch.userName (string)

The short user name for the job, or an empty string if no user information has been associated with this job.

EnfocusSwitch.userFullName (string)

The full user name for the job, or an empty string if no user information has been associated with this job.

EnfocusSwitch.userEmail (string)

The user e-mail address for the job, or an empty string if no user information has been associated with this job.

EnfocusSwitch.origin (string)

An indication of the origin of the job before it was injected in the flow. For example, a Submit point writes the absolute path of the original job (on the client computer) in this tag. Currently this tag is read only and should never be written by the scripts. The updated value will not be used anywhere.

EnfocusSwitch.initiated (string)

Initiated date as string in "YYYY-MM-DD'T'hh:mm:ss" format (e.g. "2021-06-10T00:01:02").

The date and time when the job reached the first entry point in Switch.

EnfocusSwitch.submittedTo (string)

Display name of the Submit point to which the job was submitted.

EnfocusSwitch.state (string)

Job state. Setting this private data field from scripting will not be reflected in the Statistics pane and the History dashboard widgets/GraphQL API.

For possible values and examples for getPrivateData and setPrivateData, refer to *EnfocusSwitchPrivateDataTag enumeration*.

4.6.1.6. Manipulating datasets

async listDatasets(): Promise<{name: string, model: *DatasetModel*, extension: string}[]>

Returns a list of all datasets' names with models and extensions for which a dataset is associated with the job.

The returned list does not contain datasets created in the same entry point invocation.

async removeDataset(name: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Removes the dataset with the specified name for the job.

Note:

- Throws an error in case the dataset does not exist.
- Throws an error if no **name** is specified.

async createDataset(name: string, filePath: string, model: *DatasetModel*): Promise<void>

NOT FOR SCRIPT EXPRESSIONS

Creates a new dataset with the specified model and associates it with the specified name. The new dataset will be uploaded when any sendTo method for the job is called. When failing the job, the dataset will not be added. Changing the name, model or extension of a dataset is only possible by creating a new dataset and removing the old one.

The values allowed for the dataset model are: XML, JDF, XMP, JSON, and Opaque.

Note:

- Throws an error if neither name, filePath, nor model are specified.
- The file or folder that was passed to createDataset will not automatically be removed by Switch when sending or failing the job. Therefore it's up to the script to make sure that temporary files or folders passed to createDataset are correctly removed after sending or failing the job.

async getDataset(name: string, accessLevel: *AccessLevel*): Promise<string>

Returns the file path for the metadata dataset for the job associated with the specified name. When calling any sendTo method, the scripting module automatically uploads/replaces the dataset if the dataset was modified, i.e the returned dataset does not include changes made in the same entry point invocation. This also means that when failing the job, the dataset will not be changed.

Note: If dataset content modification is detected:

- If called with *AccessLevel.ReadOnly*, an error is thrown.
- If called with *AccessLevel.ReadWrite*, the dataset content is updated with the modified content.

The above applies to ***Job::sendToSingle***, ***Job::sendTo***, ***Job::sendToLog*** and ***Job::sendToData***.

Throws an error if:

- Unknown **AccessLevel** is provided;
- Any error happens when the dataset content is transferred/accessed.

Remark: This method can be used in script expressions but only with AccessLevel.ReadOnly.

Example 1:

```
const fs = require("fs-extra");

async function jobArrived(s, flowElement, job) {
  // READ-WRITE example
  const sourceFile = await flowElement.getPropertyStringValue("SourceFilePath");
  const tempPath = await job.getDataset(XMLDataset, AccessLevel.ReadWrite);
  await fs.copy(sourceFile, tempPath); // i.e. replace dataset file with another file
  await job.sendToSingle();
}
```

Example 2:

```
async function jobArrived(s, flowElement, job) {
 // READ-ONLY example
 const jobPath = await job.get(AccessLevel.ReadOnly);
```

```
 const tempPath = await job.getDataset(XMLDataset, AccessLevel.ReadOnly);
 const content = await fs.readFileSync(tempPath, { encoding: "utf8" }); // i.e. read
 file content;
 await job.log(LogLevel.Info, "Dataset content is: ", content); // no changes allowed
 (readonly operation)
 await job.sendToSingle();
}
```

4.6.1.7. Creating a new child job

async createChild(path: string): Promise<Job> NOT FOR SCRIPT EXPRESSIONS

Creates a child job from a file or folder that already exists at the specified path. Returns a job instance representing a new job that inherits the processing history, external metadata and private data from the 'parent' job. It should be separately routed using one of the sendTo methods.

If the new job should not inherit the properties of the input job, or if there is no input job as is usually the case inside the timerFired entry point, then use FlowElement::createJob.

Example:

```
async function jobArrived(s, flowElement, job) {
try {
const reportJob = await job.createChild(<path_to_report_file>);
await reportJob.sendToLog(Connection.Level.Success, 'XML');
await job.sendToData(Connection.Level.Success);
} catch(e) {
// catch an error
}
// remove <path_to_report_file> in case no longer needed
}
```

Note: The file or folder that was passed to createChild will not automatically be removed by Switch when sending or failing the job. Therefore it's up to the script to make sure that temporary files or folders passed to createChild are correctly removed after sending or failing the job.

4.6.1.8. Routing a job

After a job is sent to an output connection, no other calls for this Job object are allowed, except the other sendTo calls and Job::fail. It's a good practice to create a new or child job, if you want to route the original job to more than one connection. A child job is a new job that inherits all the properties of the parent job (job ticket, datasets, etc.). There is a new method in the Job class to create a child job. The creation of a child job was not possible in the legacy scripting.

It is allowed to call *sendTo* methods multiple times for the same incoming job, on the condition that it is not modified in the entry point. Sending a modified job multiple times will result in unpredictable behavior.

Contrary to the usage in legacy scripting, the original file or folder used for newly created jobs is NOT removed from its location when using a sendTo method, it is up to the script writer to do that (see *the example* at the end of this topic).

The *Job::fail* method can still be called after a sendTo method. In such a case it overrides the previously called sendTo method and fails the job. If none of the sendTo methods is called for a newly created job, the job is discarded. In case of an incoming job, the job stays in the input folder and its content is not modified.

async sendToNull(): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Marks the job as completed without generating any output.

async sendToSingle(newName?: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Sends the job to the single outgoing 'move' connection.

Throws an error in case there are no outgoing connections.

The optional argument *newName* allows to rename the job.

If the script is not configured to use 'move' connections, the element is failed with an appropriate error message.

If the flow element instance has two or more outgoing connections, the job is moved to only one connection.

Example:

```
await job.sendToSingle('newName.pdf');
await job.sendToSingle();
```

async sendTo(c: Connection, newName?: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Sends the job to the specified outgoing 'move' connection.

If the script is not configured to use 'move' connections, the element is failed with an appropriate error message.

The optional argument *newName* allows to rename the job.

Example:

```
const outConnections = flowElement.getOutConnections();
for (const connection of outConnections) {
  if (job.isFile() && connection.getName() === 'move') {
    await job.sendTo(connection, "sendTo.txt"); // here it is
  }
}
```

async sendToData(level: *Connection.Level*, newName?: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Sends the jobs to the outgoing "data" traffic light connections that have the specified connection level property enabled. The optional argument *newName* allows to rename the job.

If the script is not configured to use traffic light connections, this function logs an error and does nothing.

If the flow element has no outgoing connections of the specified level, the job is failed with an appropriate error message and moved to the problem jobs folder.

Example:

```
await job.sendToData(Connection.Level.Warning, 'data_warning.txt');
await job.sendToData(Connection.Level.Success, 'data_success.txt');
```

async sendToLog(level: *Connection.Level*, model: *DatasetModel*, newName?: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Sends the job to the outgoing "log" traffic light connections that have the specified connection level property enabled. The optional argument *newName* allows to rename the job.

For "data with log" traffic light connections, the job is attached as a metadata dataset to all "data" jobs, that is jobs routed via *SendToData*. The model argument defines the metadata dataset model and the metadata dataset name is taken from the "Dataset name" property of the outgoing connections.

Note that metadata datasets have automatically generated names and defined extensions. Therefore after the job is attached as a metadata dataset, its name is lost but the filename extension is preserved. If the *newName* argument is defined, the extension of newName will be used for the attached metadata datasets.

If the script is not configured to use traffic light connections, this function logs an error and does nothing.

If the flow element has no outgoing connections of the specified level, this function logs a warning and the job is discarded.

For possible values of the model argument, see *DatasetModel*. If the model argument has an unsupported value, an exception is thrown.

Example:

```
await job.sendToLog(Connection.Level.Error, DatasetModel.XML, 'log_error.xml');
await job.sendToLog(Connection.Level.Warning, DatasetModel.XML, 'log_warning.xml');
await job.sendToLog(Connection.Level.Success, DatasetModel.XML, 'log_success.xml');
```

Note: For "data with log" traffic light connections, the ConnectionLevel argument is ignored. For example, if one job is routed via sendToLog(Connection.Level.Error, ...) and another job via sendToLog(Connection.Level.Success, ...), the job that was routed last will be attached as a dataset in the end. To consider the level argument, use a "log" traffic light connection instead.

fail(message: string, messageParams?: (number | boolean | string)[]): void NOT FOR SCRIPT EXPRESSIONS

Logs a fatal error for the job with the specified message and moves the job to the problem jobs folder.

messageParams are used as substitutes for %1, %2 etc. in the message.

The job and its dataset's content is not modified.

Newly created datasets are discarded as well. Note that removing existing datasets and operations on the job's private data are preserved.

Newly created jobs are not moved to the problem jobs folder (i.e. a job created using 'createChild()' or 'createJob()' and failed during the same entry point invocation is not moved).

If the message string contains references to non-existing message parameters, an error is thrown. See *Log*.

Example:

```
job.fail('Something went wrong with the job %1', [job.getName()])
```

async processLater(seconds: number): Promise<void> NOT FOR SCRIPT EXPRESSIONS

Schedules the job to be processed at a later moment so that jobArrived will be called again for the same job and dynamic properties set on the element will be re-evaluated.

The <seconds> parameter specifies the minimum time interval after which Switch will schedule the job for processing again. The default value for the interval is 300 seconds (5 minutes).

When processLater is called from jobArrived, a minimum of 10 seconds will be applied and a warning will be logged if a smaller value was passed.

The job and its dataset's content are not modified. Newly created datasets are discarded as well. Note that removing existing datasets and operations on the job's private data are preserved.

processLater cannot be called on new or child jobs created in the current entry point.

Note: It is allowed to change the private data of the job.

Example:

```
const thePrivateData = await job.getPrivateData("ProcessLater");
 if (thePrivateData == 1) {
   await job.setPrivateData("ProcessLaterTime", '');
   await job.sendToSingle();
 } else {
   await job.setPrivateData("ProcessLater", 1);
   const delay = Number(await flowElement.getPropertyStringValue('Delay'));
   await job.processLater(delay);
 }
```

async sendToChannel(channelId: string, newName?: string): Promise<void> NOT FOR SCRIPT EXPRESSIONS

To be called from the sending element to move the incoming job to the subscribed receiving element in the same or other flow the moment that the current entry point finishes.

If there are no active subscribers for the channel, this method will not throw an error, instead an error will be logged and the job will remain in the input folder (e.g. if the flow of the receiving element wasn't activated yet).

channelId: identifies the channel to which the job needs to be sent.

The optional argument *newName* allows you to rename the job.

See also *subscribeToChannel*

Example:

```
await job.sendToChannel( 'channelId' );
```

Routing a job: Example

```
const fs = require("fs");
const rmdir = require("rimraf"); //recursively delete non-empty folders

async function timerFired(s, flowElement) {
  let fileToBeInjected = "/path/to/a/file";
  try {
    let newJob = flowElement.createJob(fileToBeInjected);
    await newJob.sendToSingle();
    fs.unlinkSync(fileToBeInjected); //delete the injected file
  } catch (err) {
    await flowElement.log(LogLevel.Error, err.message); //just log a message and do
 nothing
    return;
  }

  let folderToBeInjected = "/path/to/a/folder";
  try {
    let newJob = flowElement.createJob(folderToBeInjected);
```

```
    await newJob.sendToSingle();
    rmdir.sync(folderToBeInjected); //delete the non-empty injected folder
  } catch (err) {
    await flowElement.failProcess(err.message); //put the element in an error state
    return;
  }
}
```

4.6.1.9. Job priority

The job priority determines the order in which jobs are processed. For more information, refer to *"Job priorities" in the Switch Reference Guide*.

getPriority(): number

Returns the priority of the job

setPriority(priority: number): void NOT FOR SCRIPT EXPRESSIONS

Sets the priority of the job.

For possible values, refer to *Priority enumeration*.

4.7. Connection class

An instance of the Connection class represents an outgoing connection for the flow element associated with the script. Connection objects can be obtained through functions of the *FlowElement class*.

4.7.1. Methods

Note: Methods with the word 'async' return a Promise. It is therefore recommended to call these functions with 'await'. The other functions can be called as they are.

getName(): string NOT FOR SCRIPT EXPRESSIONS

Returns the name of the destination folder if the connection name is empty.

getId(): string NOT FOR SCRIPT EXPRESSIONS

Returns a string that uniquely identifies the connection (within the limits of the guarantees described below). Callers should not rely on the syntax of the returned string as this may change between Switch versions.

The element ID for a particular flow element offers the following fundamental guarantees:

- It differs from the element ID for any other flow element in any currently active flow.
- It remains unchanged as long as the flow in which it resides is not edited, even if the flow is deactivated and reactivated and across Switch sessions.
- The element ID for a connection is never equal to the element ID for a non-connection flow element.

Note that holding or releasing connections or renaming the flow does not count as editing in this context. However, exporting and re-importing (or upgrading) a flow, or renaming a flow element inside the flow, does count as editing.

getType(): string

Returns the type of the connection as one of the following strings: "Move", "Filter", "Traffic-data", Traffic-log", "Traffic-datawithlog".

async getPropertyStringValue(tag: string): Promise<string | string[]> NOT FOR SCRIPT EXPRESSIONS

Returns the value of a custom outgoing connection property as string. The property for which to return the value is specified by its property tag.

See *FlowElement.getPropertyStringValue*.

getPropertyType(tag: string): *PropertyType* NOT FOR SCRIPT EXPRESSIONS

Returns the connection property value type.

See *FlowElement.getPropertyType*.

hasProperty(tag: string): boolean NOT FOR SCRIPT EXPRESSIONS

Returns true if a property with the specified tag is available for the connection, otherwise returns false.

getPropertyDisplayName(tag: string): string NOT FOR SCRIPT EXPRESSIONS

Returns the name of the property as visible in the user interface. (However, this is an untranslated name.) This method is mostly intended for including the property name in a log message.

getFileCount(): Promise<number> NOT FOR SCRIPT EXPRESSIONS

Returns the number of files currently residing in the folder at the other end of this connection (the originating folder for an incoming connection, the target folder for an outgoing connection). If nested is false, only items directly inside the folder are counted (i.e. it counts each file directly present). If nested is true, the number of files in subfolders is counted as well, recursively (and subfolders themselves do not contribute to the count).

Example:

```
async function timerFired(s, flowElement) {

  var connlist = flowElement.getOutConnections();
  for(var i = 0; i < connlist.length; i++){
    var conn = connlist.at(i);
    var name = await conn.getName();
    await flowElement.log(LogLevel.Info, "name is: " + JSON.stringify(name));
    var fcount = await conn.getFileCount(false);  // count files present directly in
 connection
    await flowElement.log(LogLevel.Info, "File count: " + fcount);
    var fcountDeep = await conn.getFileCount(true); // deep count files present in
 connection
    await flowElement.log(LogLevel.Info, "File count (deep counted) : " +
 fcountDeep);
  }
}
```

4.8. ImageDocument class

The ImageDocument class allows retrieving certain information about image file contents (supported file formats: JPEG, TIFF, PNG). It reads the values from EXIF and XMP.

The class does not allow modifying file contents. Each ImageDocument instance references a particular file, which may be any accessible file (whether it is a job or not). All the static methods in this class might throw an exception, so it is advised to wrap it in a try-catch block.

4.8.1. Static methods

open(path: string): Promise<ImageDocument>;

Opens the image document to extract image data.

getWidth(path: string): Promise<number>;

Returns the valid image width, in pixels.

getHeight(path: string): Promise<number>;

Returns the valid image height, in pixels.

getColorMode(path:string):Promise<ImageDocument.ColorMode>;

Returns the color mode used.

getColorSpace(path:string):Promise<ImageDocument.ColorSpace>;

Returns the color space used.

getICCProfile(path:string):Promise<string>;

Returns the name of ICC color profile used (Photoshop only).

getSamplesPerPixel(path:string):Promise<number>;

Returns the number of components per pixel.

Example script

```
async function jobArrived(s:Switch, flowElement:FlowElement, job:Job){
  const jobPath = await job.get(AccessLevel.ReadOnly)
  try{
      const image: ImageDocument = await EnfocusSwitch.ImageDocument.open(jobPath);
      const imageWidth: number = image.getWidth();
      const imageHeight: number =  image.getHeight();
      const imageColorMode: ImageDocument.ColorMode =  image.getColorMode();
      const imageColorSpace: ImageDocument.ColorSpace =  image.getColorSpace();
      const imageICCProfile: string =  image.getICCProfile();
      const imageSamplesPerPixel: number =  image.getSamplesPerPixel();
      await job.log(LogLevel.Warning, `Result: ${imageWidth},
          height: ${imageHeight},
          colorMode: ${imageColorMode},
          colorSpace: ${imageColorSpace},
          ICCProfile: ${imageICCProfile},
          SamplesPerPixel: ${imageSamplesPerPixel}`, []);
      image.close();

      // Static methods
      const width: number = await EnfocusSwitch.ImageDocument.getWidth(jobPath)
      const height: number = await EnfocusSwitch.ImageDocument.getHeight(jobPath)
```

```
        const colorMode: ImageDocument.ColorMode = await
 EnfocusSwitch.ImageDocument.getColorMode(jobPath)
        const colorSpace: ImageDocument.ColorSpace = await
 EnfocusSwitch.ImageDocument.getColorSpace(jobPath)
        const ICCProfile: string = await
 EnfocusSwitch.ImageDocument.getICCProfile(jobPath)
        const samplesPerPixel: number = await
 EnfocusSwitch.ImageDocument.getSamplesPerPixel(jobPath)
        await job.log(LogLevel.Warning, `Result 2: ${width},
            height: ${height},
            colorMode: ${colorMode},
            colorSpace: ${colorSpace},
            ICCProfile: ${ICCProfile},
            SamplesPerPixel: ${samplesPerPixel}`, []);

        await job.sendToSingle();
    }
    catch (e) {
        job.fail(`${job.getName()} : ${e.message}`, []);
    }
}
```

4.8.2. Instance methods

close();

Closes the image file. This method should be called when an instance of the ImageDocument class is not required anymore.

getWidth(): number;

Returns the valid image width, in pixels.

getHeight(): number;

Returns the valid image height, in pixels.

getColorMode():ImageDocument.ColorMode;

Returns the color mode used.

getColorSpace():ImageDocument.ColorSpace;

Returns the color space used.

getICCProfile():string;

Returns the name of ICC color profile used (Photoshop only).

getSamplesPerPixel():number;

Returns the number of components per pixel.

Example script

```
  async function jobArrived(s:Switch, flowElement:FlowElement, job:Job){
        const jobPath = await job.get(AccessLevel.ReadOnly)
        try{
        const image: ImageDocument = await EnfocusSwitch.ImageDocument.open(jobPath);
        const imageWidth: number = image.getWidth();
        const imageHeight: number =  image.getHeight();
        const imageColorMode: ImageDocument.ColorMode =  image.getColorMode();
        const imageColorSpace: ImageDocument.ColorSpace =  image.getColorSpace();
        const imageICCProfile: string =  image.getICCProfile();
        const imageSamplesPerPixel: number =  image.getSamplesPerPixel();
        await job.log(LogLevel.Warning, `Result: ${imageWidth},
        height: ${imageHeight},
        colorMode: ${imageColorMode},
        colorSpace: ${imageColorSpace},
```

```
        ICCProfile: ${imageICCProfile},
        SamplesPerPixel: ${imageSamplesPerPixel}`, []);
        image.close();

        // Static methods
        const width: number = await EnfocusSwitch.ImageDocument.getWidth(jobPath)
        const height: number = await EnfocusSwitch.ImageDocument.getHeight(jobPath)
        const colorMode: ImageDocument.ColorMode = await
EnfocusSwitch.ImageDocument.getColorMode(jobPath)
        const colorSpace: ImageDocument.ColorSpace = await
EnfocusSwitch.ImageDocument.getColorSpace(jobPath)
        const ICCProfile: string = await
EnfocusSwitch.ImageDocument.getICCProfile(jobPath)
        const samplesPerPixel: number = await
EnfocusSwitch.ImageDocument.getSamplesPerPixel(jobPath)
        await job.log(LogLevel.Warning, `Result 2: ${width},
        height: ${height},
        colorMode: ${colorMode},
        colorSpace: ${colorSpace},
        ICCProfile: ${ICCProfile},
        SamplesPerPixel: ${samplesPerPixel}`, []);

        await job.sendToSingle();
        }
        catch (e) {
        job.fail(`${job.getName()} : ${e.message}`, []);
        }
        }
```

4.9. PdfDocument class

The PdfDocument class allows retrieving certain PDF information about PDF file contents.

The class does not allow modifying file contents.

Each PdfDocument instance references a particular file, which may be any accessible file (whether it is a job or not).

4.9.1. Static methods

Note:

- All methods in this class might throw an exception, so it is advised to wrap them in a try-catch block.
- The path in the following methods is always the absolute path to the PDF file.
- If you want to call multiple functions on one PDF file, it's advised to use the instance methods.

open(path: string): PdfDocument;

Constructs a PdfDocument instance associated with a file specified through its absolute file path.

Note: After having created the document instance, continue using the instance methods listed in *PdfDocument - Instance methods*.

getNumberOfPages(path: string): number;

Returns the number of pages in the PDF document.

getPDFVersion(path: string): string;

Returns the version of the PDF file format (for example: "1.6").

getPDFXVersion(path: string): string;

Returns the PDF/X version of the document, or an empty string if there is no PDF/X version.

Note: The PDF/X version indicates a claim that the document conforms to the PDF/X specification, but it does not offer any guarantees.

getSecurityMethod(path: string): string;

Returns the method used to protect the document.

getPageHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

Returns the height of the PDF page, in points. It does the same as getPageMediaBoxHeight.

If effective is true, then the page rotation and scaling are taken into account.

getPageWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

Returns the width of the PDF page, in points. It does the same as getPageMediaBoxWidth.

If effective is true, then the page rotation and scaling are taken into account.

getPageRotation(path: string, pageNumber: number = 1): number;

Returns the rotation of the PDF page, in degrees.

getPageScaling(path: string, pageNumber: number = 1): number;

Returns the scaling of the PDF page as a factor.

getPageLabel(path: string, pageNumber: number = 1): string;

Returns the page label of the PDF page, or an empty string if the page has no page label.

The following methods return the height or width of the corresponding page box of the PDF page, in points:

getPageMediaBoxHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageCropBoxHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageBleedBoxHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageTrimBoxHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageArtBoxHeight(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageMediaBoxWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageCropBoxWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageBleedBoxWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageTrimBoxWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

getPageArtBoxWidth(path: string, pageNumber: number = 1, effective: boolean = true): number;

Note: If effective is true, then the page rotation and scaling are taken into account.

Example script with a static method

```
try {
  // Get the number of pages directly using the static method
  const numPages: number = PdfDocument.getNumberOfPages(jobPath);
} catch (err) {
  job.fail("PDF error: %1", [err]);
}
```

Example script with both static and instance methods

```
let pdfDoc: PdfDocument;
try {
  // Open PDF file using the static open method
  pdfDoc = PdfDocument.open(jobPath);
  //Get the number of pages using the instance method
  const numPages: number = pdfDoc.getNumberOfPages();
  // Close the PDF document
  pdfDoc.close();
  pdfDoc = null;
} catch (err) {
  job.fail("PDF error: %1", [err]);
  if (pdfDoc) {
    pdfDoc.close();
  }
}
```

4.9.2. Instance methods

Note:

- All the instance methods in this class might throw an exception, so it is advised to wrap them in a try-catch block.
- An instance of the PdfDocument class is created by the static method PdfDocument.open.

close();

Closes the PDF file. This method should be called when an instance of the PdfDocument class is not required anymore.

getNumberOfPages( ): number;

Returns the number of pages in the PDF document.

getPDFVersion( ): string;

Returns the version of the PDF file format (for example: "1.6").

getPDFXVersion( ): string;

Returns the PDF/X version of the document, or an empty string if there is no PDF/X version.

Note: The PDF/X version indicates a claim that the document conforms to the PDF/X specification, but it does not offer any guarantees.

getSecurityMethod( ): string;

Returns the method used to protect the document.

getPage(pageNumber: number = 1): PdfPage;

Constructs a PdfPage instance for the PDF page with number <pageNumber>. The first page in the PDF has number 1.

getXMP(): XmpDocument;

Returns an XmpDocument instance with the contents of the XMP metadata stream of the PDF document.

4.10. PdfPage class

The PdfPage class allows retrieving certain PDF information about PDF page contents.

The class does not allow modifying page contents.

Each PdfPage instance references a particular PDF page in a file. It can be constructed using the PdfDocument.getPage call.

4.10.1. Methods

Note:

- All methods in this class might throw an exception, so it is advised to wrap them in a try-catch block.
- A PdfPage instance is created by calling the getPage instance method of the PdfDocument class. See *PdfDocument - Instance methods*.

getRotation(): number;

Returns the rotation of the PDF page, in degrees.

getScaling(): number;

Returns the scaling of the PDF page as a factor.

getPageLabel(): string;

Returns the page label of the PDF page, or an empty string if the page has no page label.

getHeight(effective?: boolean): number;

Returns the height of the PDF page, in points. It does the same as getMediaBoxHeight.

If effective is true, then the page rotation and scaling are taken into account.

getWidth(effective?: boolean): number;

Returns the width of the PDF page, in points. It is the same as getMediaBoxWidth.

If effective is true, then the page rotation and scaling are taken into account.

The following methods return the height and width of the corresponding page box of the PDF page, in points:

getMediaBoxHeight(effective?: boolean): number;

getCropBoxHeight(effective?: boolean): number;

getBleedBoxHeight(effective?: boolean): number;

getTrimBoxHeight(effective?: boolean): number;

getArtBoxHeight(effective?: boolean): number;

getMediaBoxWidth(effective?: boolean): number;

getCropBoxWidth(effective?: boolean): number;

getBleedBoxWidth(effective?: boolean): number;

getTrimBoxWidth(effective?: boolean): number;

getArtBoxWidth(effective?: boolean): number;

Note: If effective is true, then the page rotation and scaling are taken into account.

Example script

```
// Script that checks if all pages have identical page boxes
async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
  const jobPath = await job.get(AccessLevel.ReadOnly);

  let pdfDoc: PdfDocument;
  try {
    // Open PDF file
    pdfDoc = PdfDocument.open(jobPath);
    const numPages: number = pdfDoc.getNumberOfPages();

    let pageBoxesEqual: boolean = true;
    let pdfPage: PdfPage;
    let abHeight: number;
    let abWidth: number;
    let bbHeight: number;
    let bbWidth: number;
    let cbHeight: number;
    let cbWidth: number;
    let mbHeight: number;
    let mbWidth: number;
    let tbHeight: number;
    let tbWidth: number;

    if (numPages >= 1) {
      // Get page boxes for the first page
      pdfPage = pdfDoc.getPage(1);
      abHeight = pdfPage.getArtBoxHeight(false);
      abWidth = pdfPage.getArtBoxWidth(false);
      bbHeight = pdfPage.getBleedBoxHeight(false);
      bbWidth = pdfPage.getBleedBoxWidth(false);
      cbHeight = pdfPage.getCropBoxHeight(false);
      cbWidth = pdfPage.getCropBoxWidth(false);
      mbHeight = pdfPage.getMediaBoxHeight(false);
      mbWidth = pdfPage.getMediaBoxWidth(false);
      tbHeight = pdfPage.getTrimBoxHeight(false);
      tbWidth = pdfPage.getTrimBoxWidth(false);
    }
```

```
    // Go over all the pages
    for (let i = 2; i <= numPages; ++i) {
      pdfPage = pdfDoc.getPage(i);

      // Compare page boxes
      if (
        pdfPage.getArtBoxHeight(false) !== abHeight ||
        pdfPage.getBleedBoxHeight(false) !== bbHeight ||
        pdfPage.getCropBoxHeight(false) !== cbHeight ||
        pdfPage.getMediaBoxHeight(false) !== mbHeight ||
        pdfPage.getTrimBoxHeight(false) !== tbHeight ||
        pdfPage.getArtBoxWidth(false) !== abWidth ||
        pdfPage.getBleedBoxWidth(false) !== bbWidth ||
        pdfPage.getCropBoxWidth(false) !== cbWidth ||
        pdfPage.getMediaBoxWidth(false) !== mbWidth ||
        pdfPage.getTrimBoxWidth(false) !== tbWidth
      ) {
        pageBoxesEqual = false;
        break;
      }
    }

    // Close the PDF document
    pdfDoc.close();
    pdfDoc = null;

    if (pageBoxesEqual) {
      await job.sendToData(Connection.Level.Success);
    } else {
      await job.sendToData(Connection.Level.Error);
    }
  } catch (err) {
    job.fail("PDF error: %1", [err]);
    if (pdfDoc) {
      pdfDoc.close();
    }
  }
}
```

4.11. HttpRequest class

Represents the incoming webhook request.

4.11.1. Methods

Note: The methods in this class cannot be used in Node.js script expressions. NOT FOR SCRIPT EXPRESSIONS

getBodyAsString(): string

Returns the body of the request as a string.

4.11.2. Properties

method: HttpRequest.Method

The HTTP request method.

path: string

The request URL path ("/myscript/notify" for "*https://myswitch.com:51088/scripting/myscript/notify*"). Note that */scripting* is not a part of the *path* here.

query: { [propName: string]: string | string[] }

An object with the properties representing the query string parameters from the request's URL.

headers: { [header: string]: string }

An object with the properties representing the HTTP request headers (like 'Content-Type').

remoteAddress: string

The request's origin IP address.

body?: ArrayBuffer

The body of the request.

4.12. HttpResponse class

Represents the response to the webhook request.

4.12.1. Methods

Note: The methods in this class cannot be used in Node.js script expressions. NOT FOR SCRIPT EXPRESSIONS

setStatusCode(statusCode: number): void

Sets the status code of the response. The default status code is 200.

setHeader(name: string, value: string): void

Sets the header *name* to *value*.

setBody(data: ArrayBuffer | string): void

Sets the response body to *data*.

4.13. XmlDocument class

The XmlDocument class allows you to find certain information in XML documents by using Xpath 1.0.

This class does not allow modifying XML contents.

Each XmlDocument instance references an XML content loaded from file or memory.

4.13.1. Methods

Note: All the methods in this class might throw an exception, so it is advised to wrap them into a try-catch block.

static open(path:string): XmlDocument;

Constructs an XmlDocument instance associated with a file specified through its absolute file path.

static parse(xmlString: string): XmlDocument;

Constructs an XmlDocument instance associated with an XML string.

evaluate(xpath: string,prefixMap?: { [name: string]: string }): boolean | number | string | undefined;

Evaluates the XPath 1.0 expression against the the loaded XML content and returns the result.

Note: XPath does not support the default namespace concept, so you always have to explicitly provide a namespace prefix.

prefixMap is an object containing pairs of key and value where value is a string representing the namespace URI associated with the key-prefix. This enables the conversion between the prefixes used in the XPath expressions and the possibly different prefixes used in the document.

getDefaultNSMap():{ [name: string]: string };

Returns a new prefix map object that already contains all mappings that occur in the XML content.

The default namespace (if any) is included in the default map with two special prefixes: "default_switch_ns" and short version "dn". This allows XPath queries to refer to the default namespace using these special prefixes.

Example

```
/*
Use this xml as example
<?xml version="1.0" encoding="utf-8"?>
<root>
    <message>Hello world!</message>
    <jobComplete>false</jobComplete>
    <jobType>B</jobType>
    <price>4567.5</price>
</root>

list of xpath functions: https://developer.mozilla.org/en-US/docs/Web/XPath/Functions
*/

async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {

    let result = "";
    let dataset = "Xml";
    result = await getXpath(job, dataset, "string(/root/message)");
    result = await getXpath(job, dataset, "string(/root/price)");
    result = await getXpath(job, dataset, "ceiling(/root/price)");
    result = await getXpath(job, dataset, "floor(/root/price)");
    result = await getXpath(job, dataset, "string-length(/root/price)");
    result = await getXpath(job, dataset, "contains(/root/price, '5')");
    result = await getXpath(job, dataset, "starts-with(/root/price, '5')");
    result = await getXpath(job, dataset, "substring-after(/root/price, '5')");
    await job.sendToSingle();
}

async function getXpath(job: Job, dataset: string, xpath: string): Promise<string> {
    let result: string | number | boolean;
    try {
        const datasetPath = await job.getDataset(dataset, AccessLevel.ReadOnly);
        const XML = XmlDocument.open(datasetPath);
        const NSmap = XML.getDefaultNSMap();
        const evaluation = XML.evaluate(xpath, NSmap);
        result = evaluation as string;
        await job.log(LogLevel.Warning, xpath + " " + result as string);
```

```
    } catch (error: any) {
        result = error.message;
    }
    return result as string;
}
```

4.14. XmpDocument class

The XmpDocument class allows to find certain information in XMP documents.

This class does not allow modifying XMP contents.

4.14.1. Query language

The XMP data model expects a backing file that conforms to the Adobe XMP specification dated June 2005. The XMP backing file is parsed into an XMP-specific document object model. This XMP object model is then queried with an XMP location path (which is a small subset of XPath 1.0). The standard built-in aliases for XMP property names are automatically resolved.

Note: An XMP packet or file is a subset of RDF serialized to XML. Because a particular RDF construct (and thus an XMP property) can be serialized to XML in various ways, it is virtually impossible to correctly query an XMP file using generic XPath 1.0 expressions.

The query functions do not provide a separate argument to specify the top-level namespace (also called the "schema namespace" in XMP). This means that all property names in the XMP location path (including the top-level property name) must have a namespace prefix.

4.14.2. Methods

Note: All the methods in this class might throw an exception, so it is advised to wrap them into a try-catch block.

static open(path: string): XmpDocument;

Constructs an XmpDocument instance associated with a file specified through its absolute file path. The file should contain XMP information only, i.e. no path to a PDF or image file should be provided.

save(path: string): void;

Saves the XMP packet as a file specified through its absolute file path.

evaluate(xmpLocationPath: string, additionalPrefixMap?: { [name: string]: string; }): boolean | number | string | undefined;

Evaluates the XMP location path (refer to 'XMP location path syntax' in the Switch Reference Guide) against the XMP content and returns the result.

The result is the undefined object (as opposed to an empty string or a zero number) if the location path does not point to a leaf property (i.e. the property is not present or it is present but it is an array or a struct rather than a leaf). Otherwise a result of appropriate type will be returned:

- A boolean value if the string is "true", "false", "t" or "f", done with a case insensitive comparison.

- A number value if the string can be interpreted as a decimal number (with or without a decimal point) or as a rational number (two decimal integer numbers separated by a forward slash). For example, "1.25" and "5/4" represent the same number.
- A string value in all other cases.

Namespace prefixes in the XMP location path are resolved into namespace URIs using the prefix map provided to the function as a second argument. If the map argument is omitted or empty, the default prefix map is used for resolving prefixes. The default prefix map includes all mappings for the *standard XMP namespaces*, augmented with any extra mappings that occur in the XMP file.

Note: This call can (partly) be used to replace getLocalizedText (used in legacy scripting) by using an appropriate language selector in the XMP location path. For example, for an exact match with a specific language one can use a language selector such as [@xml:lang="en-US"] for US English, and to get an x-default item one can simply use [@xml:lang="x-default"].

Example

```
async function jobArrived(s: Switch, flowElement: FlowElement, job: Job) {
    try {
        const jobPath = await job.get(AccessLevel.ReadOnly);

        // For demonstration purposes, let's assume we'll only get PDF files.
        // We get the PDF document's XMP data, query it for the document title
        // and log the result in the Switch messages.

        let pdfFile = PdfDocument.open(jobPath);
        const xmpDocument = pdfFile.getXMP();
        pdfFile.close();

        let result = xmpDocument.evaluate('dc:title/*[1]');
        await job.log(LogLevel.Warning, 'XMP query to get the title of job %1
 returned: %2', [job.getName(), JSON.stringify(result)]);

        // Additional prefix mappings can be passed easily as well.

        result = xmpDocument.evaluate('custom:field', { custom: "http://
ns.yourdomain.com/custom/1.0" });

        let prefixMap = {
            'yourns1': "http://ns.yourdomain.com/yourns1/1.0/",
            'yourns2': "http://ns.yourdomain.com/yourns2/1.0/"
           };
        result = xmpDocument.evaluate('yourns1:thing/yourns2:field', prefixMap);

        await job.sendToSingle();
    } catch (error) {
        await job.fail("Failed to process the job '%1': %2.", [job.getName(),
 JSON.stringify(error)]);
  }
}
```

4.14.3. Standard XMP namespaces

The default prefix map includes mappings for the following standard XMP namespaces when using XmpDocument::evaluate() in Node.js based scripts.

Prefix	Namespace
AEScart	http://ns.adobe.com/aes/cart/
album	http://ns.adobe.com/album/1.0/

Prefix	Namespace
asf	http://ns.adobe.com/asf/1.0/
aux	http://ns.adobe.com/exif/1.0/aux/
bext	http://ns.adobe.com/bwf/bext/1.0/
bmsp	http://ns.adobe.com/StockPhoto/1.0/
creatorAtom	http://ns.adobe.com/creatorAtom/1.0/
crs	http://ns.adobe.com/camera-raw-settings/1.0/
dc	http://purl.org/dc/elements/1.1/
DICOM	http://ns.adobe.com/DICOM/
exif	http://ns.adobe.com/exif/1.0/
exifEX	http://cipa.jp/exif/1.0/
Iptc4xmpCore	http://iptc.org/std/Iptc4xmpCore/1.0/xmlns/
Iptc4xmpExt	http://iptc.org/std/Iptc4xmpExt/2008-02-29/
iX	http://ns.adobe.com/iX/1.0/
iXML	http://ns.adobe.com/ixml/1.0/
jpeg	http://ns.adobe.com/jpeg/1.0/
jp2k	http://ns.adobe.com/jp2k/1.0/
pdf	http://ns.adobe.com/pdf/1.3/
pdfaExtension	http://www.aiim.org/pdfa/ns/extension/
pdfaField	http://www.aiim.org/pdfa/ns/field#
pdfaid	http://www.aiim.org/pdfa/ns/id/
pdfaProperty	http://www.aiim.org/pdfa/ns/property#
pdfaSchema	http://www.aiim.org/pdfa/ns/schema#
pdfaType	http://www.aiim.org/pdfa/ns/type#
pdfx	http://ns.adobe.com/pdfx/1.3/
pdfxid	http://www.npes.org/pdfx/ns/id/
photoshop	http://ns.adobe.com/photoshop/1.0/
plus	http://ns.useplus.org/ldf/xmp/1.0/
png	http://ns.adobe.com/png/1.0/
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
riffinfo	http://ns.adobe.com/riff/info/
stDim	http://ns.adobe.com/xap/1.0/sType/Dimensions#
stEvt	http://ns.adobe.com/xap/1.0/sType/ResourceEvent#
stFnt	http://ns.adobe.com/xap/1.0/sType/Font#

Prefix	Namespace
stJob	http://ns.adobe.com/xap/1.0/sType/Job#
stMfs	http://ns.adobe.com/xap/1.0/sType/ManifestItem#
stRef	http://ns.adobe.com/xap/1.0/sType/ResourceRef#
stVer	http://ns.adobe.com/xap/1.0/sType/Version#
tiff	http://ns.adobe.com/tiff/1.0/
wav	http://ns.adobe.com/xmp/wav/1.0/
x	adobe:ns:meta/
xml	http://www.w3.org/XML/1998/namespace
xmp	http://ns.adobe.com/xap/1.0/
xmpBJ	http://ns.adobe.com/xap/1.0/bj/
xmpDM	http://ns.adobe.com/xmp/1.0/DynamicMedia/
xmpG	http://ns.adobe.com/xap/1.0/g/
xmpGImg	http://ns.adobe.com/xap/1.0/g/img/
xmpidq	http://ns.adobe.com/xmp/Identifier/qual/1.0/
xmpMM	http://ns.adobe.com/xap/1.0/mm/
xmpNote	http://ns.adobe.com/xmp/note/
xmpRights	http://ns.adobe.com/xap/1.0/rights/
xmpScript	http://ns.adobe.com/xmp/1.0/Script/
xmpT	http://ns.adobe.com/xap/1.0/t/
xmpTPg	http://ns.adobe.com/xap/1.0/t/pg/

4.15. Enumerations

4.15.1. AccessLevel

AccessLevel is an enumeration that defines access rights to the job content. It is used to define access rights when a user calls the *job::get* method.

Allowed values are:

- AccessLevel.ReadOnly
- AccessLevel.ReadWrite

Note: It is also available in global scope as EnfocusSwitch.AccessLevel if you need to use it outside of main.js.

4.15.2. Connection.Level

Connection.Level is an enumeration that defines the traffic light level of a Connection.

It's used to easily provide one of the following supported levels:

- Connection.Level.Error
- Connection.Level.Warning
- Connection.Level.Success.

Note: It is also available in global scope as EnfocusSwitch.Connection.Level if you need to use it outside of main.js.

Example:

```
await job.sendToLog(Connection.Level.Error, DatasetModel.XML, 'log_error.xml');
          await
 job.sendToLog(Connection.Level.Warning, DatasetModel.XML, 'log_warning.xml');
          await
 job.sendToLog(Connection.Level.Success, DatasetModel.XML, 'log_success.xml');

          await job.sendToData(Connection.Level.Error, 'data_error.txt');
          await job.sendToData(Connection.Level.Warning, 'data_warning.txt');
          await job.sendToData(Connection.Level.Success, 'data_success.txt');
```

4.15.3. DatasetModel

DatasetModel is an enumeration that defines the metadata data model. Possible values are:

- DatasetModel.Opaque
- DatasetModel.XML
- DatasetModel.XMP
- DatasetModel.JDF
- DatasetModel.JSON

Note: It is also available in global scope as EnfocusSwitch.DataModel if you need to use it outside of main.js.

4.15.4. LogLevel

LogLevel is an enumeration that represents supported levels of log messages. It is used to define the log level when a user calls the *Job::log* or the *FlowElement::log* method.

Allowed values are:

- LogLevel.Error
- LogLevel.Warning
- LogLevel.Info
- LogLevel.Debug

Note:

- It is also available in global scope as EnfocusSwitch.LogLevel if you need to use it outside of main.js.
- Debug messages are only displayed if the 'Log debug messages' preference in Switch Designer is enabled.

4.15.5. PropertyType

PropertyType is an enumeration that defines the property value type.

Possible values are:

- PropertyType.Literal
- PropertyType.Number
- PropertyType.Date
- PropertyType.HoursAndMinutes
- PropertyType.Boolean
- PropertyType.String
- PropertyType.FilePath
- PropertyType.FolderPath
- PropertyType.FileType
- PropertyType.FolderPattern
- PropertyType.Regex
- PropertyType.OAuthToken

Note: It is also available in global scope as EnfocusSwitch.PropertyType if you need to use it outside of main.js.

Editors

The property value type depends on the editor that the user used to enter the property value. Here is the correspondence between property value types and editors:

PropertyType	Editor
Literal	Literal
Number	Inline editor: Number
Date	Inline editor: Date (YYYY-MM-DD) Inline editor: Date and time (YYYY-MM-DDTHH:MM:SS) Both are supported by Date.parse()
HoursAndMinutes	Inline editor: Hours and minutes (HH:MM)
Boolean	Inline editor: No/Yes list Condition with variables
FilePath	Choose file
FolderPath	Choose folder
FileType	File type

PropertyType	Editor
	File types File patterns
FolderPattern	Folder patterns
Regex	Regular expression
OAuthToken	OAuth 2.0 authorization
String	Everything else

Remarks

- If an *Inline editor: String* was used, but the property allows for a *Choose file* or *Choose folder* editor, the property type is considered *FilePath* or *FolderPath* respectively.
- In case the *Literal editor* and literal editor name None was used, an empty string ("") will be returned as property string value. For all other literal editor names, the literal editor name will be returned as property string value.
- Note that *PropertyType.Number* is an integer (as it is not possible to define an editor in SwitchScripter that produces floating-point values for a script property).

4.15.6. Scope

Scope is an enumeration that represents the scope for global data.

It is used when the user manages global data and calls *Switch::getGlobalData*, *Switch::setGlobalData*, or *Switch::removeGlobalData*.

Possible values are:

- **Scope.FlowElement**: The data is read/written for this specific instance of the element in the flow. Other flow element instances are not able to access it regardless of whether they are using the same script or not.
- **Scope.FlowElements**: The data is read/written for all instances of this specific type of element in the same flow, i.e. all flow element instances using the same script in the same flow are able to read and write to it, while other types of elements using a different script are not able to access it.
- **Scope.Element**: The data is read/written for all instances of this specific type of element in all flows, i.e. all flow element instances using the same script are able to read and write to it, while other types of elements using a different script are not able to access it.
- **Scope.Flow**: The data is read/written for all instances of all elements in the same flow, i.e. all flow element instances in the flow are able to read and write to it, while flow element instances in the other flows are not able to access it.
- **Scope.Global**: The data is read/written in global space, in other words, any flow element instance using any script can access it.

Note: It is strongly advised to only use global scope when really required and to use Element or FlowElement scope where possible. When writing data in global scope it is the script programmer's responsibility to use distinct names as global data tags to prevent clashes with other scripts (including those written by other programmers)

Note: It is also available in global scope as EnfocusSwitch.Scope if you need to use it outside of main.js.

Overview:

Scope	Element types/instances	Flows
Scope.Global	All	All
Scope.Flow	All	Same flow
Scope.Element	Same element type	All
Scope.FlowElements	Same element type	Same flow
Scope.FlowElement	Same instance only	-

4.15.7. EnfocusSwitchPrivateDataTag

EnfocusSwitchPrivateDataTag is an enumeration that defines private data tags reserved for Switch built-in elements and can be read by the scripts.

For the detailed explanation of enumeration values, refer to *Private data used by built-in elements*.

Possible values are:

- EnfocusSwitchPrivateDataTag.emailAddresses
- EnfocusSwitchPrivateDataTag.emailBody
- EnfocusSwitchPrivateDataTag.hierarchy
- EnfocusSwitchPrivateDataTag.origin
- EnfocusSwitchPrivateDataTag.userEmail
- EnfocusSwitchPrivateDataTag.userFullName
- EnfocusSwitchPrivateDataTag.userName
- EnfocusSwitchPrivateDataTag.initiated
- EnfocusSwitchPrivateDataTag.submittedTo
- EnfocusSwitchPrivateDataTag.state

4.15.8. ImageDocument.ColorMode

ColorMode is an enumeration that defines the metadata data model.

Allowed values are:

- ImageDocument.ColorMode.Bitmap
- ImageDocument.ColorMode.Gray
- ImageDocument.ColorMode.IndexedColor
- ImageDocument.ColorMode.RGB
- ImageDocument.ColorMode.CMYK
- ImageDocument.ColorMode.Multichannel
- ImageDocument.ColorMode.Duotone
- ImageDocument.ColorMode.LabColor

- ImageDocument.ColorMode.Unknown

4.15.9. ImageDocument.ColorSpace

ColorSpace is an enumeration that defines the metadata data model.

Allowed values are:

- SRGB = 'sRGB',
- Uncalibrated = 'uncalibrated',

4.15.10. HttpRequest.Method

HttpRequest.Method is an enumeration that defines possible values for the HTTP method to subscribe on. It's used to easily provide one of the following values:

- HttpRequest.Method.POST
- HttpRequest.Method.PUT
- HttpRequest.Method.DELETE

Note:

It is also available in global scope as EnfocusSwitch.HttpRequest.Method, if you need to use it outside of main.js.

4.15.11. Priority

Priority is an enumeration that defines the priority of a job. See *Job priority* on page 100.

Possible values are:

- Low = -100000
- BelowNormal = -10000
- Normal = 0
- AboveNormal = 10000
- High = 100000

Note: These are predefined values that can be used to set the priority of a job, but **any number** can be used to the the job priority.

Example:

```
Myjob.setPriority(Priority.High);
```

5. Samples

For sample scripts, refer to: *https://github.com/EnfocusSW*.

6. Third-Party License Information

The third-party applications included in Switch are listed below. Additional information about third-party licenses can be found in the Resources > Licenses subfolder of the Switch installation folder.

```
This product includes lzw-ab.

    Copyright (c) David Bryant
    All rights reserved.

    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of Conifer Software nor the names of its contributors
      may be used to endorse or promote products derived from this software
      without specific prior written permission.

    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
    ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
    SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
    CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
    OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
    OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

------------------------------------------------------------

This product includes Botan.

      Copyright (C) 1999-2023 The Botan Authors
      All rights reserved.

      Redistribution and use in source and binary forms, with or without
      modification, are permitted provided that the following conditions are met:

      1. Redistributions of source code must retain the above copyright notice,
         this list of conditions, and the following disclaimer.

      2. Redistributions in binary form must reproduce the above copyright
         notice, this list of conditions, and the following disclaimer in the
         documentation and/or other materials provided with the distribution.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
      AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
      IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
      ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
      CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
      SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
      INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
      CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
      ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
      POSSIBILITY OF SUCH DAMAGE.

------------------------------------------------------------

This product includes ICC Profiles.

    Some ICC Profiles were created by FFEI Ltd. (www.ffei.co.uk) using Fujifilm
    ColourKit Profiler Suite (www.colourprofiling.com)
```

```
------------------------------------------------------------

This product includes ICC Profiles.

    Some ICC profiles are copyright (C) by European Color Initiative, www.eci.org

------------------------------------------------------------

This product includes ICC Profiles.

    Some ICC profiles are copyright (C) of WAN-IFRA, www.wan-ifra.org

------------------------------------------------------------

This product includes ICC Profiles.

    Some ICC profiles are copyright (C) IDEAlliance(R). G7(R), GRACol(R) and
    SWOP(R) are all registered trademarks of IDEAlliance(C).

------------------------------------------------------------

This product includes PANTONE Color Libraries.

    PANTONE® and other Pantone trademarks are the property of Pantone LLC.
    Pantone is a wholly owned subsidiary of X-Rite, Incorporated.

------------------------------------------------------------

This product includes curl.

   COPYRIGHT AND PERMISSION NOTICE

   Copyright (c) 1996 - 2023, Daniel Stenberg, <daniel@haxx.se>, and many
   contributors, see the THANKS file.

   All rights reserved.

   Permission to use, copy, modify, and distribute this software for any purpose
   with or without fee is hereby granted, provided that the above copyright
   notice and this permission notice appear in all copies.

   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN
   NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
   DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
   OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
   OR OTHER DEALINGS IN THE SOFTWARE.

   Except as contained in this notice, the name of a copyright holder shall not
   be used in advertising or otherwise to promote the sale, use or other dealings
   in this Software without prior written authorization of the copyright holder.

------------------------------------------------------------

This product includes LibTIFF.

      Copyright (c) 1988-1997 Sam Leffler
      Copyright (c) 1991-1997 Silicon Graphics, Inc.

      Permission to use, copy, modify, distribute, and sell this software and
      its documentation for any purpose is hereby granted without fee, provided
      that (i) the above copyright notices and this permission notice appear in
      all copies of the software and related documentation, and (ii) the names of
      Sam Leffler and Silicon Graphics may not be used in any advertising or
      publicity relating to the software without the specific, prior written
      permission of Sam Leffler and Silicon Graphics.

      THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
      EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
      WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

      IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR
      ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
      OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
      WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF
```

```
------------------------------------------------------------

This product includes JPEGLib.
This software is copyright (C) 1991-2016, Thomas G. Lane, Guido Vollbeding.
      All Rights Reserved.

      This software is based in part on the work of the Independent JPEG Group.
------------------------------------------------------------

This product includes Little CMS.

    Little CMS
    Copyright (c) 1998-2020 Marti Maria Saguer

    Permission is hereby granted, free of charge, to any person obtaining
    a copy of this software and associated documentation files (the
    "Software"), to deal in the Software without restriction, including
    without limitation the rights to use, copy, modify, merge, publish,
    distribute, sublicense, and/or sell copies of the Software, and to
    permit persons to whom the Software is furnished to do so, subject
    to the following conditions:

    The above copyright notice and this permission notice shall be
    included in all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
    EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
    IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
    CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
    TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
    SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

------------------------------------------------------------

This product includes libxml2.

    Copyright (C) 1998-2012 Daniel Veillard.  All Rights Reserved.

    Permission is hereby granted, free of charge, to any person obtaining a copy
    of this software and associated documentation files (the "Software"), to deal
    in the Software without restriction, including without limitation the rights
    to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the Software is fur-
    nished to do so, subject to the following conditions:

    The above copyright notice and this permission notice shall be included in
    all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FIT-
    NESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
    THE SOFTWARE.

------------------------------------------------------------

This product includes mongodb.

        Copyright (c) 2018 MongoDB, Inc.

        THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
        APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
        HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM тАЬAS ISтАЭ WITHOUT WARRANTY
        OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
        THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
        PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
        IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
        ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

------------------------------------------------------------

This product includes IP*Works!.

      Copyright (c) 2017 /n software inc. - All rights reserved.
```

```
 MAY NOT APPLY. EACH EXCLUSION OF LIMITATION IS INTENDED TO BE SEPARATE AND THEREFORE
 SEVERABLE.

------------------------------------------------------------

This product includes sqlite3.

Copyright (c) MapBox
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this
  list of conditions and the following disclaimer in the documentation and/or
  other materials provided with the distribution.
- Neither the name "MapBox" nor the names of its contributors may be
  used to endorse or promote products derived from this software without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

------------------------------------------------------------

This product includes javascript-obfuscator.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
  * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

------------------------------------------------------------

This product includes OpenJPEG.

        The copyright in this software is being made available under the 2-clauses
        BSD License, included below. This software may be subject to other third
        party and contributor rights, including patent rights, and no such rights
        are granted under this license.

        Copyright (c) 2002-2014, Universite catholique de Louvain (UCL), Belgium
        Copyright (c) 2002-2014, Professor Benoit Macq
        Copyright (c) 2003-2014, Antonin Descampe
        Copyright (c) 2003-2009, Francois-Olivier Devaux
        Copyright (c) 2005, Herve Drolon, FreeImage Team
        Copyright (c) 2002-2003, Yannick Verschueren
        Copyright (c) 2001-2003, David Janssens
        Copyright (c) 2011-2012, Centre National d'Etudes Spatiales (CNES), France
        Copyright (c) 2012, CS Systemes d'Information, France

        All rights reserved.
```

```
      DISCLAIMER

      This software is provided 'as is' with no explicit or implied warranties
      in respect of its properties, including, but not limited to, correctness
      and/or fitness for purpose.

---------------------------------------------------------------

This product includes QtActiveQt.

        Copyright (C) 2022 The Qt Company Ltd.

        Redistribution and use in source and binary forms, with or without
 modification,
        are permitted provided that the following conditions are met: *
 Redistributions of
        source code must retain the above copyright notice, this list of conditions
 and the
        following disclaimer. * Redistributions in binary form must reproduce the
 above
        copyright notice, this list of conditions and the following disclaimer in the
        documentation and/or other materials provided with the distribution. * Neither
 the
        name of The Qt Company Ltd nor the names of its contributors may be used to
 endorse
        or promote products derived from this software without specific prior written
        permission.

        THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 AND ANY
        EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 WARRANTIES
        OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
 EVENT
        SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
        INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 LIMITED
        TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 PROFITS; OR
        BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
 IN
        CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 ARISING IN
        ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 SUCH
        DAMAGE.

---------------------------------------------------------------

This product includes QtDeclarative.

      The Qt Toolkit is Copyright (C) 2019 The Qt Company Ltd.

---------------------------------------------------------------

This product includes QtShaderTools.

      The Qt Toolkit is Copyright (C) 2019 The Qt Company Ltd.

---------------------------------------------------------------

This product includes QtSingleApplication.

        Copyright (C) 2013 Digia Plc and/or its subsidiary(-ies).
        Contact: http://www.qt-project.org/legal

        This file is part of the Qt Solutions component.

        You may use this file under the terms of the BSD license as follows:

        Redistribution and use in source and binary forms, with or without
        modification, are permitted provided that the following conditions are
        met:
          * Redistributions of source code must retain the above copyright
            notice, this list of conditions and the following disclaimer.
          * Redistributions in binary form must reproduce the above copyright
            notice, this list of conditions and the following disclaimer in
```

```
3. This notice may not be removed or altered from any source distribution.
------------------------------------------------------------


------------------------------------------------------------------------------------------------------------------------
```

7. Copyrights