CYMIS v0.1 Specification

Contents

1 Overview

CYMIS (CYber sleuth Mod Installation Script) documents are json-based files that can generate customised mod versions from a set of input data. This is achieved by:

- Generating a set of boolean flags, intended to be done *via* an installation wizard;
- Generating the required mod files based on the values of the boolean flags.

The details of how a CYMIS document can be built to satisfy these requirements are given in section 2. This section focusses on the concepts the CYMIS format is built upon, rather than the specification details.

1.1 Components

A CYMIS implementation is made up of two components — a **wizard** and an **installer**. The job of the **wizard** is to generate a set of booleans. The job of the **installer** is to decide which files to include in the final mod build based on the values of those flags.

1.2 Flags and the Flag Table

A CYMIS implementation stores a list of defined flags in a **Flag Table**. These flags are stored under a name and are boolean-valued.
The two parts of a CYMIS implementation – the **wizard** and the **installer** – interact with the Flag Table in the following ways:

- The Wizard **creates** and **modifies the values** of flags;
- The Installer **reads** flags and **decides** what to do based on their values.

1.3 Important Files

There are three critical components to a CYMIS-ready mod:

- a “modfiles" folder,
- a “modoptions" folder,
- an INSTALL.json file.

The “modfiles" folder should be **empty**. It is the location in which the mod will be built.
The “modoptions" folder contains the data that the installer can install into the “modfiles" folder, based on the flags set in the CYMIS wizard.
The “INSTALL.json" file contains the CYMIS document. The information that must go into this file is defined in section 2.

2 Specification

2.1 Top Level

The top level of the CYMIS script must contain the following three labels:

- Version
- Wizard
- Install

“Version" states which CYMIS version the script should be interpreted as. If the script is written to satisfy the stipulations of this document, that version should be **0.1**.
“Wizard" is a list of pages that make up the installation wizard, specified in section 2.2. These pages contain boolean flags that are used to determine which mod files are used and/or generated.
“Install" is a list of possible routes the installation process could take, depending on which flags have been set in the wizard. This is specified in section 2.4.

The top level of a CYMIS document will therefore look like the following:

```
{
    "Version": 0.1
    "Wizard": [],
    "Install": []
}
```

2.2 Wizard Script

The Wizard is defined as a series of pages. Each page is defined by a dictionary containing the following items:

- Title
- Contents
- Flags

“Title" is the title of a page in the wizard.
“Contents" is the text that is displayed after the title, intended to describe what the current installer page does or what its purpose is. “Flags" is a list of flags that can be set on the page and will be displayed as UI elements. The definitions of these flags are given in section 2.3.

The contents of the “Wizard" item of the CYMIS document will therefore look like the following:

```
...
    "Wizard":
    [
        {
            "Title": "The First Page",
            "Contents": "This is the first page.",
            "Flags": []
        },
        {
            "Title": "The Second Page",
            "Contents": "This is the second page.",
            "Flags": []
        },
        ...
    ],
...
```

2.3 Flag Types

The CYMIS specification defines a number of flag types that can be used. Each flag type can itself potentially define more than a single flag, depending on the implementation. Each flag type must contain at a minimum a “Type" keyword, which defines which flag type it is, and some way of providing names for flags to be entered into the Flag Table.

2.3.1 Flag

The “Flag" flag type is intended to be implemented as a labelled checkbox, taking True or False values. The specification can contain the following items, with optional values in square brackets:

- Type
- Name
- Description
- [Default]

“Type" should be set to **“Flag"**.
“Name" is the name this flag is referred to by.
“Description" is the label attached to the checkbox.
“Default" defines whether the Flag begins in a True or False state. By default, it is False.

Some example flag definitions are given below.

```
{
    "Type": "Flag",
    "Name": "False Flag 1",
    "Description": "This Flag is False by default."
}
```

```
{
    "Type": "Flag",
    "Name": "False Flag 2",
    "Description": "This Flag is also False by default.",
    "Default": false
}
```

```
{
    "Type": "Flag",
    "Name": "True Flag 1",
    "Description": "This Flag is True by default.",
    "Default": true
}
```

2.3.2 HiddenFlag

The “HiddenFlag" flag type is not intended to be displayed in the UI. It should behave like a flag, but without the necessity for a “Description" item in the specification. The specification can contain the following items, with optional values in square brackets:

- Type
- Name
- [Default]

“Type" should be set to **“HiddenFlag"**.
“Name" is the name this flag is referred to by.
“Default" defines whether the Flag begins in a True or False state. By default, it is False.

Some example flag definitions are given below.

```
{
    "Type": "HiddenFlag",
    "Name": "False HiddenFlag 1",
}
```

```
{
    "Type": "HiddenFlag",
    "Name": "False HiddenFlag 2",
    "Default": false
}
```

```
{
    "Type": "Flag",
    "Name": "True HiddenFlag 1",
    "Default": true
}
```

2.3.3 ChooseOne

The “ChooseOne" flag type is intended to be displayed as a group of mutually exclusive checkable UI elements, such as a list of radio buttons. This flag type requires neither a Name nor Description, since it is a group of flags. However, the flags defined by this flag type should themselves each have a Name and Description. The specification can contain the following items, with optional values in square brackets:

- Type
- Flags
- [Default]

“Type" should be set to **“ChooseOne"**.
“Flags" is a list of flags provided by the ChooseOne group.
“Default" is the name of the flag that is selected at the start. By default, it is the first flag.

Each flag in the “Flags" item should contain the following items:

- Name
- Description

“Name" is the name this flag is referred to by.
“Description" is the label attached to the radio button.

```
{
    "Type": "ChooseOne",
    "Flags":
    [
        {
            "Name": "ExampleFlag1",
            "Description": "This is the first option."
        },
        {
            "Name": "ExampleFlag2",
            "Description": "This is the second option."
        },
        {
            "Name": "ExampleFlag3",
            "Description": "This is the third option."
        }
    ],
    "Default": "ExampleFlag2"
}
```

2.4 Install Script

The Install section of the CYMIS document describes how to build the data in the “modfiles" folder, given a set of possible build options and flags stating which paths should be followed. This follows a simple if-then format, where a single flag – or combinations thereof – are entered into the “if" section, and the build option is defined in the “then" section. Note that the “if" element is optional, and can be omitted if a certain build path should always be followed.

An “if" statement can be built from nested single-entry dictionaries that resolve down to flags and condition operations. The most basic “if" statement is a single flag:

```
"if": "Flag Name"
```

A more complicated “if" statement will contain a rule and an argument to be passed to the rule:

```
"if": {<rule name>: <rule argument>}
```

For example, checking if multiple flags are true:

```
"if": {"and": ["Flag 1", "Flag 2"]}
```

or if one flag is true and another is false:

```
"if": {"and": ["Flag 1", {"not": "Flag 2"}]}
```

Complex “if" statements can be built up in this manner. Any flag name can be replaced with a rule and argument in order to create arbitrarily deeply-nested conditions.

Build options, on the other hand, are simply a list of rules-and-arguments. The rules that can be used in a build option are detailed in section 2.6.

The contents of the “Install" item of the CYMIS document will therefore look like the following:

```
...
    "Install":
    [
        {
            "if": <condition>,
            "then": []
        },
        {
            "if": <condition>,
            "then": []
        },
        {
            "then": []
        },
        ...
    ],
...
```

2.5 Condition Rules

In the following section, the word “condition" is intended to mean either a **flag name** or a **condition rule plus its argument**, since both are valid values to pass to a condition rule.

2.5.1 and

The “and" rule takes a **list** of conditions and returns True if **all** the contained conditions are true.

```
"if": {"and": [<condition 1>, <condition 2>, <condition 3>, ...
    , <condition N>]}
```

2.5.2 or

The “or" rule takes a **list** of conditions and returns True if **any** the contained conditions are true.

```
"if": {"or": [<condition 1>, <condition 2>, <condition 3>, ...,
    <condition N>]}
```

2.5.3 not

The “not" rule takes a **single** condition and returns the opposite of the value of that condition.

```
"if": {"not": <condition>}
```

2.6 Installation Rules

Installation rules are individual entries in a “build path" that can be executed. They consist of a name for the rule, and a list of named arguments that get passed to the rule. Rules do not have a common set of arguments; each rule defines its own set. The only common factor between rule specifications is the “rule" keyword, which defines which rule is being selected.

2.6.1 copy

Copies an item from a location within the mod into the “modfiles" folder of a mod. The “copy" rule has two arguments:

- source
- destination

“source" is the item within the “modoptions" folder of a mod to be copied. It can be either a file or a directory. The path is separated by forward-slashes. “destination" is the location within the “modfiles" folder of a mod the source is to be copied to. It can be either a file or a directory. The path is separated by forward-slashes.

An example use of the rule to copy a file at “modoptions/scripts/common_scripts.txt" to “modfiles/script64/t3004.txt" is given below.

```
1 {"rule": "copy": "source": "scripts/common_scripts.txt", "
     destination": "script64/t3004.txt"}
```