Upkit Documentation

Release 0.4.x

Vu Le

Oct 26, 2018

Contents

CHAPTER 1

Introduction

1.1 What is Upkit?

Upkit is a command line toolkit that helps create/organize your Unity3D projects. With a simple configuration file, Upkit automatically resolves the project dependencies, symbolic-links them and generates a ready-to-use Unity project for you.

For those in a hurry, please go to Getting Started *to see* Upkit *in action.*

1.2 Why should you use it?

1.2.1 Our usecase

If you are like us, these are what you need when developing a Unity project:

- Total separation of 3rd party assets, plugins, dependencies from your assets/codes, to reduce the project size.
- Quick package swapping for prototyping and production.
- Simple dependency resolving, from Nuget or Git repositories, or elsewhere.
- Simple configuration.

1.2.2 Limitations of existing tools

At first glance, Upkit shares some similarities with Projeny, which is a great tool that we frequently used in our team. However, as Projeny model imposes a flat, exclusive package hierarchy, off-the-shelf packages do not often work well together. For example, two packages having the same native library folder `Plugins/Android` will clash. Even when there are no name clashes, Unity-compatible Nuget packages are not easily linked at times.

Unity 2018 officially comes with an easy-to-use built-in Package Manager. As of this writing, however, most of the Asset Store packages are still unavailable in the Package Manager, except those from Unity Technologies. Another

drawback with current Package Manager is that we cannot use it for internal cross-project packages. This means that most of the time, we have to fall back to traditional approaches.

1.2.3 Upkit remedies those issues and adds some more tricks

Upkit was initially designed as our solution to the aforementioned limitations, which is a tool sitting between Nuget (dependency resolving step) and Projeny (project linking step) in our pipeline. As our projects evolve, we decided to simplify the whole process by combining the two steps into Upkit, making it even easier to use by adding the following features:

- Single (YAML) file configuration, for dependency resolving, linking, etc.
- Link anything with *Linkspec* determining how folders, files are linked to your Unity project.
- Create distributable packages (with Linkspec).
- Out-of-the-box support for Nuget and Git dependencies.

1.3 Authors

- **Vu Le** - *Initial work* - FindersEyes

1.4 License

This project is licensed under the MIT License - see the LICENSE.md file for details

1.5 Acknowledgments

- This tool uses `xmltodict`, `pyyaml`, `yamlordereddictloader`, and `jinja2` under the hood. Thanks to the respected authors for the hard work.

CHAPTER 2

Getting started with Upkit

These instructions will use `upkit` to create a simple Unity3D project which depends on Newtonsoft.Json on Nuget Gallery.

The source code to this project can be also found under `examples/simple-app`.

2.1 Prerequisites

- Python 2.7 or above, with `pip`.
- (optional) `nuget` for resolving Nuget dependencies.
- (optional) `git` for resolving Git dependencies.

2.2 Installation

```
$ pip install upkit
```

2.3 Step 1: Create Upkit project

Creating a new Upkit project is as simple as:

```
$ upkit create-package simple-app
```

2.4 Step 2: Edit Upkit config file `upkit.yaml`

Upkit will create a new folder named `simple-app`, where you can find `upkit.yaml`. This file contains all the information Upkit needs in order to create your Unity project. Now, modify it to let Upkit know the project will depends on `NewtonSoft.Json`:

```
# upkit.yaml
params:
  project: '{{__dir__}}/project'

links:
  - target: '{{__assets__}}'
    source: '{{__dir__}}/assets'
    content: ['*']

  - target: '{{__plugins__}}'
    source: '{{__dir__}}/plugins'
    content: ['*']

  - target: '{{__project__}}/ProjectSettings'
    source: '{{__dir__}}/settings'

  - target: '{{__project__}}/Packages'
    source: '{{__dir__}}/packages'

  # Add project dependencies here:
  - source: 'nuget:Newtonsoft.Json@11.0.2#lib/net35'
    target: '{{__plugins__}}/Newtonsoft.Json'
```

Notice the second-last line where we instruct Upkit to resolve a Nuget library with `nuget:` scheme. Yes, it's that simple.

2.5 Step 3: Link to create Unity projects

The final step is to generate a Unity project, by calling:

```
$ cd simple-app
$ upkit link
```

Upkit will take a few seconds to resolve project's dependencies and generate a Unity project under `simple-app/project`. Open the folder in Unity as a project and you are ready to go.

CHAPTER 3

Link configuration and linkspec

3.1 Link configuration file

At the heart of each `link` operation is a YAML configuration file, which defines how symbolic links should be created to make a Unity project. By default, the file is named `upkit.yaml`, and can be changed to whatever needed. A typical configuration should look like this:

```
params:
  project: '{{__dir__}}/project'
  project_packages: '{{__dir__}}/packages'

links:
  - source: '{{project_packages}}/Scripts'
    target: '{{__assets__}}/Scripts'
```

3.1.1 Parameters

To make linking more configurable, there are two types of parameters supported by Upkit:

- **Built-in parameters:** those with name enclosed by __, for instance `__dir__` and `__assets__`, are generated by Upkit and shall not be defined. The list of built-in parameters are defined *here*.
- **User parameters:** those defined by user, under `params` section.

For each configuration file, there *MUST* be a `project` parameter defined, which tells Upkit where to generate the links.

Defining and expanding parameters

Defining a parameter can be as simple as:

```
params:
  some_param: 'some value'
```

However, most of the time, parameters are defined by expanding other existing parameters using Jinja syntax, such as:

```
params:
  platform: 'ios'
  project: '{{__dir__}}/project-{{platform}}'
  project_settings: '{{project}}/ProjectSettings'
```

Builtin parameters

The table below describes Upkit built-in parameters.

Note that only `__cwd__` and `__dir__` are accessible in `params` section.

Overwriting parameters

Given a configuration, its user-defined parameters can be overwriten at link-time by passing `-p param_name=param_value` to `link` command. This is particularly useful when you need to use the configuration file as a template for multiple Unity projects with slightly different parameters. For example, a configuration for multiple platforms may look like:

```
params:
  platform: 'ios'
  project: '{{__dir__}}/project-{{platform}}'
...
```

Execute the following commands:

```
$ upkit link
$ upkit link -p platform=android
$ upkit link -p platform=windows
```

will create `project-ios`, `project-android` and `project-windows` as separate Unity project folders.

3.1.2 Linkspec

To generate Unity projects, Upkit requires a list of link specifications, or *linkspec*s, which basically is a way to tell Upkit where to find a package, its content (all or partial) and a target to which the content is linked. A linkspec may be defined using properties in the table below:

Linkspec properties

`source` property

The `source` property describes a source package location containing files and folder to link. There are three types of source packages supported by Upkit:

- **Local file or folder**, when `source` takes the syntax `/path/to/local-file-or-folder`.
- **A Nuget package**, when `source` takes the syntax `nuget:(package_id)@(package_version)[#(sub_path)]`, in which:

  - `package_id` and `package_verion` are required.
  - `sub_path` is optional.
- **A Git repository**, when `source` takes the syntax `git:(repository_url)[@(branch_or_tag)][#(sub_path)]`, in which:
  - `repository_url` is required.
  - `branch_or_tag` and `sub_path` are optional.

When `source` refers to a Nuget package or a Git repository, Upkit first resolves the package or repository into a local folder under the container folder given by `-w` parameter (default to `.packages`), and then uses the resolved folder as a local source. If `sub_path` is given, the sub-path in the resolved folder is used as the local source instead.

Examples:

- `{{__dir__}}/Scripts`
- `nuget:NewtonSoft.Json@11.0.2`
- `nuget:NewtonSoft.Json@11.0.2#lib/net35`
- `git:https://github.com/finderseyes/upkit.git@develop`
- `git:https://github.com/finderseyes/upkit.git#examples/simple-app`
- `git:https://github.com/finderseyes/upkit.git@develop#examples/simple-app`

`content` property

By default, if `content` is not specified, Upkit treats a linkspec as *link-all* i.e. it will create one link from its `source` to its `target`. To patially link a source, declare its `content` as a list of glob patterns in the example below:

```
# include everything
content: ['*']

# include only files or folders under scripts/ and textures/.
content: ['scripts/*', 'textures/*']
```

When `content` is specified, Upkit will create multiple links, one for each item in the source content to an item with the same name under `target`. For example, given the following content items:

```
(source)/data/child/A/
(source)/data/B.txt
(source)/C.png
```

Upon linking, the following links will be created:

```
(target)/A/    --> (source)/data/child/A/
(target)/B.txt --> (source)/data/B.txt
(target)/C.png --> (source)/C.png
```

`exclude` property

`exclude` can be used in tandem with `content` to ignore some of the files and folders in a source content. `exclude` takes a list of patterns similar to `content`. For example:

```
content: ['*']
exclude: ['Document', 'Document.meta']
```

will include everything under a source package, except its `Document` and `Document.meta`.

`target` property

As the name implies, `target` is a local path defining where a source or its content should be linked to.

`links` property

A linkspec may use sub-links, under `links` property, when it needs to define multiple link targets. Each item in `links` is also a linkspec, except that it shall not have further sub-links i.e. `source`, `target`, `content`, and `exclude` are allowed but not `links`.

`links` is often used in package linkspec files as explained in *Package linkspec*. However, it can also be used to link packages where no linkspec file is provided, or to overwrite the linkspec of incompatible packages.

3.2 Package linkspec

A package linkspec file could be included in a package to make linking with it easier when shared across multiple projects. Given a package `A`, Upkit tries to look for its linkspec file in the following paths:

```
A/linkspec.yaml
A/linkspec.yml
A/content/linkspec.yaml
A/content/linkspec.yml
```

The format of linkspec file is also a linkspec as explained in *Linkspec* section.

In the case where a package `A` has a linkspec file, linking with it can be as simple as:

```
# upkit.yaml
links:
  - source: '/source/to/A'
```

3.2.1 `__source__` and `__target__` parameters

In linkspec files, two additional built-in parameters `__source__` and `__target__` are available, if specified.

3.2.2 Overwrite package linkspec

A package linkspec can be overwriten in a configuration file if one of these properties are defined:

- `content`
- `exclude`
- `links`

In such case, Upkit will ignore the package linkspec and use the one in link configuration. For example:

```
# upkit.yaml
links:
  - source: '/source/to/A'
    target: '/target/to-link'
    content: ['data/*']
```

In this case, Upkit links all items under `A/data` to `target`, regardless of whatever `A` linkspec file defines.

CHAPTER 4

Tutorials

4.1 Create a reusable package

Source code for this tutorial can be found in examples/shared-package

In this tutorial we will create a package `DemoPackage` which exports the following items when linked to project:

```
.
├── Assets/
│   └── DemoPackage/
│       └── Scenes/
└── Plugins/
    └── DemoPackage/
```

4.1.1 Step 1: Create `app` and `demo-package` project

Run the following commands to create the projects:

```
$ upkit create-package app
$ upkit create-package demo-package
```

You will notice the following structure each generated project:

```
.
├── assets/          -> project Assets content
├── packages/        -> Unity 2018 packages folder
├── plugins/         -> project Plugins content
├── project/         -> the generated project
├── settings/        -> project settings
├── linkspec.yaml    -> package linkspec
├── package.nuspec   -> predefined Nuspec file, if you want to build to a Nuget package
└── upkit.yaml       -> link configuration
```

Note we also use create `app` using `create-package` command, as it can also be shared to another project.

4.1.2 Step 2: Build demo-package

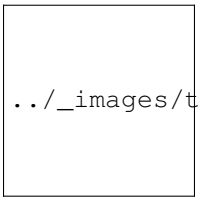
Let's assume that demo-package has a few scripts and a demo scene as an example to its users. Create the following folders in demo-packages:

```
• (demo-package)
├── assets/
│   └── Scenes/
└── plugins/
    └── DemoPackage/
```

Then link it

```
$ cd demo-package && upkit link
```

Open demo-package/project in Unity, you will see the project structure as:



Add a scene to the Assets/Scenes and create something fancy under Assets/Plugins/DemoPackage.

4.1.3 Step 3: Update demo-package linkspec

The next step is to edit the package linkspec so that others can use it. The default generated linkspec.yaml would suffice in most cases, we want to modify it so that the package demo Scenes will be linked under DemoPackage/Scenes the target project to avoid name conflicts.

Open demo-package/linkspec.yaml and modify its first link target from target: '{{__assets__}}' to target: '{{__assets__}}/DemoPackage'.

4.1.4 Step 4: Link demo-package with app

Linking with demo-package is as simple as adding it as a source in app/upkit.yaml:

```
# app/upkit.yaml
...
links:
  ...
  - source: '{{__dir__}}/../demo-package'
```

Finally, from app folder, run $ upkit link, then open the Unity project under app/project. That's it, the demo-package is linked to your app already.

4.2 Repackage an existing Unity package

Work in progress

CHAPTER 5

Upkit Commands

5.1 `link` command

Usage Resolves and links dependencies for a project with a given configuration.

Syntax

```
$ upkit link [-w PACKAGE_FOLDER] [-p PARAMS] [config]
```

Parameters

- `config` (optional, default to `upkit.yaml`) is the path to the link configuration file.
- `-w` (optional, default to `.packages`) is the path to the folder containing resolved Nuget and Git packages.
- `-p` (optional) defines a parameter to use when linking, and can be passed multiple times for multiple parameters, for example `upkit link -p a=1 -p b=2`.
  - If there is an existing parameter in the given configuration file, its value will be overwriten by the value in `-p` parameter.

5.2 `create-package` command

Usage Creates an empty package, which can be used as the boilerplate for a new Upkit project.

Syntax

```
$ upkit create-package [--link] location
```

Parameters

- `location` (required) is the name or location of an empty folder for the new package.
- `--link` (optional, default to `False`) to execute `link` command after the project is generated.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search