The NAP Release Packaging System

1. The v0.1 Focus

With v0.1 the focus has been on:

- Being able to create the release for three platforms
- Having a consistent feature set across all three platforms
- Making it easy for end users to..
    - Create projects
    - Create modules
    - Run demos
- Minimising SDK development interruptions

As a result the following haven’t been prioritised:

- Making it easy for people working at NAP source level, eg.:
    - Providing user tools (creating projects, modules, etc) there
    - Simplifying the workflow from creating a new module with third party requirements, through to packaged project
    - Prioritising source cleanliness over end-user usability
- Creating GUI user tools
- Allowing projects to be located outside the NAP release

2. How The Framework Packaging Process Works

Here’s I’ll attempt to roughly detail the process we go through to create a NAP framework release.

2.1. Working Directories

The packaging process outputs into a separate set of working directories which are `packaging_bin`, `packaging_build` and `packaging_lib`. The use of each is hopefully fairly self explanatory.

Separate paths are used for packaging output as it allows one to work on SDK work and do packaging work without having to rebuild everything each time. As you'll see below due to various defines influencing SDK behaviour the built libs used against NAP source differ from those used in a packaged framework release. Having separate output paths saves time and removes headaches (at the cost of disk space). The need for this may change in future releases.

The directory `packaging_staging` is used as a staging area into which the final output for the release is placed. It should only ever be seen when a packaging build is in progress. If a `packaging_staging` directory is found to already exist at the start of a new packaging run it's removed (regardless of whether a `--clean` is specified).

Note that on Linux `packaging_build` becomes `packaging_build_release` and `packaging_build_debug` due to the need for separate build directories per configuration.

2.2. The Python Wrapper

There is a Python wrapper around what is largely a CMake packaging process.

`package.py` performs the following:

1. Processes command line arguments
2. Creates the build timestamp
3. Retrieves the git revision
4. Ensures we're not going to overwrite an existing package
5. Cleans the build
6. Runs configure via CMake
7. Builds via `cmake` (Win64), `xcodebuild` (macOS) or `make` (Linux)
8. Finalises the release to a folder or a compressed archive, as requested

The configure and build steps (#6 and #7 above) are repeated for release and debug configurations.

For the sake of completeness, `package.py` currently takes the following command line arguments:

`--no-zip` / `-nz`
Don't zip the release, package to a directory

`--no-timestamp` / `-nt`
Don't include a timestamp in the created archive and folder name. For final releases.

`--clean` / `-c`
Do a clean packaging run

`--include-apps` / `-a`
Include Naivi apps (packaging them as projects)

The default behaviour of a packaging run is to:

- Include a timestamp in the archive/folder name
- Zip the release when it's completed
- Not include Naivi apps

CPack was investigated as an alternative to doing the archive compression directly in Python (in step #8 above). At the time it was determined to be simpler to do it with Python due to the fact that we run the build twice (once for each configuration). It would be possible to create a CPack-driven process but it would need to be clear that the extra work, which would be non-trivial, could be justified.

2.3. CMake Flags

The following CMake flags are populated by `package.py`:

`NAP_PACKAGED_BUILD`

This will always be set when a packaging build is being run and it indicates that a framework release is being created. As well as being made available to CMake it will also be defined in C++ via `packaginginfo.h` which is populated during the build.

This flag is currently used to:

- Set the different output directories for framework builds (as described in Working Directories above)
- Help Core locate project data, project.json, module.json, NAP modules and Python's modules in the different folder structures for NAP source and NAP release (see Project Runtime Folder Environments below)
- Not run unit tests for packaging builds (see Unit Test Issues below)

`BUILD_TIMESTAMP`
`BUILD_GIT_REVISION`

These populate the build timestamp and git revision into the build information file which is located at `cmake/build_info.json` within each release.

`PACKAGE_NAIVI_APPS`

Determines whether Naivi apps should be packaged into the release as projects. This is a convenience function that will be removed sometime after v0.1.

2.4. The Build

After the CMake configure a full build is performed using the `install` target. As of v0.1 this builds not only all libraries required for the release but all the projects. In fact the build is so straightforward that even if Naivi apps aren't requested to be included in the release they will still be built. Reducing build times hasn't been a focus with the work for v0.1. Stopping building these during packaging should be fairly trivial, see Disable Projects Building During Packaging under Looking Forward.

The `install` target is used as it's the install phase we use to deploy into the staging area for the release. Thus, for every piece of content included in the framework there's an associated call to CMake's install command somewhere throughout NAP's CMake. The built NAP modules for example are installed in the NAP CMake macro `package_module` (in `packaging_macros.cmake`), called from within each packaged module's `CMakeLists.txt`. Within `package_module`, `install(TARGETS …)` is used to include the module shared libraries into the release.

It's through the lack of a call to `install` that built content is excluded from the framework release.

2.5. Including Third Party Libraries

Third party libraries are deployed into the staging area using calls to CMake's install command. Those install calls are placed within the CMakeLists.txt file for the primary module or library using the third party dependency.

For example the packaging work for libartnet is down the bottom of `modules/artnet/CMakeLists.txt`, and the work for RTTR is down the bottom of `rtti/CMakeLists.txt`.

All third party libraries are deployed into the release under `thirdparty/`. The deployed contents for each library are cleaned up compared to their layout in the thirdparty git repository. This cleanup largely entails:

1. Only including libs for the single operating system
2. Removing operating system paths
3. Only including minimum docs
4. Including licenses wherever possible

2.6. Including The User Tools, CMake Modules, etc

The folder `dist` within the NAP repo contains content that exists purely* for distribution into the packaged release. The deployment of the contents of `dist/` into the release is done via the NAP CMake macro `package_nap` in `cmake/packaging_macros.cmake`.

Here's a rough breakdown of what sits within `dist/`:

`linux`, `macos` and `win64` contain the `check_build_environment` script used as an onboarding tool for new users to verify and setup their build environment, specific to each platform.

`win64` also contains:

- The package and regenerate shortcuts for within each project directory
- The regenerate shortcuts for within each module directory
- The MSVC redistributable help file which is deployed with each packaged project
- The user tools wrappers, which are placed within the root of the `tools` directory in the framework release

`unix` contains the same for macOS and Linux, excluding the MSVC redistributable help.

`cmake` contains all the CMake modules and shared macros which are deployed to `cmake/` in the release.

Finally, `user_scripts` contains the user tools scripts that are deployed into `tools/platform` in the release. The idea with the `tools/platform` directory is that these are tools that are used by the platform, vs. the top-level `tools/` directory which contains tools for the user.

* Although `dist/` should only contain content used within the framework release `user_scripts/platform/project_info_parse_to_cmake.py` and `dist/user_scripts/platform/module_info_parse_to_cmake.py` are used at the NAP source level to parse the `project.json` and `module.json` files into CMake module lists.

2.7. Deploying Headers and Demos

NAP library headers are deployed from a call to CMake's install command at the bottom of each lib's `CMakeLists.txt` file, eg. in `core/CMakeLists.txt` for core.

Demos are packaged into the release via the NAP CMake function `nap_source_project_packaging_and_shared_postprocessing`. This function allows for specification of which directory within the release should contain the demo (eg. for future separation of demos and examples and publishing Naivi apps as user projects) and specifying whether the project should only be packaged if Naivi apps are getting packaged. See the function documentation in `configure.cmake` for further info.

The call to `nap_source_project_packaging_and_shared_postprocessing` is made at the bottom of the `CMakeLists.txt` for each demo/project.

3. Project Runtime Folder Environments

Due to the focus on the packaging system during development for v0.1 efforts were made to not disturb the NAP source directory structure whilst creating a clean release structure for end users. As a result of that with v0.1 we have a situation where projects may run in one of three directory structures:

1. Within NAP source
2. Against a NAP framework release
3. As a packaged project

One of the more visible impacts of this is needing to support the different relationships between the project binary location and module locations on Unix, and between the project binaries and the service configuration and project information files (`config.json` and `project.json`) on all platforms.

For an example of this look at `modulemanager.cpp`.

This is a work in progress. Looking forward it's worth exploring the possibility of having similar directory structures for source and framework release, effectively combining #1 and #2 above. See more in Simplifying The Runtime Path Environments.

4. project/module.json Processing

The project information file `project.json` and the module information file `module.json` are populated into CMake from the JSON using Python.

Look at macro `module_json_to_cmake` in `configure.cmake` for an example of how this is done.

The process goes like this (for `module.json`):

1. CMake's configure_file is called on the input file to trigger CMake reconfigure when the input JSON file is changed (however see issues with this in Triggering CMake Reconfigure via project/module.json Updates)
2. `dist/user_scripts/platform/module_info_parse_to_cmake.py` (repo path) is called to parse the JSON file and create a small CMake file containing a list of the module dependencies
3. CMake's include is used to bring the list into CMake

5. Module Interdependency

One of the last additions to the build system pre v0.1 was the module interdependency system. The module interdependency system allows the user to specify only the top-level modules that they want to use in the project info file `project.json`. Prior to this change all modules required to run the project had to be specified. For example previously it was necessary to add `mod_napimgui` to the modules list in `project.json` because `mod_napapp`

uses it, however this behaviour is complicated for the user as they always need to know which modules each module depends on.

The module interdependency solves this via a module info file, `module.json`, for each module.

Here's a sample `module.json` file, for `mod_napapp`:

```
{
	"dependencies": [
		"mod_naprender",
		"mod_napsdlinput",
		"mod_napsdlwindow",
		"mod_napimgui",
		"mod_napinput",
		"mod_napscene",
		"mod_napmath"
	]
}
```

This module info JSON file is used in two contexts:

- By CMake in the build system to define modules for linking, header paths, packaging into projects, etc
- By the SDK to specify the full set of modules to be loaded and to build module search paths on Unix when running against a framework release

In the first context, working from CMake, `module.json` is loaded directly from root of the module directory. However when working from the SDK the module JSON files are loaded from duplicates of the main `module.json` file which are automatically deployed alongside any built NAP module. For example beside `mod_napapp.dll` the module information is contained in `mod_napapp.json`.

The reason for this was to keep things simple in the SDK implementation; loading a JSON file sitting alongside the module shared lib is cleaner than searching through different paths in the different Project Runtime Folder Environments. Basically this change erred on the side of SDK implementation simplicity over a cleaner filesystem structure.

See Module Interdependency Refinement in Looking Forward for notes on potential future updates.

6. User Tools

The following build tools are provided to the user in the directory `tools/` in the framework release:

`check_build_environment` - Checks and assists with setup of the build environment
No options are available.

`create_module NewModuleName` - Create a new module
Options:
`-ng, --no-generate` Don't generate the solution for the created module

`create_project NewProjectName` - Create a new project
Options:
`-ng, --no-generate` Don't generate the solution for the created project

`package_project ProjectName` - Package a project
Options:

`-ns, --no-show`	Don't show the generated package
`-nn, --no-napkin`	Don't include Napkin
`-nz, --no-zip`	Don't zip package

`regenerate_module ModuleName` - Call CMake reconfigure on the module
No options are available.

`regenerate_project ProjectName` - Call CMake reconfigure on the project
Options:
`-ns, --no-show` Don't show the generated solution

`package_project` and `regenerate_project` are also accessible from shortcuts in each project folder, and the same goes for `regenerate_module` for modules.

The user tools are not (yet?) available when working against the NAP source.

7. Project and Module Templates

`create_project` and `create_module`, described in the previous section, use CMake's configure_file to generate blank projects and modules from templates.

These templates are located in `dist/cmake/project_creator/template` and `dist/cmake/module_creator/template` (repo paths) and the following terms are available for substitution in their respective templates:

- `PROJECT_NAME_PASCALCASE`
- `PROJECT_NAME_LOWERCASE`
- `MODULE_NAME_PASCALCASE`
- `MODULE_NAME_LOWERCASE`

The creation from template process is basically:
1. Deploy each template file into output directory, processing for substitution
2. Deploy project/module directory tools shortcuts
3. Call CMake reconfigure (if enabled)

Default modules for a project are defined in the calling Python script (`dist/user_scripts/platform/create_project.py` - repo path). This is due to `create_project` originally optionally taking a list of modules which would be populated into the `project.json`. Restoring this functionality at some point may be useful for automated testing and other purposes.

8. Provisions for Custom CMake Working Against a Framework Release

At the benefit of simplifying things for the user (and at the risk of creating a brittle system) NAP v0.1 contains a fairly streamlined CMake setup for projects and modules operating against a framework release.

Effectively, all modules and projects use the CMake templates (not to be confused with the CMake project/module creation templates) located at `dist/cmake/nap_project.cmake` and `dist/cmake/nap_module.cmake` (repo paths). These templates are included by stub `CMakeLists.txt` files deployed in each module and project directory.

This mechanism allows us to:

- Encapsulate a lot of the complexity
- Update these templates later without need to modify user projects
- Re-use CMake logic

The risk, as mentioned above, with this approach is reducing build system flexibility for the end user. An initial modest approach at alleviating that has been made through the provision of hooks into blocks of CMake logic for each project and module via `project_extra.cmake` and `module_extra.cmake` files sitting in the project and module directories. The user documentation contains details about using these files.

The `module/project_extra.cmake` files are more of a proposal and it will be interesting what feedback we get back on them. In the long run I believe a more complex system will be needed but hopefully it can grow from what's here.

Also of note is that the `module_extra.cmake` files for the prebuilt NAP modules are deployed into the framework release and are included when when the prebuilt NAP modules are used in projects.

9. How The Project Packaging Process Works

The project packaging process is fairly similar to the framework release packaging process so I won't go into as much detail.

There is again a Python wrapper around what is largely a CMake packaging process. The Python process performs the following:

1. Processes command line arguments

2. Processes the project info file `project.json` for project name and version, used for the package file/folder name
3. Creates the build timestamp
4. Runs configure via CMake
5. Builds via `cmake` (Win64), `xcodebuild` (macOS) or `make` (Linux)
6. Includes the NAP framework release details into the project info file `project.json`
7. Finalises the package to a folder or a compressed archive, as requested
8. Shows the packaged release in file manager, as requested

As described in the framework release packaging notes it's CMake's install command that does the work of deploying content into the packaged project. In the case of the packaged project those `install` calls are contained in the following places:

- `dist/cmake/nap_project.cmake` (repo path) for the project binary, etc
- `module_extra.cmake` for each user module and project module deploys the module shared libraries, the third party shared libraries and the third party licenses

10. macOS and Linux RPATH Management

A reasonable amount of time during this development was spent on the shared library RPATH management on the Unices. By this I refer to manipulating the runtime search paths on binaries and shared libraries so that dependent shared libs can be found.

Typically a less heavy-handed approach might be more suitable than what's been built for the v0.1 release, however with the various runtime directory structures it was necessary to make direct changes to the search paths.

Primarily we want to be able to provide for these cases:

- Compiled NAP modules need to be able to find their shared libraries in their new relative location in third party
- `fbxconverter` needs to be able to find its NAP modules and requirements in third party
- Packaged projects need to search for libraries in `lib` next to the binary

The CMake wiki page relating to Unix RPATH management is worth a read if delving into some of this. Taking approaches more in line with CMake's approach was attempted and CMake's RPATH management is definitely utilised in some parts of the build system.

It's not going to be able to, or try to, cover everything relating to the NAP build system and RPATH management here.

10.1. Updating the ID Name of a macOS Shared Library

Also of note is that for macOS it's necessary to set all shared libraries' install names with a `@rpath/` prefix otherwise they won't even be found at runtime against NAP source. With a

lot of third party libraries in the thirdparty repo there are `NAP_NOTES.txt` files with describe how to do this.

But basically the main tools at your disposal are `otool` and `install_name_tool`.

`otool -D` will show you the shared library's id name:

```
$ otool -D lib/libout123.0.dylib
lib/libout123.0.dylib:
/Users/cheywood/workspace/naivi/thirdpartybuild/mpg123-1.25.6/install/osx/lib
/libout123.0.dylib
```

What we want is to replace that whole folder path with @rpath/ . We can do that using `install_name_tool -id`:

```
$ install_name_tool -id @rpath/libout123.0.dylib lib/libout123.0.dylib
$ otool -D lib/libmpg123.0.dylib
lib/libmpg123.0.dylib:
@rpath/libmpg123.dylib
```

11. Separated CMake Module Set

As noted in Including The User Tools, CMake Modules, etc above there is a separate set of CMake modules which are packaged into the framework release.

The reason for the duplication was:

- Allowing for the different third party paths in the platform release
- Providing CMake modules for the NAP libraries
- Reducing disruption at the NAP source level during development
- Simplicity in seeing which CMake module are being used where

See Reducing CMake Module Duplication under Looking Forward for some thoughts on improving this.

12. Napkin Specifics

12.1. Usage Within The Framework Release

Making Napkin accessible at within a NAP framework release requires solving three problems:

1. Ensuring all required modules are built for the project
2. Providing Napkin that matches the build configuration (release/debug) of those built modules
3. Ensuring that Napkin can locate all of those built modules

The solution provided with NAP v0.1 for these problems is to deploy Napkin into the binary output directory for any project once the project's compiled. This solves #1 above as the modules are built for the project. #2 is solved by deploying the correct Napkin built for the configuration of the project and finally #3 is solved by Napkin using the same module locating behaviour of the projects, which is made possible by Napkin being deployed in the same location as the project binaries.

12.2. Deployment Into Packaged Project

The majority of the work packaging Napkin into packaged projects is performed via `dist/cmake/install_napkin_with_project.cmake` (repo path). This file contains the logic for both deploying the project into the project binary output directory plus also into the packaged project.

The primary tasks performed are:

1. Deploying the binary
2. Deploying the resources
3. Deploying the main Qt libs
4. Deploying the Qt plugins
5. Deploying all required NAP plugins
6. Deploying Qt license/s
7. Unix: Making any required library runtime search path changes to allow shared libraries to be found

12.3. Qt Deployment

At the NAP source level `cmake/packaging_macros.cmake` contains a macro `package_qt` which is called from (weakly named) macro `package_nap` in the same file. `package_qt` contains some fairly hairy code which takes Qt from an official qt.io release (and only an official release - packaging system versions aren't supported) and deploys it into the framework release.

`package_qt` performs:

1. Package Qt licenses
2. Package each required Qt framework into the release. This set tends to vary slightly from platform to platform.
3. Package Qt platform plugins for the OS
4. Perform considerable tenuous RPATH management on Unix
5. Package additional required libraries (from Qt release) on Linux

At the framework release level all the logic for deploying Napkin and Qt alongside the project binary and into the packaged project are contained in `dist/cmake/install_napkin_with_project.cmake` (repo path). Similar tasks are performed to those above, and again some fairly heavy RPATH management is performed on Unix.

13. Python Specifics

With NAP v0.1 we're using Python packaged into thirdparty.

13.1. Packaging In Third-party Benefits

Packaging Python in `thirdparty` has the following benefits:

- Having a consistent Python version/build for SDK use
- Having a consistent Python version/build for user tools and build system
- Easing packaging Python into packaged projects
- Removing a dependency the user needs to manage

13.2. Usage

The packaged Python is used in these contexts:

- SDK runtime at NAP source level
- CMake at NAP source level
- SDK runtime at framework release level
- CMake at framework release level
- User tools at framework release level
- SDK runtime in packaged projects

There is only one use case where system Python may be used which is `check_build_environment` on Unix. In this case we don't make the assumption that the machine running `check_build_environment` is of the right architecture etc to run the script and use system Python instead. On Windows in the same context a batch script is used to do initial architecture checks before handing over to Python in thirdparty.

13.3. Provided Builds

As of v0.1 the Python builds in thirdparty are our own for macOS and Linux (see the NAP_NOTES.txt files for each) and on Windows we use a special embeddable zip provided directly by PSF. The embedded distribution is interesting as it's simpler and considerably smaller to deploy. Currently these packages are only available for Windows however in the future having a similar approach for Unix is fairly appealing. Some effort was put into creating similar zips for Unix but it didn't come to fruition.

13.4. Deployment

The `package_python` macro in `cmake/packaging_macros.cmake` is used to deploy Python into the framework release.

When working against a framework release Python is deployed on Windows via NAP CMake macro `win64_copy_python_dlls_postbuild` which is located in `dist_macros.cmake` and called from `naprtti.cmake`. On Unix Python remains located in `thirdparty`.

Python's library of modules are deployed on Windows via macro `win64_copy_python_dlls_postbuild` in `dist_macros.cmake`. This is typically called from called from `install_napkin_with_project.cmake` however if Napkin isn't to be included this logic instead is called from the top of `module_extra.cmake` for `mod_nappython`.

For Unix the deployment of Python's library of modules into a packaged project occurs in `module_extra.cmake` for `mod_nappython`. When running a project utilising `mod_nappython` against a framework release on Unix the Python library modules will be found in `thirdparty`.

Finally, when packaging a project, Python shared libraries are deployed on Unix via calls to CMake's install in `naprtti.cmake`. On Windows the DLL copying has already occurred as the post-build step described above.

14. Platform Differences

14.1. Packaging Third Party Libraries Into Packaged Projects

Due to the necessity to deploy all DLLs alongside the binary on Windows the copying of libraries is performed as a post-build step and as a result there is no need for extra packaging of third party libs via CMake's install command on Windows.

15. Unit Test Issues

The unit tests are currently disabled when doing a packaging build due to incompatibilities in the project runtime directory environments. When the unit tests are built they reside within the NAP source directory structure yet their path handing is built to run within a framework release. This incompatibility results in the unit tests failing if enabled.

See Running Unit Tests During Packaging under Looking Forward for possible solutions for this.

16. How To Integrate A New Third Party Requirement

I nearly re-wrote what's in the user docs on working with a third-party library, but let's instead use that as a base and add the additional pieces. i.e. read that link first.

The tasks we need to perform are the earlier stage of the process:

- Creating a CMake module that works against NAP source for the third party library
- Making our module `CMakeLists.txt` use the third party library at NAP source level

- Packaging the third party library into `thirdparty/` in the framework release
- Packaging the files described in the user docs into the framework release

Finally, one additional step we want to make is sourcing our library from `thirdparty/` instead of alongside the module. We'll look at that while deploying the files described in the user docs.

Creating a CMake Module for libfoo in NAP Source

Let's take Findfoo.cmake from the user docs and update it for NAP source. Create `cmake/Findfoo.cmake` with the following:

```
# Setup our fictional library paths
set(FOO_DIR ${THIRDPARTY_DIR}/libfoo)
if (WIN32)
  set(FOO_LIBS_DIR ${FOO_DIR}/msvc/bin)
  set(FOO_LIBS ${FOO_LIBS_DIR}/libfoo.lib)
  set(FOO_LIBS_DLL ${FOO_LIBS_DIR}/libfoo.dll)
elseif(APPLE)
  set(FOO_LIBS_DIR ${FOO_DIR}/osx/bin)
  set(FOO_LIBS ${FOO_LIBS_DIR}/libfoo.dylib)
  set(FOO_LIBS_DLL ${FOO_LIBS})
else()
  set(FOO_LIBS_DIR ${FOO_DIR}/linux/bin)
  set(FOO_LIBS ${FOO_LIBS_DIR}/libfoo.so)
  set(FOO_LIBS_DLL ${FOO_LIBS})
endif()

set(FOO_INCLUDE_DIRECTORIES ${FOO_DIR}/include)

# Hide from CMake GUI
mark_as_advanced(FOO_DIR)
mark_as_advanced(FOO_LIBS_DIR)

# Standard find package handling
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(foo REQUIRED_VARS FOO_DIR FOO_LIBS
FOO_INCLUDE_DIRECTORIES)

# Setup our shared import library
add_library(foo SHARED IMPORTED)

# Set shared library and include directory on our import library
set_target_properties(foo PROPERTIES
                  IMPORTED_CONFIGURATIONS "Debug;Release"
                  IMPORTED_LOCATION_RELEASE ${FOO_LIBS_DLL}
                  IMPORTED_LOCATION_DEBUG ${FOO_LIBS_DLL}
                  )

# Add Windows import library properties
```

```
if(WIN32)
      set_target_properties(foo PROPERTIES
                        IMPORTED_IMPLIB_RELEASE ${FOO_LIBS}
                        IMPORTED_IMPLIB_DEBUG ${FOO_LIBS}
                        )
endif()
```

The only changes here are the updated paths in the top section. As noted in 17.10. Reducing CMake Logic Duplication, in future reducing this approach of using duplicate CMake modules could be addressed.

Obviously we're only using one instance of libfoo here for both build configurations. Look at some our other CMake modules or examples of including separate builds for release and debug configurations.

Using The Third-Party Library From Our Module

Now let's add libfoo into our NAP module. In `CMakeLists.txt` for our module add the following:

```
find_package(foo REQUIRED)
target_link_libraries(${PROJECT_NAME} foo)
target_include_directories(${PROJECT_NAME} PUBLIC ${FOO_INCLUDE_DIRECTORIES})
```

Packaging The Third-Party Library Into The Framework Release

Add the following to the bottom of your module `CMakeLists.txt` to package the library into the framework release:

```
# Package libfoo into platform release
install(DIRECTORY ${FOO_LIBS_DIR}/ DESTINATION thirdparty/libfoo/bin)
# Package headers
install(DIRECTORY ${FOO_INCLUDE_DIRECTORIES}/ DESTINATION
thirdparty/libfoo/include)
```

If licenses are available for the third party library these should be installed into the third party release and then installed into the packaged project via further `install` calls in `module_extra.cmake`.

Packaging the Files From The User Docs Into The Framework Release

The files created during the user documentation on third party library deployment should be placed here within the NAP source:

`module_extra.json` should be placed in `modules/MODULENAME/dist/cmake`

`Findfoo.cmake` should be placed in `dist/cmake`

In addition to this, the paths of the CMake module created for use at NAP source level (`dist/cmake/Findfoo.cmake`) should be updated to refer to `${THIRDPARTY_DIR}` instead of a path alongside the NAP module source. The path alongside the NAP module source is used in the user documentation in an attempt to contain the dependencies with the module and allow for easier updating of the framework later when `thirdparty/` is replaced.

17. Looking Forward

Explored here are ideas for future improvements to the build/packaging system.

17.1. The Strength In Using CMake

It would have been possible to implement a lot of what we have in Python instead of CMake, and probably in less time.

The benefits in doing most of the work in CMake instead of Python include:

- CMake has been tuned for this specific task over a long time. It's good at the job.
- CMake makes it cleaner to integrate with other CMake projects
- Keeping things in CMake space allows us to change them into actions that are called at different phases of the build process
- Having things in CMake makes it easier to potentially later transition to projects (or even the framework) using installers
- This is an industry standard way to create build systems. It's possible to use C++ to create websites.. but it's not the best tool for the job.

Some of the downsides of CMake for this use:

- People need to learn the architecture of the built system
- It's a complex system, and like any other complex system there's a learning curve
- CMake is less familiar to people than Python

Finally, the part of the current system that is likely to be the most difficult maintenance issue, the Unix RPATH management, would be the same problem whether managed from Python or CMake.

17.2. Simplifying The Runtime Path Environments

As mentioned elsewhere, with v0.1 the focus was more on the framework release user and minimising disruptions than having a homogeneous path system between the NAP source and NAP framework release. Doing so may not even be the right approach but it's worth thinking about for the benefits of cleaning up some of the path managing work in the SDK.

One way to work towards this would be to output libraries and project binaries into path structures similar to the framework release structure. One downside of this would be the

added complexity of cleaning the build, but so long as we clean through the IDE this might not be too much of an issue.

If we can get past that, then the final hurdle would be getting used to the idea of (when working with NAP source) needing to rebuild projects on Windows to copy across any updated modules.

If we're on board with all of that we have the potential to:

- Remove the separate module path handing for NAP source vs release
- Remove the separate special NAP source path handling for `project.json` and `config.json`

17.3. Disable Projects Building During Packaging

Disabling building of projects during packaging should be as trivial as in each `CMakeLists.txt` not defining the targets if the packaging flag `NAP_PACKAGED_BUILD` is defined. Although there's probably a cleaner solution out there somewhere too.

17.4. Triggering CMake Reconfigure via project/module.json Updates

There is currently code in place to automatically trigger CMake reconfigure when `project.json` or `module.json` are updated. The obvious appeal in this is that the user can in theory change the module list in their project info JSON, hit build and all should be ready.

In practice however this is working as expected on Linux but on macOS and Windows although the reconfigure is triggered some of the new module changes don't appear to take effect until the next build.

Unsure if this is resolvable on our end or more of a CMake issue but fixing it would be a nice bonus for the user.

For the curious the only way at the moment to trigger this behaviour (of JSON changes triggering reconfigure) was to use CMake's configure_file to populate to a dummy file; a change to any input file to `configure_file` is added to their watch list.

17.5. Supporting Projects Outside The Framework Release Directory Structure

Supporting projects located outside the framework release directory structure is a very useful thing to have for benefits such as easing project source control management etc.

It's worth doing however it will be non-trivial to achieve.

The start of a rough list of required changes goes like:

- Updating `project.json` (or `config.json`?) to allow it to point to NAP root
- Using the NAP root parameter in the SDK to find NAP modules

- Helping the user tools shortcuts in the project directory find Python in `thirdparty`
- Helping CMake find NAP root for project templates etc
- Updating any CMake logic containing assumptions about relative project path locations to work from NAP root instead
- Updating regenerate and package user tools (when called from project directory shortcuts) to work cleanly when provided a directory to work with
- Updating regenerate and package user tools deployed under `tools/` to take directories as parameters (as the tools would no longer be able to find projects by name)

17.6. Simplifying 'Sandbox' Usage

Currently on init the SDK will only the load modules (and their dependencies) specified in `project.json`. This is a little limiting for people who want to experiment with all available modules as they have to list them all in the project module list to get what they want.

The API however provides for this use case (via a parameter to `ModuleManager::loadModules`), and Napkin currently makes use of it. Exposing this via a command line parameter and/or a property in the project info JSON would make this kind of sandbox usage easier.

17.7. User Tools At Source Level

There's no real reason why we can't create a set of user tools that work at NAP source level, or update and integrate the existing set so that they'll work. Maybe we want to ask first if we want to though? Are we encouraging people to develop against a platform release or NAP source?

The problem is also slightly different as the user tools provided with a framework release are somewhat specific to the operating system that the framework release is for. Working against NAP source no assumptions can be made and the OS has to be determined at runtime.

`create_project` and `create_module` are probably the most useful. They'd need to use a slightly different template for creating the build system part of their contents. `check_build_environment` is also tempting and should encompass Qt.

Questions remain:

- Do we want to do it / is it worth it?
- Where would they sit in the repo?
- Is it updating and extending the existing tools or a duplicate set?

17.8. Module Interdependency Refinement

As described above the module interdependency system currently works from a `module.json` file which sits in the module directory (for CMake) plus is also deployed alongside the built library (for the SDK).

This results in two extra copies of `module.json`, one for both build configurations. As there isn't any need for modifying the module dependency content after build time it's tempting to look at a solution where CMake automatically populates the module dependency information for each module into something (a header?) that could be brought in at build time.

This would remove the duplicate files and should be reasonably easy to implement using eg. CMake's configure_file and our existing logic for parsing `module.json` into CMake.

17.9. Running Unit Tests During Packaging Builds

As described above the unit tests are currently disabled during packaging builds as a result of the incompatibility between the path management in the core library of the built SDK and the layout of the NAP source repo.

Two possible solutions that immediately come to mind for this are:

- Deploying the unit tests into the staging area and running them from Python after the build has been completed. If they complete successfully then they'd be removed from the staging area before archiving the release.
- Ending up with a folder structure in NAP source that mimics the framework release. Somewhat covered above.

17.10. Reducing CMake Logic Duplication

Steps to reduce duplication of CMake logic include

- Sharing CMake modules for third party libraries between both NAP source and framework release contexts by adding search paths for each of them into the single set of modules. Some updating to the users of those modules is likely needed.
- Build a package of CMake macros and functions that are shared between both NAP source and framework release
- Look for other improvements to be gained through simplifying the runtime path environments

17.11. Sharable User Modules

This is an exciting one to start thinking about. It wouldn't be hugely difficult to get something off the ground.

Things to consider

- Supporting sharing of modules as source or binary
- Supporting binary modules that only support some platforms, dealing with that cleanly

- Supporting third-party dependencies packaged alongside modules
- Support for module versions, and other meta-data
- Ensuring the module API version is sufficiently visible
- A process for users showing..
    - How to build the module on multiple platforms
    - How to include third party dependencies into the package
    - How to bundle binary modules built across multiple platforms into a single package
    - How to deploy third party dependencies from shared modules into packaged apps