Package Creation Workflow

This document will discuss workflow.

Rationale

The logical approach to designing the workflow can be derived from the structural components of the DPM Package Structure specification and where human interaction is required with each piece.

The `metadata` directory is generated first and then requires human review and modification in some areas — so it will need a phase.

The `contents` directory is generated by the compilation of your software or is the data you want to deploy, and so is the core deploy payload of the package. It already exists when the package is decided to be built since DPM assumes the software is already compiled, so a stage that builds the directory structure of the package would cover the creation of the contents component — it requires no further modification and can simply be recursively copied from the directory structure you've created that you want your files deployed in.

The `hooks` directory is entirely created by the end user — it can exist before package creation but would be populated with empty files without that for the user to have something to work with — and requires review after being put in place — so an `optional` phase needs dedicated to hooks — for generating a blank if nothing is provided — or for populating a hooks directory in the open package directory structure from a path provided

The signing directory is a product of `optional` signing after the package is built. It can be signed after all the other pieces are in place, or not at all, and can even be signed after all the archives are compressed and the package is sealed. So, a DPM module can sign it after contents/hooks/metadata is updated but nothing is compressed yet, or it can be signed after the package is done — though this `should` be done at the time of package creation.

So, to recap:

1. `contents`: already exists when packages are created or can be filled in by the user after the package structure is created at the “stage” output directory
2. `hooks`: can be introduced if they exist at the time of “stage” or can be created blank if not, and even can be replaced prior to the metadata generation
3. `metadata`: can be generated after contents and hooks are in place but must be modified by the user in key areas (CONTENTS_MANIFEST_DIGEST) prior to sealing the package.
4. `signatures`: this optionally takes place before or after sealing the package using a dedicated signing module.

DPM Package Creation Workflow

1. Build a Package Stage

So, a “stage” command to the DPM “build”module (“dpm build stage”):

- creates an empty package directory using the name, version, architecture, os options supplied at the output directory (–output) location specified
- stages the directory supplied as –contents to $(–output)/contents
- either stages the hooks directory supplied as –hooks to $(–output)/hooks or generates a hooks directory with empty files for the triggers if not supplied
- generates the initial metadata according to spec.

2. Update Metadata

After the user modifies the contents_manifest_digest file to indicate controlled/not-controlled files or to change ownership or permissions, populates PROVIDES/REPLACES/everything else, the package is ready to be optionally signed and non-optionally sealed into a finalized package.

Optionally Sign the Stage

Then, an optional “sign” command to the DPM sign module (“dpm sign stage”).

This updates the metadata prior to sealing it. Any tampering after this point will be immediately identifiable during verification of the package using one of the `verify` modules.

3. Seal the Stage into a Package

Then, a seal command (“dpm seal package”), to finalize the package:

The user will do a “seal package” which generates the PACKAGE_DIGEST if not already signed, the same for HOOKS_DIGEST, gzips the contents, hooks, metadata contents, and then gzips the package directory. This is now a package.

Optionally Sign the Package

Then, a second opportunity to “sign” the package will exist using the DPM sign module (“dpm sign package”) which opens it back up, signs its relevant pieces, regenerates the metadata, and seals it back up. Functionally, there’s difference in the resulting package between one signed at the staging phase or one signed after the seal phase — even if signing during the staging phase is more efficient, such as when generating packages from a build system.

On Future Modules

This is different and therefore “bad”.

Yes, everything different than you’re used to is “bad” — especially if you’re not looking at what problems the change solves.

This is more hands on than, say, “slackpkg”.

There will be a “wizard” module that reads a file very similar to what you’d expect in, say, an RPM spec file for rpm-based distributions and does most of the work and accommodates automated bulk package build processes.

This doesn’t compile the software. What compiles the software?

DPM assumes you already did that and staged what you want deployed in your package in the directory structure you want it deployed in.

However, there will be future support for an “sdpm” package type that is the raw source code of what you want to build, a hopefully very simple build script, various patches that need applied, a CHANGELOG etc to encapsulate to the compilation process in the package manager. Truthfully, this is better off not existing, as it creates more problems than it solves and introduces the responsibility of software compilation to the package manager, overloading its purpose. The core purpose of a package manager is to manage the lifecycle of files on the server, so this is not viewed as part of the critical path to DPM release.