Creating and Running Your Own Homebrew Tap

I love Homebrew, I really do. But some of the limitations of the system, while understandable from a maintenance point of view, can be really frutrating.

In How to Create and Maintain a Tap the Homebrew team shows how to create a tap and how to add formulae and casks to it.

At first I though it would be easy; just use `brew tap-new` to create the custom tap, create the corresponding repository, create the formulas that I want to use and push them to the repo.

But it's not that easy.

The post will discuss how to create a custom tap, two ways to add packages to a tap, how to install Node packages using Homebrew, how to install Python modules for the Homebrew-installed Python environment and the pitfalls I found along the way.

Creating a Tap

How to Create and Maintain a Tap outlines the process for creating and maintaining a tap.

Creaating a new tap is deceptively simple. Just run the following command:

```
brew tap-new caraya/homebrew-rivendellweb
```

Even here there's a little deception. We use `homebrew-rivendellweb` as the name of the tap so users can use the shortened name when 'tapping' the tap or adding the tap to a user's Homebrew installation.

```
brew tap caraya/rivendellweb
```

Homebrew will use the full `caraya/homebrew-rivendellweb` name under the

hood.

There are three possible use cases I can think of for adding a package that is not Node or Python to a tap:

- I want to add a brand new package for our own use without going through approval for inclusion in homebrew/core (either because I don't expect wide usage or because I just want it for my own use)
- I want to modify an existing package that is already in homebrew/core but I don't think the modification will be useful for other users (or it would have been done already)
- I'm testing a package before deciding if I want to submit it to homebrew/core for inclusion

Adding Software to Homebrew givevs a good introduction to the process of creating a Homebrew formula that is applicable to all taps, just not Homebrw core.

The Formula Cookbook provides more detailed information about creating a formula. This is applicable to all taps.

Adding A Brand New Package To A Tap

The easiest way to add a package is to start with an application that is not part of core and is available as a zip or tar ball and the custom tap that we want to add the formula to.

The command is simple:

```
brew create --tap caraya/rivendellweb \
https://example.com/foo-0.1.tar.gz
```

This will download the tarball, extract it to a temporary directory and create a formula in the specified tap directory.

You will get the skeleton of a Formula in the homebrew/core directory. This is important as you'll have to move the formula to your tap's directory.

```
class Foo < Formula
  desc ""
  homepage ""
  url "https://example.com""oo-0.1.tar.gz"
  sha256 "85cc828a96735bdafcf29eb6291ca91bac846579bcef7308536e0c875d6c81d
  lic"85cc828a96735bdafcf29eb6291ca91bac846579bcef7308536e0c875d6c81d7"# 
                          "--disable-silent-rules",
                          "--prefix=#{prefix}"
    # system "cmake", ".", *std_cmake_args
    s"--disable-dependency-tracking"#{prefix}"
    # system "cmake", ".", *std_cmake_args
    system "make", "install"
  end

  test do
    system "false"
  end
end
```

This is where it gets interesting. The default is seldom enough to make the formula work and it will definitely change depending on the package.

The first step is to change the `desc`, `url` and `license` fields.

For the rest of the formula, you will likely need to look at the Formula Cookbook for more information about the commands available for creating and editing a formula.

The final step is to verify that the tap is free of errors and will work as intended.

The command to audit the formula is:

```
brew audit --strict\
  --new-formula \
  --online <formula>
```

Once it is free of errors, you can push it to commit and push it to your Tap

repository from a branch. The commands to do this are:

```
cd "$(brew --repository repository/name)"
git checkout -b <some-descriptive-name>
git add Formula/foo.rb
git commit
```

When you run the commit command without the `-m` flag you will be taken to your default editor to write the message. The established standard for Git commit messages is:

- A commit summary of 50 characters or less
- Two (2) newlines
- A thorough explanation of the commit (can be as long as it needs to)

This is a personal tap but I like to keep the same discipline I need to have if I ever decide to commit to homebrew.

See Tim Pope's A Note About Git Commit Messages for more information about commit messages. Even though we're not working with email commits, we should stick to the same set of rules.

Because the branch you're comitting to doesn't exist on the remote server you'll have to do something like this the first time you push your changes:

```
git push --set-upstream origin <branch name>
```

This will create the remote branch and matching with the branch of the pull request you just created.

Once you've committed the formula to the repository and pushed it, you can create a pull request for it.

The pull request will run tests and create the bottles for the formula but won't publish them. Once the tests are completed add the `pr-pull` label to the PR. This will run an additional action to publish the bottles.

This has proven the hardest part of the process. The errors are cryptic if you don't understand the way Homebrew works (and I don't) and they are not intuitive

to fix.

Once you have resolved all the issues the pull request will be complete and bottles for the fomula will be available for download for people who use the tap.

Adding Modified Versions Of Packages Already in Homebrew/core To A Tap

What prompted me to look at custom local taps is Graphics Magick and it's Perl support.

I want to install the PerlMagick package that works with GraphicsMagick. The Package is available on Homebrew Core, so just instqalling the package from core should fix the problem, right?

It should, but it doesn't.

The formula in homebrew/core doesn't include the Perl module dependency. There is no way to add the `--with-perl` option to the formula at install time and there is no way to install the Perl module other than doing it at the same time as the application.

I'm pretty sure that any PR to add the Perl module to the formula in homebrew/core will be rejected as having low usage and a even lower interest (or it would have been done already). So the options are: compile from source or modify the formula and put it in a personal tap. I chose to do the later.

So now onto the process.

Because the formula must be unique across all taps, including homebrew/core, we need to create a new formula. For this example, I'll pick `graphicsmagick-perl` since it indicates the name of the file and what I'm adding to it.

To make sure that the formula gets created in the right tap, run the following command:

```
brew create --tap caraya/rivendellweb \
<url to the source of the package>
```

You will get the skeleton of a Formula in the tap directory spcify.

Rather than crreating a brand new formula I'll use the formula for Graphicsmagick available on homebrew/core and adds the extra bits I need. In the example below, I added Perl as a dependency, `depends_on: perl`, and `--with-perl` to the configuration options array.

I spent about a week troubleshooting issues with the formula. Issues ended when I removed dependencies for a given image format (jbig) and it worked properly.

The rest of the process is the same as for a brand new formula.

Adding Node Packages

Creating Node Formulae for Homebrew is a little bit of a challenge. They will only work with Node versions that are available in Homebrew. If you use NVM to manage your Node installations, you won't be able to use Homebrew's Node formulas so be aware of this if you decide.

Node for forumula authors provides a detailed guide to creating Node formulas for Homebrew.

Running npm install

Homebrew provides two helper methods in a `Language::Node module`. Before you can use them you have to require the Node language module at the beginning of your formula file with:

```
require "language/node"
```

The two methods are:

- ***std_npm_install_args*** for formuls for compatible with npm's global module format (like azure-cli or webpack)

- ***local_npm_install_args*** for formulas where the npm install call is not the only required install step or need to also compile non-JavaScript sources (like elixirscript or grunt-cli)

```
system "npm", "install", *Language::Node.std_npm_install_args(libexec)
```

Node modules should be installed to libexec. This prevents the Node modules from contaminating the global node_modules, which is important so that npm doesn't try to manage Homebrew-installed Node modules.

Download URL

If the Node module is also available on the npm registry, use npm hosted release tarballs over other hosted source tarballs. NPM tarballs don't include the ignored files such as tests, resulting in a smaller download size, and any necessary transpilation step has already been completed.

The npm registry URLs usually have the format of:

```
https://registry.npmjs.org/<name>/-/<name>-<version>.tgz
```

Dependencies

Node modules which are compatible with the latest Node version should declare a dependency on the node formula.

```
depends_on "node"
```

If your formula requires being executed with an older Node version you should use one of the versioned node formulae (e.g. node@12).

```
depends_on "node@12"
```

Special requirements for native addons

If your Node module or any of its dependencies is a native addon you have to

declare an additional dependency. The compilation of the native addon results in an invocation of node-gyp so we need an additional build time dependency on “python” (because GYP depends on Python).

```
depends_on "python" => :build
```

Formulas with a native addon dependency will only work with the major version (12, 14, 16) it was compiled with. This means that you need to revision every formula with a Node native addon with every major version bump of the node formula. To help with this, write a meaningful test which would fail in such a case (invoked with an ABI-incompatible Node version).

Adding Python Modules

Python presents a similar set of problems to Node but slightly more complex. Theredefault version of Python available on macOS Big Sur is 2.7.16. Furhermore, in the release notes for macOS 10.15 (Catalina), Apple states the following:

> Scripting language runtimes such as Python, Ruby, and Perl are included in macOS for compatibility with legacy software. Future versions of macOS won’t include scripting language runtimes by default, and might require you to install additional packages. If your software depends on scripting languages, it’s recommended that you bundle the runtime within the app. (49764202)
>
> Use of Python 2.7 isn’t recommended as this version is included in macOS for compatibility with legacy software. Future versions of macOS won’t include Python 2.7. Instead, it’s recommended that you run python3 from within Terminal. (51097165)

We will not talk about what it would take for users to migrate their code from Python 2 to Python 3. Instead we’ll look at building Python3-based formulas for Homebrew.

Python for Formula Authors points out some of the issues of working with Python in Homebrew.

Python declarations

Formulas for apps that require Python 3 must declare an unconditional dependency on `python@3.x`. These apps must work with the current Homebrew Python 3.x formula.

Applications that work with Python 2 should use the Apple-provided system Python in /usr/bin on systems that provide Python 2.7. No explicit Python dependency is needed since /usr/bin is always in PATH for Homebrew formulae.

Installation Notes

You should install your Python pplications into a Python virtualenv environment rooted in libexec. This prevents the app's Python modules from contaminating the system site-packages and vice versa.

Declare all Python module dependencies and their dependencies, recursively as resources in the formula and install them into the virtualenv, as well. Don't rely on `setup.py` or `pip` to perform automatic dependency resolution Because Homebrew doesn't like install scripts that are pulling from the master branch of Git repositories or unversioned, unchecksummed tarballs. Specify dependencies explicitly in the formula using `resources` stanzas.

Use `brew update-python-resources` to help you write resource stanzas. To use it, simply run `brew update-python-resources <formula>`. Sometimes, `brew update-python-resources` won't be able to automatically update the resources. If this happens, try running `brew update-python-resources --print-only <formula>` to print the resource stanzas instead of applying the changes directly to the file. You can then copy and paste resources as needed.

If the brew native method doesn't work, you can use homebrew-pypi-poet to help writing resource stanzas. You can do it like this:

```
# Use a temporary directory for
# the virtual environment
cd "$(mktemp -d)"

$(mktemp -d)"$(mktemp -d)"# Create and source a new virtual
# environment in the venv/ directory
```

```
python3 -m venv venv
source venv/bin/activate

# Install the package of interest as
# well as homebrew-pypi-poet
pip install some_package homebrew-pypi-poet
poet some_package

# Destroy the virtual environment
deactivate
rm -rf venv
```

Homebrew provides helper methods for instantiating and populating virtualenvs. You can use them by putting the following statement at the top of the formula definition:

```
include Language::Python::Virtualenv
```

For most applications, all you will need to write is:

```
def install
  virtualenv_install_with_resources
end
```

For more information, check Python for Formula Authors, Homebrew's documentation on virtualenvs, Homebrew's documentation on resources, and the Homebrew Python Language API.

Creating formulas for other languages

In this post I covered the basics of creating formulas for the three languages that I expect to use the most, autotools (the standard ./configure, make and make install method), Node.js (and the disadvantages to doing so), and Python.

Homebrew can provide templates for creating formulas for other languages and installation systems.

The following is an abbreviated list of the output from the brew create command listing only the languages and build methods supported out of the box.

```
Usage: brew create [options] URL

--autotools         Create a basic template for an
                    Autotools-style build.
--cask              Create a basic template for a cask.
--cmake             Create a basic template for a CMake-style
                    build.
--crystal           Create a basic template for a Crystal
                    build.
--go                Create a basic template for a Go build.
--meson             Create a basic template for a Meson-style
                    build.
--node              Create a basic template for a Node build.
--perl              Create a basic template for a Perl build.
--python            Create a basic template for a Python build.
--ruby              Create a basic template for a Ruby build.
--rust              Create a basic template for a Rust build.
```

The ones I'm most intrigued by are `--go` and `--rust`. I may want to explore them in more depth in the future, along with a more detailed explanation of how I built my customized GraphicsMagick formula.