Creating OPAM packages

An OPAM package is basically just a bunch of data on a software project: * A name, version and description * Some dependencies and constraints on other packages, compiler versions, etc. * Build, install and remove instructions * Some additional information (bugtracker, homepage, license, doc...)

This document will go through a simple way to get it in the right format, whether you are packaging your own project or someone else's. It's not a complete guide to the opam file format.

Creating a local package

We'll be assuming that you have a working OPAM installation. If not, please first read the quick install guide.

Get the source

Let's start from the root directory of your project source, typically obtained from a version-controlled repository or from a tarball.

```
$ wget https://.../project.tar.gz && tar xvzf project.tar.gz
# Or
$ git clone git://.../project.git
...
$ cd project
```

Now let's create an `opam` subdirectory that will hold the local OPAM metadata. It's a good idea to keep it with the package source to ease working with development versions – the OPAM repository will hold it for releases.

```
$ mkdir opam
```

Opam pin

OPAM 1.2 provides a feature that allows you to register packages locally, without the need for them to exist in a repository. We call this *pinning*, because it is an extension of the very common feature that allows to *pin* a package to a specific version number in other package managers.

So let's create a package *pinned* to the current directory. We just need to choose a name and issue the command:

```
$ opam pin add <project> .
```

The opam file

At this stage, OPAM will be looking for metadata for the package `<project>`, on its repository and in the source directory. Not finding any, it will open an editor with a pre-filled template for your package's `opam` file. It's best to keep the project's `README` file open at this stage, as it should contain the information we're interested in, only in a less normalised format.

```
opam-version: "1.2.0"
maintainer: "Name <email>"
author: "Name <email>"
name: "project"
version: "0.1"
homepage: ""
bug-reports: ""
license: ""
build: [
  ["./configure" "--prefix=%{prefix}%"]
  [make]
  [make "install"]
]
remove: ["ocamlfind" "remove" "project"]
depends: "ocamlfind"
```

The `opam-version`, `maintainer` and `version` fields are mandatory ; you should remove the others rather than leave them empty. - You'll probably be the `maintainer` for now, so give a way to contact you in case your package needs maintenance. - Most interesting is the `build` field, that tells OPAM how to build and install the project. Each element of the list is a single command in square brackets, containing arguments either as a string (`"install"`) or a variable name (`make`, defined by default to point at the chosen "make" command – think `$(MAKE)` in Makefiles). `%{prefix}%` is another syntax to replace variables within strings. - `remove` is similar to `build`, but tells OPAM how to uninstall. This is indeed `remove: [ ["ocamlind" "remove" "project"] ]`, but the extra square brackets are optional when there is a single element, just add them if you need more than one command. - `depends` should be a list of existing OPAM package names that your package relies on to build and run. You'll be guaranteed those are there when you execute the `build` instructions, and won't be changed or removed while your package is installed.

A few other fields are available, but that should be enough to get started. Like `remove`, most fields may contain lists in square brackets rather than a single element: `maintainer`, `author`, `homepage`, `bug-reports`, `license` and `depends`.

One you save and quit, OPAM will syntax-check and let you edit again in case of errors.

Installing

The best test is to let OPAM attempt to install your just created package. It should prompt you to do so already, but as for any package, you can do it by hand with:

```
$ opam install <project> --verbose
```

At this point, OPAM will get a snapshot of the project, resolve its dependencies and propose a course of actions for the install. Let it go and see if your project installs successfully ; it's a good idea to use `--verbose` as it will make OPAM print the output of the externals commands, in particular the ones in the `build` instructions.

You can now check that everything is installed as expected. Do also check that `opam remove <project>` works properly.

If you need to change anything, simply do

```
opam pin edit <project>
```

to get back to editing the `opam` file. Manually editing the `opam/opam` file in your source tree also works.

So far, so good ! You may have missed dependencies in case they were already installed on your system, but that will be checked automatically by the continuous integration system when you attempt to publish your package to the OPAM repository, so don't worry.

Getting a full OPAM package

There are still two things missing for a complete package. - An appealing description. Put it in a simple utf-8 text file `opam/descr`. Like for git commits, the first line is a short summary, and a longer text may follow. - An URL where OPAM may download the project source for the release. If your project is hosted on github, pushing `TAG` will automatically provide https://github.com/me/project/archive/TAG.zip. This shoud be put in `opam/url`, which has a format similar to `opam`:

```
archive: "https://address/of/project.1.0.tar.gz"
checksum: "3ffed1987a040024076c08f4a7af9b21"
```

The checksum is a simple md5 of the archive, which you can obtain with:

```
curl -L "https://address/of/project.1.0.tar.gz" | md5sum
```

That's it !

Publishing

Publishing is currently handled through Github, using the pull-request mechanism. If you're not familiar with it, it is a fancy way to: - Make a patch to the OPAM repository - Propose this patch for review and integration. This will also trigger tests that your package installs successfully from scratch.

Here is how to do it from scratch:

1. Go to https://github.com/ocaml/opam-repository and hit the `Fork` button on the top right corner (you may be asked to login or register)
2. Get the `clone URL` on the right, and, from the shell, run `git clone <url>` to get your local copy
3. Now we'll add the new package description into `opam-repository/packages/<project>/<project>.<ver` and make that a git commit:

   ```
   $ cd opam-repository/packages
   $ mkdir -p <project>/<project>.<version>
   $ cp <project-src>/opam/* <project>/<project>.<version>
   $ git add <project>
   $ git commit -m "Added new fancy <project>.<version>"
   ```

4. Sending that back to github is just a matter of running `git push`
5. Back to the web interface, refresh, hit the `Pull request` button, check your changes and confirm;
6. Wait for feedback !

Don't forget to `opam pin remove <project>` once your project is on the repository, if you don't want to continue using your local version. Remember that as long as the package is pinned, OPAM will use the metadata found in its source if any, but otherwise only what is in the OPAM repository matters. Use `git pin list` to list all currently pinned packages.

Some tricks

- You may skip the first step and pin to a remote version-controlled repository directly, using for example

  ```
  $ opam pin add <project> git://github.com/me/project.git
  ```

- OPAM will propose to save your opam file back to your source, but if you want to take a peek at the internal version it's at `~/.opam/<switch>/overlay/<project>/`

- You can set OPAM to use your local clone of the repository with

  ```
  $ opam repository add my-dev-repo path/to/opam-repository
  ```

  Don't forget to `opam pin remove <project>`, and test your changes to the repo directly. Don't forget to `opam update my-dev-repo` each time to keep OPAM in sync (`opam update` synches all repos, this will be faster).

- Pins can also be used to try out experimental changes to a project with minimal effort: you can pin to a git repository and even to a specific branch, tag or hash by adding `#BRANCH` to the target. So say you want to try out Joe's GitHub pull-request present on his branch `new-feature` on his fork of `project`, just do

  ```
  $ opam pin project git://github.com/Joe/project.git#new-feature
  ```

and OPAM will use that to get the source (and possibly updated metadata) of the package; this works with any branch of any git repo, it's not github specific.

More on opam files

The opam files can express much more than what was shown above. Without getting into too much details, here are some of the most useful features:

- **Version constraints**: an optional version constraint can be added after any package name in `depends`: simply write `"package" {>= "3.2"}`. Warning, versions are strings too, don't forget the quotes.
- **Formulas**: depends are by default a conjunction (all of them are required), but you can use the logical "and" `&` and "or" `|` operators, and even group with parens. The same is true for version constraints: `("pkg1" & "pkg2") | "pkg3" {>= "3.2" & != "3.7"}`.
- **OS and OCaml constraints**: The `available` field is a formula that determines your package availability based on the os, OCaml version or other constraints. For example:

  ```
  available: [ os != "darwin" | ocaml-version >= "4.00" ]
  ```

- **Conflicts**: some packages just can't coexist. The `conflicts` field is a list of packages, with optional version constraints.
- **Optional dependencies**: they change the way your package builds, but are not mandatory. The `depopts` field is a simple list of package names. If you require specific versions, add a `conflicts` field with the ones that won't work.

- **Variables**: you can get a list of predefined variables that you can use in your opam rules with `opam config list`.
- **Filters**: full commands, or single commands arguments, may need to be omitted depending on the environment. This uses the same optional argument syntax as above, postfix curly braces, with boolean conditions:

```
["./configure" "--with-foo" {ocaml-version > "3.12"} "--prefix=%{prefix}%"]
[make "byte"] { !ocaml-native }
[make "native"] { ocaml-native }
```

For more, see the OPAM Developer's Manual