Developing with OPAM

The goal of this tutorial is to give some insights about how to efficiently use OPAM to ease the development of OCaml software. Mostly we're going to study some real cases of projects whose developing workflow is organized around OPAM in order to understand how they use OPAM and how you can adapt it for your own projects. The examples will vary in complexity, going from the simplest to the most complex.

Simple project

By simple project, we mean a project that has OCaml libraries as dependencies, but that does not include libraries itself. In this case, you can obviously install the libraries using OPAM. That's the normal, ordinary thing that OPAM do for you anyway, so we can say there is nothing special here.

Bigger project

There are some cases where your project is simple, but still the scenario diverge from what has been presented above. Various things might happen, that we're going to examine in order. Let us list them for more clarity:

1. OPAM's repository is not even up to date!
2. Your project uses libraries that are not packaged in OPAM
3. Your project uses libraries that are packaged in OPAM but you need a specific version that is not packaged, or you need to use a modified version of this library.
4. Your project uses development version of libraries (git, darcs, ...) that are not and may never be in OPAM's official repository
5. Your project uses a library that you hacked yourself, and you don't plan to release the modifications nor put them anywhere on the web

I want more control over my OPAM repository

Let us start by addressing issue 1. It may happen that the official OPAM repository is not up to date, or contains buggy packages. In this case, the first important thing to do is inform OPAM's team about it by creating a new *issue* on the opam-repository github page, since the problems you're having with it are very likely to affect other users as well.

If you're not wanting to wait for someone to fix it, here is what you can do:

Creating your own OPAM repository by cloning `opam-repository`

This will give you maximal control over the repository OPAM is using. Just do a:

```
git clone git://github.com/ocaml/opam-repository.git
opam remote add local /path/to/opam-repository
```

Now you can hack into OPAM's packages at will, add your own, etc. The drawback of this method is that it is now your responsability to keep this repository up-to-date by periodically running `git pull` and handling conflicts if they appear. Don't forget to submit a pull-request to github if you fixed issues that may affect others!

Creating your own packages

To address issue 2 where the libraries you need are not packaged for OPAM, you are encouraged to package them yourself and you will find some information on how to create packages in the OPAM packaging tutorial. If the library or program you are packaging has versions — that is, there is an URL where you can download a specific version of that library, and that link points to an archive that does not change with time — this package is eligible to be included in the OPAM official repository and you are encouraged to submit a pull request for it. If you find yourself unable to create this package, you can make a request an addition on opam-repository's issue tracker.

Leveraging `opam pin`

Regarding issue 3 above, if you need a specific version of a library, you should use

```
opam pin <package> <version>
```

to make sure OPAM will not automatically upgrade to another version. If the version you need is not packaged, you should first consider packaging it – this is usually just a matter of copying the previous version's package and adjusting version, url and checksum ; see above.

Otherwise, you can easily locally divert a package url and definition using:

```
opam pin <package> <url>
```

This will make OPAM get the source of the package at , which can possibly point to a version-control system. In this case, OPAM will update at every `opam update` command and suggest an upgrade if there were any changes. You can even specify a local path instead to quickly and easily attempt patches on the library.

This will use the dependencies, build instructions and other metadata found for the latest version of the package in the repository. In case they don't apply anymore, you have two options: * locally, use `opam pin --edit` to customize the opam file. * the most convenient for developpement packages is to include an `opam` file at their root, which will be picked up by OPAM. You can instead provide an `opam` directory that will then override *all* metadata (including `descr` and `files/`)

Note that this can even be used to temporarily create packages that don't exist in the repositories. Remember that you'll need to properly define them at some point if you plan to distribute your software though !

The custom metadata is located at `<opamroot>/<switch>/overlay/<package>`, if you need to access it directly.

Interlude: keeping your OPAM installation clean

As you can probably guess by now, adapting OPAM to your developing needs is likely to interfere with your normal use of it. What if, for example, you need to *pin* some libraries and that other OPAM packaged programs do not like that ? At this point, it is probably wiser to maintain two "instances" of OPAM, one for your "normal" use, and one for the project you are developing. And perhaps more, if you develop more than one such project. You can create such instances by using the `opam switch` command, here is how:

```
opam switch install 4.00.0-myproject --alias-of 4.00.0
```

This will make OPAM install a new OCaml 4.00.0 for your project under the name `4.00.0-myproject`. With OPAM, each *compiler* is associated with its own set of packages, so you can have as many different OPAM installations as you have compilers. If you did not install OCaml 4.00.0 under OPAM before, this command is going to take some time since it needs to download and install OCaml 4.00.0 for you. To switch back and forth between your compiler instances, do:

```
opam switch list
opam switch &lt;alias&gt;
eval $(opam config env)
```

The first command will list the available instances you can switch to, whereas the second one will actually make opam switch to compiler *alias*. The last one is to update variables like `PATH` in your current shell's environment so that the proper commands are picked up. If getting the right environment for each project still is too much burden, consider using the ocp-manager tool: it makes using the right switch transparent without need for modifying environment variables.

Conclusion

It is time to conclude this tutorial by addressing our last points and summarize a bit. OPAM helps you in your developing tasks because it is flexible with repositories. You can achieve more flexibility by managing your repositories yourself. Most projects that use OPAM for development — for example Mirage — use an additional repository that contains all development packages required to build the development version of the project. When you add repositories with `opam repo`, if the same package — that is, same name and same version, which identify a package under OPAM — is present in two repositories, the one from the repository with the highest priority (the last you added, by default) will be used.

This implies a possibile workflow that we're going to present now:

Possible workflow

- Use the default OPAM repository as a base, i.e. you started by `opam init`.
- Create a repository for your project, add it to OPAM with `opam repo add dev <uri>`, `<uri>` being an *http* or *git* url, or a filesystem's path.
- Add specific packages to your remote repository, don't forget to do an `opam update` after each modification.

This workflow is inspired by the one used by Mirage, and helps us solve our points 4 and 5.

To use development packages as well as private code, simply make packages for them in your development repository. You will most likely use the standard packages as a template for the development packages, and modify them to use the development version instead of a released version.

OPAM supports **git, mercurial and darcs packages**, that is, it is able to use a git, mercurial or darcs repository as a source for a package. You will find how to achieve that in the relevant section of the packaging page of the OPAM website, as well as another method if you use *Github* to host your project. Support for other VCS might be added in the future.

Similarly, to use private code, you have to create a package out of it, either purely locally with `opam pin` or in your private repository if you need to share it.

If you use development packages, you can either change their names, for example naming them `foo-git` for the development version of library *foo*, or using a specific version number: letters being ordered after numbers, we suggest using a package directory `foo.dev` in your private opam repository, that will make the version higher than any normal version number.

At this point, you probably have figured out what solution will work best for you, and we wish you good luck for your projects. We welcome suggestions for improvement and different workflows.