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 do 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 use libraries that are not packaged in OPAM
3. Your project use 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 use development version of libraries (git, darcs, ...) that are not and will never be in OPAM's official repository
5. Your project use 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. Is it very possible, even likely, that the OCamlPro 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, and maybe even make a patch and create a pull request, if you feel inspired and generous.

In any case, you don't want to wait for the OPAM team to fix the problem, and you don't have to. There is two things you can try at this point:

Using github's (unstable) opam-repository as your repository

Doing that, you will be able to use the bleeding-edge repository where all newly created packages are added. It can help if the problem has already been fixed in the git repository, but the changes are not yet propagated to the official HTTP one. The drawback is that this repository is generally more unstable than the other. If you know debian, think about its stable *vs* unstable repositories. To do that:

```
opam remote add unstable git://github.com/OCamlPro/opam-repository.git
```

This is **not the recommended** method however, as it is not likely to solve your problem.

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/OCamlPro/opam-repository.git
opam remote add local /path/to/opam-repository
```

Now you can hack into OPAM's packages at will, add your own ones, etc. The only drawback of this method is that it is now your responsability to keep this repository up-to-date by periodically `git pull`ing OPAM's official one and handling conflicts if they appear.

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 is 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 add an issue in opam-repository's issue tracker on github but there is no guarantee that anybody will find the time or interest to make the package for you, although the OPAM team try to make an official package for all useful OCaml libraries and programs.

Leveraging `opam pin`

Regarding issue 3 where you need a specific version of a package that is not packaged in the official repository, you can create a new package for this specific version in your own personal repository — just copy the directory of the corresponding package and hack it to adapt to the version you need. You can then do:

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

to make OPAM use this specific version of the library and prevent OPAM to upgrade it with the other OPAM packages on `opam upgrades`. If you need a modified version of an OPAM library, you should use:

```
opam pin <path>
```

to instruct OPAM to use a given path (in the filesystem's sense) to use as a source for a package. This way, you can download the version of the library you want to use yourself, modify it to your needs, and tell OPAM to use this path as the source path for building the package for the library. Doing so will have the effect of locking this library to a version and a source path, prevent it to be upgraded with other OPAM packages when an `opam upgrade` is performed.

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 list
opam switch <alias>
```

The first command will list the available instances you can switch to, whereas the second one will actually make opam switch to compiler *alias*.

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 remote`, 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 you added first will be used.

This default priority for repositories can be **fine-tuned** on a package basis by editing `~/.opam/repo/index`: for each package, one or more repositories are specified, in the case there is more than one, the order of the repositories reflects the order in which OPAM will try to get the package from them, the first being the most prioritary. You can edit this file by hand, thus changing this order, just don't forget do to a `opam update` afterwards.

This implies a possibility of a 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 remote 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 of it.

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 and darcs packages**, that is, it is able to use a git 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 and add this package to your private repository.

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 high version number that will have priority over all released version, maybe adding a package directory `foo.999` in your private opam repository.

At this point, you probably have figured out what solution will work best for you, and we wish you good luck for your projects. Please feel free to make any suggestion on how to improve this tutorial and make us know if you use a different workflow from what has been described in this tutorial.