Creating OPAM packages

In this tutorial, you will learn how to package an OCaml library or software for OPAM. The first section will introduce you the *ounit* OPAM package. Albeit simple, it is a real example of what an OPAM package is, and actually most packages in the OPAM repository are that simple. The second section will be a comprehensive guide illustrated by more complicated examples of real packaging cases.

An important thing to keep in mind is that OPAM is (at least for now) **only** a package manager, as opposed to — for example — the *oasis/odb* suite, which includes a build system (*oasis*) as well as a package manager (*odb*). This means that OPAM cannot help you to actually build your ocaml software or library, to do so, you need to use dedicated tools such as *ocamlbuild*, *oasis*, *omake* and others. OPAM packages are just built the same way you would build the software yourself (with shell commands).

The *ounit* OPAM package

Let get started by learning from the *ounit* OPAM package. This is a widely used OCaml library to create unit tests for OCaml projects, and is a perfect example of a minimalistic yet complete package.

By reading about *ounit* on the upstream website, you learn that it is a library to create unit tests, that its latest version is *1.1.2*, and that it depends on ocamlfind. You know the URL you can download its source tarball, and you must compute the *md5sum* of this source tarball by running `md5sum ounit-1.1.2.tar.gz` (or, on some operating systems, `md5` instead of `md5sum`).

You also learn that to build and install it, you have to:

```
make build
make install
```

And that's all the information you need to build an OPAM package.

A minimum OPAM package is a directory containing three files: `descr`, `opam`, and `url`. The name of the directory defines the package name and version: `<package-name>.<package-version>`. In our case, the directory will be `ounit.1.1.2` and contain the following files:

- `packages/<package-name>.<package-version>/descr`

```
Unit testing framework inspired by the JUnit tool and the HUnit tool
```

- packages/<package-name>.<package-version>/opam

```
opam-version: "1"
maintainer: "contact@ocamlpro.com"
build: [
  [make "build"]
  [make "install"]
]
remove: [
  ["ocamlfind" "remove" "oUnit"]
]
depends: ["ocamlfind"]
```

- packages/<package-name>.<package-version>/url

```
archive: "http://forge.ocamlcore.org/frs/download.php/886/ounit-1.1.2.tar.gz"
checksum: "14e4d8ee551004dbcc1607f438ef7d83"
```

Notes

descr

This file is pure text. Its first line will be used as a short description of the package, it is what is displayed for each package when you do `opam list`. The whole text contained in there is displayed when you do `opam search <package>`. Therefore you should put a short meaningful description on the first line, and a long description starting from the second line.

opam

The full ABNF specification of the syntax for *opam* files is available in OPAM developer manual. In this file, `opam-version` MUST be `1`, and you should put your email in the `maintainer` field. `build` has OCaml type `string list list`, and contains the build instructions. Here,

```
make build
make install
```

gets translated into

```
build: [
  [make "build"]
  [make "install"]
]
```

You should adapt this to the required commands to build your package, and each line contains the shell commands corresponding to a `string list`. Note that `make` is a special variable which will be automatically translated to either `make` on linux and OSX or 'gmake' on BSD systems.

The `remove` field follows the same syntax as the `build` field. The `depends` field is a `string list` of dependencies, with each dependency being another OPAM package. Here *ounit* depends only on *ocamlfind*.

`url`

This file contains at least one `archive` line containing the URL of the source package, and optionally a `checksum` line that must contain the MD5 sum of the source package if it is present. It is good practice to systematically add a checksum line to your packages, unless the source package has no fixed version (the typical example being a source package hosted on *github* with no tags). This checksum will be checked when creating and installing the package.

The URL can also contain a single `git` or `darcs` field instead of `archive`, which points to GIT or DARCS repository URL. This will be checked out and updated every time `opam update` is run, which is useful for development packages.

Testing custom OPAM packages

The easiest way to test your new packages is to set-up a local repository for testing purposes.

```
$ mkdir -k /tmp/testing
$ opam repo add testing /tmp/testing
```

These commands add a new (currently empty) repository named `testing` (you can pick an other name if your prefer) which will contain what is in `/tmp/testing` (*Remark*: you can also clone the official git repository if you don't want to start from a fresh one, this will work as well).

You can now check that this new repository exists:

```
$ opam repo  # eq. to 'opam repo list'
```

This command displays the list of repositories, with `testing` having the highest priority (you can use `opam repo priority` to change the relative repository priorities later).

Now it is time to populate `/tmp/testing/packages` with your new package files. For instance, if you want to test the version `1.1.3` of `ounit`, you have to create `/tmp/packages/ounit.1.1.3/{opam,descr,url}` following the guidelines defined above.

To take this changes into account, update your testing repository:

```
$ opam update testing
```

If everything is fine, OPAM should tell you than a new version of `ounit` is available. If this is the case, you can install it by doing:

```
$ opam install ounit.1.1.3
```

Remark: you can use `opam-mk-repo` to simulate the creation of OPAM package archives done on `opam.ocamlpro.com`:

```
$ cd /tmp/testing && opam-mk-repo -g ounit
```

This command will: * download the upstream archive, and generate the correct checksum (because of `-g`); * create the archivearchives/ounit.1.1.3+opam.tar.gzcontaining the content of the upstream archive + the files inpackages/ounit.1.1.3/files/‘.

If `archives/ounit.1.1.3+opam.tar.gz` exists, OPAM will use it directly instead of downloading the archive upstream.

If (i) the basic installation and (ii) the archive creation work, you are in good shape to submit your new package upstream (see below).

Advanced OPAM packaging guide

This section will be as comprehensive as possible on the art of creating OPAM packages, but in case of ambiguities, the ABNF syntax documentation has priority.

Since everything has already be said about the `descr` file and almost everything about the `url` file, this section is mostly about the `opam` files.

OPAM variables

OPAM maintains a set of variables (key value pairs) that can be used in `opam` files and that will be substituted by their values on package creation. The list of variables that can be used in `opam` files can be displayed by doing `opam config list`. The following example shows the `build` section of package *ocamlnet* that use the variable `bin`:

```
build: [
  ["./configure" "--bindir" bin]
  [make "all"]
  [make "opt"]
  [make "install"]
]
```

In this case, `bin` will be substituted by the value of the `bin` variable. In case you need to substitue a substring, you can use `"--bindir=%{bin}%"`: here `%{bin}%` will be substituted by the value of `bin`.

Optional dependencies

We mentioned the ability to specify optional dependencies for packages. If a package has optional dependencies, they will not be installed automatically, but will be taken into account if they are present *before* the installation. If the optional dependency is not present, but are subsequently installed, then the depending package will also be recompiled to take advantage of the newly installed library.

Let us see how it works via the following example, which shows the `opam` file of the *lwt* package:

```
opam-version: "1"
maintainer: "contact@ocamlpro.com"
build: [
  ["./configure" "--%{conf-libev:enable}%-libev" "--%{react:enable}%-react" "--%{ssl:enable}
  [make "build"]
  [make "install"]
]
remove: [
  ["ocamlfind" "remove" "lwt"]
]
depends: ["ocamlfind"]
depopts: ["base-threads" "base-unix" "conf-libev" "ssl" "react"]
```

Notice the new `depopts` field, which contains the list of optional dependencies, specified in the same format as the `depends` field.

Also notice a new syntax for substitutions of the form `%{<package>:enable}%`. If *package* is installed, this pattern will be replaced by `enable`, otherwise by `disable`. This ease the building of lines of type `./configure --enable-<feature1> --disable-<feature2>`.

Version constraints

If a package depends (respectively optionally depends) on a specific version of a package, this can be specified by using the following syntax for the field `depends` (respectively `depopts`) of the `opam` file:

```
depends: [ "ocamlfind" "re" "uri" "ounit" ]
depopts: [ "async" {= "108.00.02"} "lwt" "mirage-net" ]
```

The above example shows two fields of the `opam` file of package *cohttp*, that optionally depends of the version 108.00.02 of package *async*.

The OPAM specification document specifies the format used by the `depends` and `depopts` fields. As there is much more to say about version constraints, you should read it to learn how to write very fine grained version constraints.

Manually installing binaries or libaries

Most of the time, when a package is built, binaries and/or libraries that it provides are installed automatically. But sometimes a source package does not install a binary that it produced because it has not been included its `make install` command (or equivalent). For example, the package *xmlm* has this following `opam` file:

```
opam-version: "1"
maintainer: "contact@ocamlpro.com"
build: [
  ["ocaml" "setup.ml" "-configure" "--prefix" prefix]
  ["ocaml" "setup.ml" "-build"]
  ["ocaml" "setup.ml" "-install"]
]
remove: [
  ["ocamlfind" "remove" "xmlm"]
]
depends: ["ocamlfind"]
```

and has additional file `files/xmlm.install`:

```
bin: ["_build/test/xmltrip.native" {"xmltrip"}]
```

This has the semantic: "install the file of path `_build/test/xmltrip.native` relative to the root of the source package into the directory returned by the command `opam config var bin` under the name `xmltrip`". If the source filename starts by `?`, the installation will not fail if the file is not present.

You can install additional libraries and toplevels the same way. For instance:

```
lib: [ "META" "lib/foo.cmi" "lib/foo.cmo" "lib/foo.cmx" ]
```

Remark: ideally, the `.install` should be dynamically created by the package build system at the root of the project and should be named `$(OPAM_PACKAGE_NAME).opam` (`$OPAM_PACKAGE_NAME` is automatically set by OPAM). However, due to the lack of such support for this feature in existing build system, most of the existing packages have a *static* `.install` file, which is usually sufficient for simple needs.

For comprehensive information about this facility refer to the Section 1.2.5 of OPAM developer manual

Compiler version constraints

Some packages require a specific OCaml version to work and thus can only be installed under specific compiler versions. To specify such a constraint, you can add a field `ocaml-version: [ < "4.00.0" ]` to the `opam` file. This particular constraint implies that the package cannot be built or installed under OCaml 4.00.0 or later.

Patching sources

You can instruct OPAM to apply patches to the source code before building a package. To do so, you have to add a *patches* field to the opam file, the syntax being of the form `patches: ["bugfix1.patch" "bugfix2.patch"]`, where *bugfix1.patch* and *bugfix2.patch* are two files existing in the directory `files`. Before building such a package, OPAM will substitute any opam variable (of the form `%{variable}%`) to their respective values and apply the resulting patches to the source code. Only then will the package be built. For more information, please look at packages including patches, such as `dbm.1.0`.

Git / Darcs packages

It is possible to use a git repository instead of an archive file in `url` files. To do so, you need to use the following syntax:

```
git: "<url>"
```

`<url>` being any url that git knows how to clone. For git packages, OPAM has the following behaviour:

- When installing a git package, OPAM will use git to clone its url and use it as the package source

- When updating packages, OPAM will do a `git fetch` in order to have the last patches available for git packages
- When upgrading packages, OPAM will merge the last changes before rebuilding and upgrading the packages

If you host your project on *Github*, you may not use git packages but instead use github's functionality to create a tarball from a git repository. Is is generally available at `https://github.com/<your-id>/<your-project>/tarball/<branch-or-tag>`. You can use this url to create "normal" — non-git packages from git repositories hosted on github.

Note that git packages will normally never be included in the default OPAM repository, and are mainly an aid for developers who use OPAM in their development process. If you plan to do that, please have a look at the Developing with OPAM tutorial.

Darcs packages

Darcs repositories are supported as well. OPAM behaves the same way it does with git repositories, as described above. You just need to specify the repository url using the following syntax in `url` files:

```
darcs: "<url>"
```

Where to go from here

Although this tutorial covered most packaging cases, there are still packages that requires more tuning that what have been described above. If you find yourself stuck trying to package a software or a library, please read the OPAM developer manual (you will find it in the `doc` directory in the OPAM tarball) and/or read existing OPAM package descriptions for inspiration.

Including packages to the official OPAM repository

This section will help you getting started with the process of submitting packages to the official OPAM repository. This repository is available at url [http://opam.ocamlpro.com], but its content is generated from a git repository hosted on github.

To submit a package for inclusion in the official repository, all you have to do is to fork `opam-repository` on github, commit a patch containing the package(s) you want to include, and open a pull request for it.

If the above sentence makes no sense for you, you probably don't know about either *git* or *github* (or both). *git* is a distributed version control system, very popular at the time we write those lines. Getting started with git is definitely a topic outside the scope of this short tutorial, but you can read about it on git's documentation page.

Github is a web frontend to git. It allows users to store public git repositories, and provides additional convenient features over git such as "pull requests" that automatise the process of sharing patches with others. You can learn how to use it here.

If you cannot (or do not want to) use github but still want to contribute to packages, you can try sending a mail to `contact@ocamlpro.com` with your patches to `opam-repository`. As this method requires manual intervention from our not very big OPAM team, it should only be used if the first method is not an option for you. It might as well take more time for your request to be processed in this case.