22 The OCaml Platform

So far in Part II, we've gone through a number of libraries and techniques you can use to build larger scale OCaml programs. We'll now wrap up this part by examining the tools you can use for editing, compiling, testing, documenting and publishing your own projects.

The OCaml community has developed a suite of modern tools to interface it with IDEs such as Visual Studio Code, and to generate API documentation and implement modern software engineering practices such as continuous integration (CI) and unit or fuzz testing. All you need to do is to specify your project metadata (for example, library dependencies and compiler versions), and the OCaml Platform tools that we'll describe next will do much of the heavy lifting.

Using the Opam Source-Based Package Manager

opam is the official package manager and metadata packaging format that is used in the OCaml community. We've been using it in earlier chapters to install OCaml libraries, and we're going to take a closer look at how to use opam within a full project next. You've almost certainly done this already at this point in the book, but in case you've skipped straight to this chapter make sure you first initialize opam's global state.

```
$ opam init
```

By default, opam doesn't require any special user permissions and stores all of the files it installs in `~/.opam`, such as the current build of the OCaml compiler if you didn't have one pre-installed when you initialized opam.

You can maintain multiple development environments with different packages and compilers installed, each of which is called a "switch" – the default one can be found under `~/.opam/default`. Run `opam switch` to see all the different sandboxed environments you have available:

```
$ opam switch
#  switch    compiler      description
   default   ocaml.4.13.1  default
```

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

22.1 A Hello World OCaml Project

Let's start by creating a sample OCaml project and navigating around it. Dune has a basic built-in command to initialize a project template that is suitable to get us started.

```
$ dune init proj hello
Success: initialized project component named hello
```

Dune will create a `hello/` directory and populate it with a skeleton OCaml project. This sample project has the basic metadata required for us to learn more about the opam package manager and the dune build tool that we've used earlier in the book.

22.1.1 Setting Up an Opam Local Switch

The next thing we need is a suitable environment for this project, with dune and any other library dependencies available. The best way to do this is to create a new opam sandbox, via the `opam switch create` command. If you specify a project directory argument to this command, then it creates a "local switch" that stores all the dependencies within that directory rather than under `~/.opam`. This is a convenient way to keep all your build dependencies and source code in one place on your filesystem.

Let's make a local switch for our hello world project now:

```
$ cd hello
$ opam switch create .
```

This invokes opam to install the project dependencies (in this case, just the OCaml compiler and dune as we didn't specify any more when initializing the project). All of the files from the local switch will be present under `_opam/` in the working directory. You can find the dune binary that was just compiled inside that directory, for example:

```
$ ./_opam/bin/dune --version
3.0.2
```

Since opam will install other binaries and libraries in the local switch as your project grows, you will need to add the switch to your command-line path. You can use `opam env` to add the right directories to your local shell path so that you can invoke the locally installed tools:

```
$ eval $(opam env)
```

If you prefer not to modify your shell configuration, then you can also invoke commands via `opam exec` to modify the path for the subcommand specified in the remainder of the command line.

```
$ opam exec -- dune build
```

This executes `dune build` with the opam environment variables added to the command invocation, so it will pick up the locally built dune from your project. The double dash in the command line is a common Unix convention that tells opam to stop parsing its own optional arguments for the remainder of the command, so that they don't interfere with the command that is being executed.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

22.1.2 Choosing an OCaml Compiler Version

When creating a switch, opam analyses the project dependencies and selects the newest OCaml compiler that is compatible with them. Sometimes though, you will want to select a specific version of the OCaml compiler, perhaps to ensure reproducibility or to use a particular feature. You can use `opam switch list-available` to get a full list of all the compilers that are available.

```
ocaml-system          4.13.1          The OCaml compiler
                                      (system version, from outside
                                       of opam)
ocaml-base-compiler   4.13.1          Official release 4.13.1
ocaml-variants        4.13.1+options  Official release of OCaml 4.13.1
```

You'll find many more versions present than the snippet above, but notice that there are three different types of OCaml compiler packages present.

`ocaml-system` is the name opam uses for the pre-existing version of the OCaml compiler that was already installed on your machine. This compiler is always fast to install since nothing needs to be compiled for it. The only thing needed to create a system switch is to have the right version of OCaml already installed (e.g. via `apt` or Homebrew) and to pass the same version to the switch creation as an additional argument.

For example, if you have OCaml 4.13.1 installed, then running this command will use the system compiler:

```
$ opam switch create . 4.13.1
```

On the other hand, if you didn't have that system compiler installed, then the compiler will need to be built from scratch. The command above would select the `ocaml-base-compiler` package in this case, which contains the full OCaml compiler source code. It will take a little longer than `ocaml-system`, but you have much more flexibility about the choice of versions. The default operation of `opam switch create` is to calculate the latest supported compiler version from your project metadata and use that one for the local switch.

If you always want to locally install a particular compiler, then you can refine the package description:

```
$ opam switch create . ocaml-base-compiler.4.13.1
```

Sometimes, you will also need to add custom configuration options to the compiler, such as the `flambda` optimiser. There are two packages that handle this: `ocaml-variants` is a package that detects the presence of various `ocaml-option` packages to activate configuration flags. For example, to build a compiler with `flambda`, you would:

```
$ opam switch create . ocaml-variants.4.13.1+options
    ocaml-option-flambda
```

You can specify multiple `ocaml-option` packages to cover all the customization your project needs. See the full set of option packages by using:

```
$ opam search ocaml-option
```

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

22.1.3 Structure of an OCaml Project

Back in Chapter 5 (Files, Modules, and Programs), we looked at what a simple program with a couple of OCaml modules looks like. Let's now look at the set of files in our `hello/` application to examine a fuller project structure.

```
.
|-- dune-project
|-- hello.opam
|-- lib
|   |-- dune
|-- bin
|   |-- dune
|   `-- main.ml
`-- test
    |-- dune
    `-- hello.ml
```

Some observations about this structure:

- The `dune-project` file marks the root of the project, and is used for writing down some key metadata for the project (more on that later).
- The `hello.opam` file contains metadata for registering this software as an opam project. As we'll see, we won't need to edit this manually because we can generate the file via `dune`.
- There are three source directories, each with its own `dune` file specifying the build parameters for that part of the codebase. The trio of `lib`, `bin` and `test` makes good sense for a project that is primarily an executable, rather than a reusable library. In that case, you would might use these directories as follows:
  - The `lib` directory would contain the bulk of the source.
  - The `bin` directory would contain a thin wrapper on top of the code in `lib` which actually launches the executable.
  - The `test` directory has the bulk of the tests for `lib`, which, following the advice in Chapter 18.1.2 (Where Should Tests Go?), are in a separate directory from the source.

Now we'll talk about the different parts of this structure in more detail.

22.1.4 Defining Module Names

A matching pair of `ml` and `mli` files define an OCaml module, named after the file and capitalized. Module names are the only kind of name you refer to within OCaml code.

Let's create a `Msg` module in our skeleton project inside `lib/`.

```
$ echo 'let greeting = "Hello World"' > lib/msg.ml
$ echo 'val greeting : string' > lib/msg.mli
```

A valid OCaml module name cannot contain dashes or other special characters other than underscores. If you need to refresh your memory about how files and modules interact, refer back to Chapter 5 (Files, Modules, and Programs).

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

22.1.5 Defining Libraries as Collections of Modules

One or more OCaml modules can be gathered together into a *library*, providing a convenient way to package up multiple dependencies with a single name. A project usually puts the business logic of the application into a library rather than directly into an executable binary, since this makes writing tests and documentation easier in addition to improving reusability.

Libraries are defined by putting a `dune` file into a directory, such as the one generated for us in `lib/dune`:

```
(library
 (name hello))
```

Dune will treat all OCaml modules in that directory as being part of the `hello` library (this behavior can be overridden by a `modules` field for more advanced projects). By default, dune also exposes libraries as *wrapped* under a single OCaml module, and the `name` field determines the name of that module.

In our example project, `msg.ml` is defined in `lib/dune` which defines a `hello` library. Thus, users of our newly defined module can access it as `Hello.Msg`. You can read more about wrapping and module aliases in Chapter 26.4.4 (Wrapping Libraries with Module Aliases). Although our `hello` library only currently contains a single `Msg` module, it is common to have multiple modules per library in larger projects. Other modules within the `hello` library can simply refer to `Msg`.

You must refer to library names in a `dune` file when deciding what libraries to link in, and never individual module names. You can query the installed libraries in your current switch via `ocamlfind list` at your command prompt, after running `opam install ocamlfind` to install it if necessary:

```
$ ocamlfind list
afl-persistent      (version: 1.2)
alcotest            (version: 1.5.0)
alcotest.engine     (version: 1.5.0)
alcotest.stdlib_ext (version: 1.5.0)
angstrom            (version: 0.15.0)
asn1-combinators    (version: 0.2.6)
...
```

If there's a `public_name` field present in the `dune` library definition, this determines the publicly exposed name for the library. The public library name is what you specify via the `libraries` field in other projects that use your project's libraries. Without a public name, the defined library is local to the current dune project only. The `(libraries)` field in the `lib/dune` file is empty since this is a trivial standalone library.

22.1.6 Writing Test Cases for a Library

Our next step is to define a test case in `test/dune` for our library. In Chapter 18 (Testing), we showed you how to embed tests within a library, using the inline test mechanism. In this section, we'll show you how to use dune's `test` stanza to create a test-only executable, which is useful when you're not using inline tests.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

Let's start by writing a test as a simple assertion in `test/hello.ml`.

```
let () = assert (String.equal Hello.Msg.greeting "Hello World")
```

We can use the `test` dune stanza to build an executable binary that is run when you invoke `dune runtest` (along with any inline tests defined within libraries). We'll also add a dependency on our locally defined `hello` library so that we can access it. The `test/dune` file looks like this:

```
(test
 (libraries hello)
 (name hello))
```

Once you run the tests via `dune runtest`, you can find the built artifacts in `_build/default/test/` in your project checkout.

```
$ ls -la _build/default/test
total 992
drwxr-xr-x  7 avsm  staff     224 27 Feb 16:13 .
drwxr-xr-x  9 avsm  staff     288 27 Feb 15:23 ..
drwxr-xr-x  4 avsm  staff     128 27 Feb 15:23 .hello.eobjs
drwxr-xr-x  3 avsm  staff      96 27 Feb 16:12 .merlin-conf
-r-xr-xr-x  1 avsm  staff  495766 27 Feb 16:13 hello.exe
-r--r--r--  1 avsm  staff      64 27 Feb 16:13 hello.ml
-r--r--r--  1 avsm  staff      28 27 Feb 15:23 hello.mli
```

We deliberately defined two files called `hello.ml` in both `lib/` and `test/`. It's completely fine to define an executable `hello.exe` (in `test/`) alongside the OCaml library called `hello` (in `lib/`).

22.1.7 Building an Executable Program

Finally, we want to actually use our hello world from the command-line. This is defined in `bin/dune` in a very similar fashion to test cases.

```
(executable
 (public_name hello)
 (name main)
 (libraries hello)))
```

There has to be a `bin/main.ml` alongside the `bin/dune` file that represents the entry module for the executable. Only that module and the modules and libraries it depends on will be linked into the executable. Much like libraries, the `(name)` field here has to adhere to OCaml module naming conventions, and the `public_name` field represents the binary name that is installed onto the system and just needs to be a valid Unix or Windows filename.

Now try modifying `bin/main.ml` to refer to our `Hello.Msg` module:

```
let () = print_endline Hello.Msg.greeting
```

You can build and execute the command locally using `dune exec` and the local name of the executable. You can also find the built executable in `_build/default/bin/main.exe`.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

```
$ dune build
$ dune exec -- bin/main.exe
Hello World
```

You can also refer to the public name of the executable if it's more convenient.

```
$ dune exec -- hello
Hello World
```

The `dune exec` and `opam exec` command we mentioned earlier in the chapter both nest, so you could append them to each other using the double-dash directive to separate them.

```
$ opam exec -- dune exec -- hello --args
```

This is quite a common thing to do when integrating with continuous integration systems that need systematic scripting of both opam and dune (a topic we'll come to shortly in this chapter).

22.2 Setting Up an Integrated Development Environment

Now that we've seen the basic structure of the OCaml project, it's time to setup an integrated development environment. An IDE is particularly useful for OCaml because it lets you leverage the information that's extracted by OCaml's rich type-system. A good IDE will provide you with the facilities to browse interface documentation, see inferred types for code, and to jump to the definitions of external modules.

22.2.1 Using Visual Studio Code

The recommended IDE for newcomers to OCaml is Visual Studio Code[1] using the OCaml Platform plugin[2]. The plugin uses the Language Server Protocol to communicate with your opam and dune environment. All you need to do is to install the OCaml LSP server via opam:

```
opam install ocaml-lsp-server
```

Once installed, the VSCode OCaml plugin will ask you which opam switch to use. Just the default one should be sufficient to get you going with building and browsing your interfaces.

> **What Is The Language Server Protocol?**
> The Language Server Protocol defines a communications standard between an editor or IDE and a language-specific server that provides features such as auto-completion, definition search, reference indexing and other facilities that require specialized support from language tooling. This allows a programming language toolchain to implement all this functionality just once, and then integrate cleanly into the multiplicity of

[1] https://code.visualstudio.com
[2] https://marketplace.visualstudio.com/items?itemName=ocamllabs.ocaml-platform

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

IDE environments available these days – and even go beyond conventional desktop environments to web-based notebooks such as Jupyter.

Since OCaml has a complete and mature LSP server, you'll find that an increasing number of IDEs will just support it out of the box once you install the ocaml-lsp-server. It integrates automatically with the various tools we've used in this book, such as detecting opam switches, invoking dune rules, and so on.

22.2.2 Browsing Interface Documentation

The OCaml LSP server understands how to interface with dune and examine the build artifacts (such as the typed `.cmt` interface files), so opening your local project in VS Code is sufficient to activate all the features. Try navigating over to `bin/main.ml`, where you will see the invocation to the `hello` library.

```
let () = print_endline Hello.Msg.greeting
```

First perform a build of the project to generate the type annotation files. Then hover your mouse over the `Hello.Msg.greeting` function – you should see some documentation pop up about the function and its arguments. This information comes from the *docstrings* written into the `msg.mli` interface file in the `hello` library.

Modify the `msg.mli` interface file to contain some signature documentation as follows:

```
(** This is a docstring, as it starts with "**", as opposed to normal
    comments that start with a single star.

    The top-most docstring of the module should contain a description
    of the module, what it does, how to use it, etc.

    The function-specific documentation located below the function
    signatures. *)

(** This is the docstring for the [greeting] function.

    A typical documentation for this function would be:

    Returns a greeting message.

    {4 Examples}

    {[ print_endline greeting ]} *)
val greeting : string
```

Documentation strings are parsed by the odoc[3] tool to generate HTML and PDF documentation from a collection of opam packages. If you intend your code to be used by anyone else (or indeed, by yourself a few months later) you should take the time to annotate your OCaml signature files with documentation. An easy way to preview the HTML documentation is to build it locally with dune:

[3] https://github.com/ocaml/odoc

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

```
$ opam install odoc
$ dune build @doc
```

This will leave the HTML files in `_build/default/_doc/_html`, which you can view normally with a web browser.

22.2.3 Autoformatting Your Source Code

As you develop more OCaml code, you'll find it convenient to have it formatted to a common style. The `ocamlformat` tool can help you do this easily from within VSCode.

```
$ echo 'version=0.20.1' > .ocamlformat
$ opam install ocamlformat.0.20.1
```

The `.ocamlformat` file controls the autoformatting options available, and fixes the version of the tool that is used. You can upgrade to a newer ocamlformat version whenever you want, but it is a manual process to avoid an upstream release auto-reformatting your project code without your intervention. You can examine the formatting options via `ocamlformat --help` – most of the time the defaults should be fine.

Once you've got ocamlformat configured, you can either format your project from within VSCode (`shift-alt-F` being the default), or by running:

```
$ dune build @fmt
```

This will generate a set of reformatted files in the build directory, which you can accept with `dune promote` as you did earlier in the testing chapter.

22.3 Publishing Your Code Online

With your IDE all set up you'll quickly develop useful OCaml code and want to share it with others. We'll now go through how to define opam packages, set up continuous integration and publish your code.

22.3.1 Defining Opam Packages

The only metadata file that is really *required* to participate in the open-source OCaml ecosystem is an `opam` file in your source tree. Each `opam` file defines a *package* – a collection of OCaml libraries and executable binaries or application data. Each opam package can define dependencies on other opam packages, and includes build and testing directions for your project. This is what's installed when you eventually publish the package and someone else types in `opam install hello`.

A collection of `opam` files can be stored in an *opam repository* to create a package database, with a central one for the OCaml ecosystem available at `https://github.com/ocaml/opam-repository`. The official (but not exclusive) tool used for manipulating `opam` files is the eponymous opam package manager[4] that we've been using throughout this book.

[4] `https://opam.ocaml.org`

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

How Do We Name OCaml Modules, Libraries and Packages?
Much of the time, the module, library, and package names are all the same. But there are reasons for these names to be distinct as well:

- Some libraries are exposed as multiple top-level modules, which means you need a different name for that collection of modules.
- Even when the library has a single top-level module, you might want the library name to be different from the module name to avoid name clashes at the library level.
- Package names might differ from library names if a package combines multiple libraries and/or binaries together.

It's important to understand the difference between modules, libraries and packages as you work on bigger projects. These can easily have thousands of modules, hundreds of libraries and dozens of opam packages in a single codebase.

22.3.2 Generating Project Metadata from Dune

The `hello.opam` file in our sample project is currently empty, but you don't need to write it by hand – instead, we can define our project metadata using the dune build system and have the opam file autogenerated for us. The root directory of an OCaml project built by dune has a `dune-project` file that defines the project metadata. In our example project, it starts with:

```
(lang dune 3.0)
```

The line above is the version of the syntax used in your build files, and *not* the actual version of the `dune` binary. One of the nicest features of dune is that it is forwards-compatible with older metadata. By defining the version of the dune language that you are currently using, *future* versions of dune will do their best to emulate the current behavior until you choose to upgrade your project.

The rest of the `dune-project` file defines other useful project metadata:

```
(name hello)
(documentation "https://username.github.io/hello/")
(source (github username/hello))
(license ISC)
(authors "Your name")
(maintainers "Your name")
(generate_opam_files true)
```

The fields in here all represent project metadata ranging from textual descriptions, to project URLs, to other opam package dependencies. Go ahead and edit the metadata above to reflect your own details, and then build the project:

```
$ dune build
```

The build command will update the `hello.opam` file in your source tree as well, keeping it in sync with your changes. The final part of the `dune-project` file contains dependency information for other packages your project depends on.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

```
(package
 (name hello)
 (synopsis "A short description of the project")
 (description "A short description of the project")
 (depends
  (ocaml (>= 4.08.0))
  (alcotest :with-test)
  (odoc :with-doc)))
```

The `(package)` stanza here refers to opam packages, both for the name and for the dependency specifications. This is in contrast to the `dune` files which refer to ocamlfind libraries, since those represent the compilation units for OCaml code (whereas opam packages are broader collections of package data).

Notice that the dependency specification can also include version information. One of the key features of opam is that each repository contains multiple versions of the same package. The opam CLI contains a constraint solver that will find versions of all dependencies that are compatible with your current project. When you add a dependency, you can therefore specify lower and upper version bounds as required by your use of that package. The `with-test` and `with-doc` are further constraints that only add those dependencies for test and documentation generation respectively.

Once you've defined your opam and dune dependencies, you can run various lint commands to check that your metadata is consistent.

```
$ opam dune-lint
$ opam lint
```

The `opam-dune-lint` plugin will check that the ocamlfind libraries and opam packages in your dune files match up, and offer to fix them up if it spots a mismatch. `opam lint` runs additional checks on the opam files within your project.

22.3.3 Setting up Continuous Integration

Once you have your project metadata defined, it's a good time to begin hosting it online. Two of the most popular platforms for this are GitHub[5] and GitLab[6]. The remainder of this chapter will assume you are using GitHub for simplicity, although you are encouraged to check out the alternatives to find the best solution for your own needs.

When you create a GitHub repository and push your code to it, you can also add an OCaml GitHub Action that will install the OCaml Platform tools and run your code across various architectures and operating systems. You can find the full documentation online at the Set up OCaml[7] page on the GitHub marketplace. Configuring an action is as simple as adding a `.github/workflows/test.yml` file to your project that looks something like this:

```
name: Hello world workflow
on:
  pull_request:
```

[5] https://github.com
[6] https://gitlab.com
[7] https://github.com/marketplace/actions/set-up-ocaml

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

```
  push:
jobs:
  build:
    strategy:
      matrix:
        os:
          - macos-latest
          - ubuntu-latest
          - windows-latest
        ocaml-compiler:
          - 4.13.x
    runs-on: ${{ matrix.os }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Use OCaml ${{ matrix.ocaml-compiler }}
        uses: ocaml/setup-ocaml@v2
        with:
          ocaml-compiler: ${{ matrix.ocaml-compiler }}
      - run: opam install . --deps-only --with-test
      - run: opam exec -- dune build
      - run: opam exec -- dune runtest
```

This workflow file will run your project on OCaml installations on Windows, macOS and Linux, using the latest patch release of OCaml 4.13. Notice that it also runs the test cases you have defined earlier on all those different operating systems as well. You can do an awful lot of customization of these continuous integration workflows, so refer to the online documentation for more options.

22.3.4 Other Conventions

There are a few other files you may also want to add to a project to match common conventions:

- A `Makefile` contains targets for common actions such as `all`, `build`, `test` or `clean`. While you don't need this when using VSCode, some other operating system package managers might benefit from having one present.
- The `LICENSE` defines the terms under which your code is made available. Our example defaults to the permissive ISC license, and this is generally a safe default unless you have specific plans for your project.
- A `README.md` is a Markdown-formatted introduction to your library or application.
- A `.gitignore` file contains the patterns for generated files from the OCaml tools so that they can be ignored by the Git version control software. If you're not familiar with using Git, look over one of the tutorials one such as GitHub's git hello world[8].

[8] https://guides.github.com/activities/hello-world/

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

22.3.5 Releasing Your Code into the Opam Repository

Once your continuous integration is passing, you are all set to try to tag a release of your project and share it with other users! The OCaml Platform supplies a convenient tool called `dune-release` which automates much of this process for you.

```
$ opam install dune-release
```

The first thing you need to do is to create a `CHANGES.md` file in your project in Markdown format, which contains a header per version. This is typically a succinct summary of the changes between versions that can be read by users. For our first release, we might have:

```
## v1.0.0

- Initial public release of our glorious hello world
  project (@avsm)
- Added test cases for making sure we do in fact hello world.
```

Commit this file to your repository in the root. Before you proceed with a release, you need to make sure that all of your local changes have been pushed to the remote GitHub repository, and that your working tree is clean. You can do this by using git:

```
$ git clean -dxf
$ git diff
```

This will remove any untracked files from the local checkout (such as the `_build` directory) and check that tracked files are unmodified. We should now be ready to perform the release! First create a git tag to mark this release:

```
$ dune-release tag
```

This will parse your `CHANGES.md` file and figure out the latest version, and create a local git tag in your repository after prompting you. Once that succeeds, you can start the release process via:

```
$ dune-release
```

This will begin an interactive session where you will need to enter some GitHub authentication details (via creating a personal access token). Once that is completed, the tool will run all local tests, generate documentation and upload it to your GitHub pages branch for that project, and finally offer to open a pull request to the central opam-repository. Recall that the central opam package set is all just a normal git repository, and so your opam file will be added to that and your GitHub account will create a PR.

At this point, you can sit back and relax while the central opam repository test system runs your package through a battery of installations (including on exotic architectures you might not access to, such as S390X mainframes or 32-bit ARMv7). If there is a problem detected, some friendly maintainers from the OCaml community will comment on the pull request and guide you through how to address it. You can simply delete the git tag and re-run the release process until the package is merged. Once it is merged, you can navigate to the <ocaml.org> site and view it online in an hour or so. It will also be available in the central repository for other users to install.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

Creating Lock Files for Your Projects

Before you publish a project, you might also want to create an opam lock file to include with the archive. A lock file records the exact versions of all the transitive opam dependencies at the time you generate it. All you need to do is to run:

```
$ opam lock
```

This generates a `pkgname.opam.locked` file which contains the same metadata as your original file, but with all the dependencies explicitly listed. Later on, if a user wants to reconstruct your exact opam environment (as opposed to the package solution they might calculate with a future opam repository), then they can pass an option during installation:

```
$ opam install pkgname --locked
$ opam switch create . --locked
```

Lock files are an optional but useful step to take when releasing your project to the Internet.

22.4 Learning More from Real Projects

There's a lot more customization that happens in any real project, and we can't cover every aspect in this book. The best way by far to learn more is to dive in and compile an already established project, and perhaps even contribute to it. There are thousands of libraries and executable projects released on the opam repository which you can find online at https://ocaml.org.

A selection of some include:

- `patdiff` is an OCaml implementation of the patience diff algorithm, and is a nice self-contained CLI project using Core. https://github.com/janestreet/patdiff
- The source code to this book is published as a self-contained monorepo with all the dependencies bundled together, for convenient compilation. https://github.com/realworldocaml/book
- Flow is a static typechecker for JavaScript written in OCaml that uses Base and works on macOS, Windows and Linux. It's a good example of a large, cross-platform CLI-driven tool. https://github.com/facebook/flow
- Octez is an OCaml implementation of a proof-of-stake blockchain called Tezos, which contains a comprehensive collection[9] of libraries such as interpreters for a stack language, and a shell that uses Lwt to provide networking, storage and cryptographic communications to the outside world. https://gitlab.com/tezos/tezos
- MirageOS is a library operating system written in OCaml, that can compile code to a

[9] https://tezos.gitlab.io/shell/the_big_picture.html#packages

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press

variety of embedded and hypervisor targets. There are 100s of libraries all written using dune in a variety of ways available at https://github.com/mirage.

- You can find a number of standalone OCaml libraries for unicode, parsing and computer graphics and OS interaction over at https://erratique.ch/software.

https://doi.org/10.1017/9781009129220.025 Published online by Cambridge University Press