# jbuilder Documentation

***Release***

**Jérémie Dimino**

**Sep 22, 2017**

# Contents

CHAPTER 1

# Quickstart

This document gives simple usage examples of Jbuilder. You can also look at examples for complete examples of projects using Jbuilder.

## Building a hello world program

In a directory of your choice, write this `jbuild` file:

```
(jbuild_version 1)

;; This declare the hello_world executable implemented by hello_world.ml
(executable
 ((name hello_world)))
```

This `hello_world.ml` file:

```
print_endline "Hello, world!"
```

And build it with:

```
jbuilder build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

## Building a hello world program using Lwt

In a directory of your choice, write this `jbuild` file:

```
(jbuild_version 1)

(executable
```

```
 ((name hello_world)
  (libraries (lwt.unix))))
```

This `hello_world.ml` file:

```
Lwt_main.run (Lwt_io.printf "Hello, world!\n")
```

And build it with:

```
jbuilder build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

## Building a hello world program using Core and Jane Street PPXs

Write this jbuild:

```
(jbuild_version 1)

(executable
 ((name hello_world)
  (libraries (core))
  (preprocess (pps (ppx_jane)))
 ))
```

This `hello_world.ml` file:

```
open Core

let () =
  Sexp.to_string_hum [%sexp ([3;4;5] : int list)]
  |> print_endline
```

And build it with:

```
jbuilder build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

## Defining a library using Lwt and ocaml-re

Write this jbuild:

```
(jbuild_version 1)

(library
 ((name        mylib)
  (public_name mylib)
  (libraries (re lwt))))
```

The library will be composed of all the modules in the same directory. Outside of the library, module `Foo` will be accessible as `Mylib.Foo`, unless you write an explicit `mylib.ml` file.

You can then use this library in any other directory by adding `mylib` to the `(libraries ...)` field.

## Using cppo

Add this field to your `library` or `executable` stanzas:

```
(preprocess (action (run ${bin:cppo} -V OCAML:${ocaml_version} ${<})))
```

Additionally, if you are include a `config.h` file, you need to declare the dependency to this file via:

```
(preprocessor_deps (config.h))
```

### Using the .cppo.ml style like the ocamlbuild plugin

Write this in your jbuild:

```
(rule
 ((targets (foo.ml))
  (deps    (foo.cppo.ml <other files that foo.ml includes>))
  (action  (run ${bin:cppo} ${<} -o ${@}))))
```

## Defining a library with C stubs

Assuming you have a file called `mystubs.c`, that you need to pass `-I/blah/include` to compile it and `-lblah` at link time, write this jbuild:

```
(jbuild_version 1)

(library
 ((name            mylib)
  (public_name     mylib)
  (libraries       (re lwt))
  (c_names         (mystubs)
  (c_flags         (-I/blah/include))
  (c_library_flags (-lblah)))))
```

## Defining a library with C stubs using pkg-config

Same context as before, but using `pkg-config` to query the compilation and link flags. Write this jbuild:

```
(jbuild_version 1)

(library
 ((name            mylib)
  (public_name     mylib)
  (libraries       (re lwt))
  (c_names         (mystubs))
  (c_flags         (:include c_flags.sexp))
  (c_library_flags (:include c_library_flags.sexp))))

(rule
 ((targets (c_flags.sexp
            c_library_flags.sexp))
```

```
  (deps    (config/discover.exe))
  (action  (run ${<} -ocamlc ${OCAMLC})))))
```

Then create a `config` subdirectory and write this `jbuild`:

```
(jbuild_version 1)

(executable
 ((name discover)
  (libraries (base stdio configurator))))
```

as well as this `discover.ml` file:

```
open Base
open Stdio
module C = Configurator

let write_sexp fn sexp =
  Out_channel.write_all fn ~data:(Sexp.to_string sexp)

let () =
  C.main ~name:"mylib" (fun c ->
    let default : C.Pkg_config.package_conf =
      { libs   = ["-lblah"]
      ; cflags = []
      }
    in
    let conf =
      match C.Pkg_config.get c with
      | None -> default
      | Some pc ->
        Option.value (C.Pkg_config.query pc ~package:"blah") ~default
    in

    write_sexp "c_flags.sexp"         (sexp_of_list sexp_of_string conf.cflags);
    write_sexp "c_library_flags.sexp" (sexp_of_list sexp_of_string conf.libs))
```

## Using a custom code generator

To generate a file `foo.ml` using a program from another directory:

```
(jbuild_version 1)

(rule
 ((targets (foo.ml))
  (deps    (../generator/gen.exe))
  (action  (run ${<} -o ${@}))))
```

## Defining tests

Write this in your `jbuild` file:

```
(jbuild_version 1)

(alias
 ((name    runtest)
  (deps    (my-test-program.exe))
  (action  (run ${<}))))
```

And run the tests with:

```
jbuilder runtest
```

## Building a custom toplevel

A toplevel is simply an executable calling `Topmain.main ()` and linked with the compiler libraries and `-linkall`. Moreover, currently toplevels can only be built in bytecode.

As a result, write this in your `jbuild` file:

```
(jbuild_version 1)

(executable
 ((name       mytoplevel)
  (libraries  (compiler-libs.toplevel mylib))
  (link_flags (-linkall))
  (modes      (byte))))
```

And write this in `mytoplevel.ml`

```
let () = Topmain.main ()
```

CHAPTER 2

# Overview

Jbuilder is a build system for OCaml and Reason. It is not intended as a completely generic build system that is able to build any given project in any language. On the contrary, it makes lots of choices in order to encourage a consistent development style.

This scheme is inspired from the one used inside Jane Street and adapted to the opam world. It has matured over a long time and is used daily by hundred of developers, which means that it is highly tested and productive.

When using Jbuilder, you give very little and high-level information to the build system, which in turn takes care of all the low-level details, from the compilation of your libraries, executables and documentation, to the installation, setting up of tests, setting up of the development tools such as merlin, etc.

In addition to the normal features one would expect from a build system for OCaml, Jbuilder provides a few additional ones that detach it from the crowd:

- you never need to tell Jbuilder where things such as libraries are. Jbuilder will always discover them automatically. In particular, this means that when you want to re-organize your project you need to do no more than rename your directories, Jbuilder will do the rest
- things always work the same whether your dependencies are local or installed on the system. In particular, this means that you can always drop in the source for a dependency of your project in your working copy and Jbuilder will start using it immediately. This makes Jbuilder a great choice for multi-project development
- cross-platform: as long as your code is portable, Jbuilder will be able to cross-compile it (note that Jbuilder is designed internally to make this easy but the actual support is not implemented yet)
- release directly from any revision: Jbuilder needs no setup stage. To release your project, you can simply point to a specific tag. You can of course add some release steps if you want to, but it is not necessary

The first section of this document defines some terms used in the rest of this manual. The second section specifies the Jbuilder metadata format and the third one describes how to use the `jbuilder` command.

CHAPTER 3

# Terminology

- **package**: a package is a set of libraries, executables, ... that are built and installed as one by opam
- **project**: a project is a source tree, maybe containing one or more packages
- **root**: the root is the directory from where Jbuilder can build things. Jbuilder knows how to build targets that are descendents of the root. Anything outside of the tree starting from the root is considered part of the **installed world**. How the root is determined is explained in *Finding the root*.
- **workspace**: the workspace is the subtree starting from the root. It can contain any number of projects that will be built simultaneously by jbuilder
- **installed world**: anything outside of the workspace, that Jbuilder takes for granted and doesn't know how to build
- **installation**: this is the action of copying build artifacts or other files from the `<root>/_build` directory to the installed world
- **scope**: a scope determines where private items are visible. Private items include libraries or binaries that will not be installed. In Jbuilder, scopes are sub-trees rooted where at least one `<package>.opam` file is present. Moreover, scopes are exclusive. Typically, every project defines a single scope. See *Scopes* for more details
- **build context**: a build context is a subdirectory of the `<root>/_build` directory. It contains all the build artifacts of the workspace built against a specific configuration. Without specific configuration from the user, there is always a `default` build context, which corresponds to the environment in which Jbuilder is executed. Build contexts can be specified by writing a *jbuild-workspace* file
- **build context root**: the root of a build context named `foo` is `<root>/_build/<foo>`
- **alias**: an alias is a build target that doesn't produce any file and has configurable dependencies. Alias are per-directory and some are recursive; asking an alias to be built in a given directory will trigger the construction of the alias in all children directories recursively. The most interesting ones are:
  - `runtest` which runs user defined tests
  - `install` which depends on everything that should be installed
  - `doc` which depends on the generated HTML documentation. See apidoc for details

CHAPTER 4

# Project Layout and Metadata Specification

A typical jbuilder project will have one or more `<package>.opam` file at toplevel as well as `jbuild` files wherever interesting things are: libraries, executables, tests, documents to install, etc...

It is recommended to organize your project so that you have exactly one library per directory. You can have several executables in the same directory, as long as they share the same build configuration. If you'd like to have multiple executables with different configurations in the same directory, you will have to make an explicit module list for every executable using `modules`.

The next sections describe the format of Jbuilder metadata files.

Note that the Jbuilder metadata format is versioned in order to ensure forward compatibility. There is currently only one version available, but to be future proof, you should still specify it in your `jbuild` files. If no version is specified, the latest one will be used.

## Metadata format

Most configuration files read by Jbuilder are using the S-expression syntax, which is very simple. Everything is either an atom or a list. The exact specification of S-expressions is described in the documentation of the parsexp library.

In a nutshell, the syntax is as follows:

- atoms that do no contain special characters are simply written as is. For instance: `foo`, `bar` are valid atomic S-expressions
- atoms containing special characters or spaces must be quoted using the syntax `"..."`: `"foo bar\n"`
- lists are formed by surrounding a sequence of S-expressions separated by spaces with parentheses: `(a b (c d))`
- single-line comments are introduced with the `;` character and may appear anywhere except in the middle of a quoted atom
- block comment are enclosed by `#|` and `|#` and can be nested

Note that the format is completely static. However you can do meta-programming on jbuilds files by writing them in *OCaml syntax*.

# <package>.opam files

When a <package>.opam file is present, Jbuilder will know that the package named <package> exists. It will know how to construct a <package>.install file in the same directory to handle installation via opam. Jbuilder also defines the recursive install alias, which depends on all the buildable <package>.install files in the workspace. So for instance to build everything that is installable in a workspace, run at the root:

```
$ jbuilder build @install
```

Declaring a package this way will allow you to add elements such as libraries, executables, documentation, ... to your package by declaring them in jbuild files.

Such elements can only be declared in the scope defined by the corresponding <package>.opam file. Typically, your <package>.opam files should be at the root of your project, since this is where opam pin ... will look for them.

Note that <package> must be non-empty, so in particular .opam files are ignored.

## Scopes

Any directory containing at least one <package>.opam file defines a scope. This scope is the sub-tree starting from this directory, excluding any other scopes rooted in sub-direcotries.

Typically, any given project will define a single scope. Libraries and executables that are not meant to be installed will be visible inside this scope only.

Because scopes are exclusive, if you whish to include the dependencies of the project you are currently working on into your workspace, you may copy them in a vendor directory, or any other name of your choice. Jbuilder will look for them there rather than in the installed world and there will be no overlap between the various scopes.

## Package version

Note that Jbuilder will try to determine the version number of packages defined in the workspace. While Jbuilder itself makes no use of version numbers, it can be use by external tools such as ocamlfind.

Jbuilder determines the version of a package by first looking in the <package>.opam for a version variable. If not found, it will try to read the first line of a version file in the same directory as the <package>.opam file. The version file is any file whose name is, in order in which they are looked for:

- <package>.version
- version
- VERSION

The version file can be generated by a user rule.

If the version can't be determined, Jbuilder just won't assign one.

Note that if you are using Topkg as well in your project, you shouldn't manually set a version in your <package>.opam file or write/generate on of the file listed above. See the section about *Using topkg with jbuilder* for more details.

## Odig conventions

Jbuilder follows the odig conventions and automatically installs any README*, CHANGE*, HISTORY* and LICENSE* files in the same directory as the <package>.opam file to a location where odig will find them.

Note that this includes files present in the source tree as well as generated files. So for instance a changelog generated by a user rule will be automatically installed as well.

## jbuild-ignore

By default Jbuilder traverses the whole source tree, ignoring the following files and directories:

- any file that start with `.` `#`
- any directory that start with either `.` or `_`

To ignore a subtree, simply write a `jbuild-ignore` file in the parent directory containing the name of the sub-directories to ignore.

So for instance, if you write `foo` in `src/jbuild-ignore`, then `src/foo` won't be traversed and any `jbuild` file it contains will be ignored.

`jbuild-ignore` files contain a list of directory names, one per line.

CHAPTER 5

# jbuild specification

`jbuild` files are the main part of Jbuilder, and are the origin of its name. They are used to describe libraries, executables, tests, and everything Jbuilder needs to know about.

The syntax of `jbuild` files is described in :ref:`metadata-format` section.

## Stanzas

`jbuild` files are composed of stanzas. For instance a typical `jbuild` looks like:

```
(library
 ((name mylib)
  (libraries (base lwt))))

(rule
 ((targets (foo.ml))
  (deps    (generator/gen.exe))
  (action  (run ${<} -o ${@}))))
```

The following sections describe the available stanzas and their meaning.

### jbuild_version

`(jbuild_version 1)` specifies that we are using the version 1 of the Jbuilder metadata format in this `jbuild` file.

### library

The `library` stanza must be used to describe OCaml libraries. The format of library stanzas is as follows:

```
(library
  ((name <library-name>)
   <optional-fields>
  ))
```

`<library-name>` is the real name of the library. It determines the names of the archive files generated for the library as well as the module name under which the library will be available, unless `(wrapped false)` is used (see below). It must be a valid OCaml module name but doesn't need to start with a uppercase letter.

For instance, the modules of a library named `foo` will be available as `Foo.XXX` outside of `foo` itself. It is however allowed to write an explicit `Foo` module, in which case this will be the interface of the library and you are free to expose only the modules you want.

`<optional-fields>` are:

- `(public_name <name>)` this is the name under which the library can be referred to as a dependency when it is not part of the current workspace, i.e. when it is installed. Without a `(public_name ...)` field, the library will not be installed by Jbuilder. The public name must start by the package name it is part of and optionally followed by a dot and anything else you want. The package name must be one of the packages that Jbuilder knows about, as determined by the *<package>.opam files*
- `(synopsis <string>)` should give a one-line description of the library. This is used by tools that list installed libraries
- `(modules <modules>)` specifies what modules are part of the library. By default Jbuilder will use all the .ml/.re files in the same directory as the `jbuild` file. This include ones that are present in the file system as well as ones generated by user rules. You can restrict this list by using a `(modules <modules>)` field. `<modules>` uses the *Ordered set language* where elements are module names and don't need to start with a uppercase letter. For instance to exclude module `Foo`: `(modules (:standard \ foo))`
- `(libraries (<library-dependencies>))` is used to specify the dependencies of the library. See the section about *Library dependencies* for more details
- `(wrapped <boolean>)` specifies whether the modules of the library should be available only through the top-level library module, or should all be exposed at the top level. The default is `true` and it is highly recommended to keep it this way. Because OCaml top-level modules must all be unique when linking an executables, polluting the top-level namespace will make your library unusable with other libraries if there is a module name clash. This option is only intended for libraries that manually prefix all their modules by the library name and to ease porting of existing projects to Jbuilder
- `(preprocess <preprocess-spec>)` specifies how to preprocess files if needed. The default is `no_processing`. Other options are described in the *Preprocessing specification* section
- `(preprocessor_deps (<deps-conf list>))` specifies extra dependencies of the preprocessor, for instance if the preprocessor reads a generated file. The specification of dependencies is described in the *Dependency specification* section
- `(optional)`, if present it indicates that the library should only be built and installed if all the dependencies are available, either in the workspace or in the installed world. You can use this to provide extra features without adding hard dependencies to your project
- `(c_names (<names>))`, if your library has stubs, you must list the C files in this field, without the `.c` extension
- `(cxx_names (<names>))` is the same as `c_names` but for C++ stubs
- `(install_c_headers (<names>))`, if your library has public C header files that must be installed, you must list them in this field, with the `.h` extension
- `(modes (<modes>))` modes (`byte` and `native`) which should be built by default. This is only useful when writing libraries for the OCaml toplevel

- `(no_dynlink)` is to disable dynamic linking of the library. This is for advanced use only, by default you shouldn't set this option
- `(kind <kind>)` is the kind of the library. The default is `normal`, other available choices are `ppx_rewriter` and `ppx_deriver` and must be set when the library is intended to be used as a ppx rewriter or a `[@@deriving ...]` plugin. The reason why `ppx_rewriter` and `ppx_deriver` are split is historical and hopefully we won't need two options soon
- `(ppx_runtime_libraries (<library-names>))` is for when the library is a ppx rewriter or a `[@@deriving ...]` plugin and has runtime dependencies. You need to specify these runtime dependencies here
- `(virtual_deps (<opam-packages>)`. Sometimes opam packages enable a specific feature only if another package is installed. This is for instance the case of `ctypes` which will only install `ctypes.foreign` if the dummy `ctypes-foreign` package is installed. You can specify such virtual dependencies here. You don't need to do so unless you use Jbuilder to synthesize the `depends` and `depopts` sections of your opam file
- `js_of_ocaml`. See the section about *js_of_ocaml*
- `flags`, `ocamlc_flags` and `ocamlopt_flags`. See the section about *OCaml flags*
- `(library_flags (<flags>))` is a list of flags that are passed as it to `ocamlc` and `ocamlopt` when building the library archive files. You can use this to specify `-linkall` for instance. `<flags>` is a list of strings supporting *Variables expansion*
- `(c_flags <flags>)` specifies the compilation flags for C stubs, using the *Ordered set language*. This field supports `(:include ...)` forms
- `(cxx_flags <flags>)` is the same as `c_flags` but for C++ stubs
- `(c_library_flags <flags>)` specifies the flags to pass to the C compiler when constructing the library archive file for the C stubs. `<flags>` uses the *Ordered set language* and supports `(:include ...)` forms. When you are writing bindings for a C library named `bar`, you should typically write `-lbar` here, or whatever flags are necessary to to link against this library
- `(self_build_stubs_archive <c-libname>)` indicates to Jbuilder that the library has stubs, but that the stubs are built manually. The aim of the field is to embed a library written in foreign language and/or building with another build system. It is not for casual uses, see the re2 library for an example of use

Note that when binding C libraries, Jbuilder doesn't provide special support for tools such as `pkg-config`, however it integrates easily with configurator by using `(c_flags (:include ...))` and `(c_library_flags (:include ...))`.

## executable

The `executable` stanza must be used to describe an executable. The format of executable stanzas is as follows:

```
(executable
  ((name <name>)
   <optional-fields>
  ))
```

`<name>` is a module name that contains the main entry point of the executable. There can be additional modules in the current directory, you only need to specify the entry point. Given an `executable` stanza with `(name <name>)`, Jbuilder will know how to build `<name>.exe`, `<name>.bc` and `<name>.bc.js`. `<name>.exe` is a native code executable, `<name>.bc` is a bytecode executable which requires `ocamlrun` to run and `<name>.bc.js` is a JavaScript generated using js_of_ocaml.

Note that in case native compilation is not available, `<name>.exe` will in fact be a custom byte-code executable. Custom in the sense of `ocamlc -custom`, meaning that it is a native executable that embeds the `ocamlrun` virtual machine as well as the byte code. As such you can always rely on `<name>.exe` being available. Moreover, it is usually preferable to use `<name>.exe` in custom rules or when calling the executable by hand. This is because running a byte-code executable often requires loading shared libraries that are locally built, and so requires additional setup such as setting specific environment variables and jbuilder doesn't do at the moment.

Native compilation is considered not available when there is no `ocamlopt` binary at the same place as where `ocamlc` was found, or when there is a `(modes (...))` field not listing `native`.

`<optional-fields>` are:

- `(public_name <public-name>)` specifies that the executable should be installed under that name. It is the same as adding the following stanza to your `jbuild` file:

```
(install
 ((section bin)
  (files ((<name>.exe as <public-name>)))))
```

- `(package <package>)` if there is a `(public_name ...)` field, this specifies the package the executables are part of
- `(libraries (<library-dependencies>))` specifies the library dependencies. See the section about *Library dependencies* for more details
- `(link_flags <flags>)` specifies additional flags to pass to the linker. This field supports `(:include ...)` forms
- `(modules <modules>)` specifies which modules in the current directory Jbuilder should consider when building this executable. Modules not listed here will be ignored and cannot be used inside the executable described by the current stanza. It is interpreted in the same way as the `(modules ...)` field of *library*
- `(modes (<modes>))` modes (`byte` and `native`) which should be built by default. If the stanza has a `(public_name ...)` field and `native` is not listed here, the byte-code version will be installed instead.
- `(preprocess <preprocess-spec>)` is the same as the `(preprocess ...)` field of *library*
- `(preprocessor_deps (<deps-conf list>))` is the same as the `(preprocessor_deps ...)` field of *library*
- `js_of_ocaml`. See the section about *js_of_ocaml*
- **`flags`, `ocamlc_flags` and `ocamlopt_flags`. See the section about** specifying *OCaml flags*

## executables

The `executables` stanza is the same as the `executable` stanza, except that it is used to describe several executables sharing the same configuration.

It shares the same fields as the `executable` stanza, except that instead of `(name ...)` and `(public_name ...)` you must use:

- `(names (<names>))` where `<names>` is a list of entry point names. As for `executable` you only need to specify the modules containing the entry point of each executable
- `(public_names (<names>))` describes under what name each executable should be installed. The list of names must be of the same length as the list in the `(names ...)` field. Moreover you can use `-` for executables that shouldn't be installed

## rule

The `rule` stanza is used to create custom user rules. It tells Jbuilder how to generate a specific set of files from a specific set of dependencies.

The syntax is as follows:

```
(rule
  ((targets (<filenames>))
   (action  <action>)
   <optional-fields>))
```

`<filenames>` is a list of file names. Note that currently Jbuilder only support user rules with targets in the current directory.

`<action>` is the action to run to produce the targets from the dependencies. See the *User actions* section for more details.

`<optional-fields>` are:

- `(deps (<deps-conf list>))` to specify the dependencies of the rule. See the *Dependency specification* section for more details.
- `(fallback)` to specify that this is a fallback rule. A fallback rule means that if the targets are already present in the source tree, jbuilder will ignore the rule. It is an error if only a subset of the targets are present in the tree. The common use of fallback rules is to generate default configuration files that may be generated by a configure script.

Note that contrary to makefiles or other build systems, user rules currently don't support patterns, such as a rule to produce `%.y` from `%.x` for any given `%`. This might be supported in the future.

### inferred rules

When using the action DSL (see *User actions*), it is most of the time obvious what are the dependencies and targets.

For instance:

```
(rule
  ((targets (b)
   (deps    (a)
   (action  (copy ${<} ${@})))))))
```

In this example it is obvious by inspecting the action what the dependencies and targets are. When this is the case you can use the following shorter syntax, where Jbuilder infers dependencies and targets for you:

```
(rule <action>)
```

For instance:

```
(rule (copy a b))
```

Note that in Jbuilder, targets must always be known statically. Especially, this mean that Jbuilder must be able to statically determine all targets. For instance, this `(rule ...)` stanza is rejected by Jbuilder:

```
(rule (copy a b.${read:file}))
```

## ocamllex

`(ocamllex (<names>))` is essentially a shorthand for:

```
(rule
  ((targets (<name>.ml))
   (deps    (<name>.mll))
   (action  (chdir ${ROOT} (run ${bin:ocamllex} -q -o ${<})))))
```

## ocamlyacc

`(ocamlyacc (<names>))` is essentially a shorthand for:

```
(rule
  ((targets (<name>.ml <name>.mli))
   (deps    (<name>.mly))
   (action  (chdir ${ROOT} (run ${bin:ocamlyacc} ${<})))))
```

## menhir

The basic form for defining menhir parsers (analogous to ocamlyacc) is:

```
(menhir
 ((modules (<parser1> <parser2> ...))))
```

Modular parsers can be defined by adding a `merge_into` field. This correspond to the `--base` command line option of `menhir`. With this option, a single parser named `base_name` is generated.

```
(menhir
 ((merge_into <base_name>)
  (modules (<parser1> <parser2> ...))))
```

Extra flags can be passed to menhir using the `flags` flag:

```
(menhir
 ((flags (<option1> <option2> ...))
  (modules (<parser1> <parser2> ...))))
```

## alias

The `alias` stanza lets you add dependencies to an alias, or specify an action to run to construct the alias.

The syntax is as follows:

```
(alias
  ((name    <alias-name>)
   (deps    (<deps-conf list>))
   <optional-fields>
   ))
```

`<name>` is an alias name such as `runtest`.

`<deps-conf list>` specifies the dependencies of the alias. See the *Dependency specification* section for more details.

<optional-fields> are:

- <action>, an action to run when constructing the alias. See the *User actions* section for more details.
- (package <name>) indicates that this alias stanza is part of package <name> and should be filtered out if <name> is filtered out from the command line, either with --only-packages <pkgs> or -p <pkgs>

The typical use of the alias stanza is to define tests:

```
(alias
  ((name    runtest)
   (action (run ${exe:my-test-program.exe} blah))))
```

See the section about *Running tests* for details.

Note that if your project contains several packages and you run test the tests from the opam file using a build-test field, then all your runtest alias stanzas should have a (package ...) field in order to partition the set of tests.

## install

The install stanza is what lets you describe what Jbuilder should install, either when running jbuilder install or through opam.

Libraries and executables don't need an install stanza to be installed, just a public_name field. Everything else needs an install stanza.

The syntax is as follows:

```
(install
  ((section <section>)
   (files   (<filenames>))
   <optional-fields>
  ))
```

<section> is the installation section, as described in the opam manual. The following sections are available:

- lib
- libexec
- bin
- sbin
- toplevel
- share
- share_root
- etc
- doc
- stublibs
- man
- misc

<files> is the list of files to install. Each element in the list must be either a literal filename or a S-expression of the form:

```
(<filename> as <destination>)
```

where <destination> describe how the file will be installed. For instance, to install a file mylib.el as emacs/site-lisp/mylib.el in the share_root section:

```
(install
  ((section share_root) (files ((mylib.el as emacs/site-lisp/mylib.el)))))
```

<optional-fields> are:

- (package <name>). If there are no ambiguities, you can omit this field. Otherwise you need it to specify which package these files are part of. The package is not ambiguous when the first parent directory to contain a <package>.opam file contains exactly one <package>.opam file

#### Handling of the .exe extension on Windows

Under Microsoft Windows, executables must be suffixed with .exe. Jbuilder tries to make sure that executables are always installed with this extension on Windows.

More precisely, when installing a file via an (install ...) stanza, if the source file has extension .exe or .bc, then Jbuilder implicitly adds the .exe extension to the destination, if not already present.

### copy_files

The copy_files and copy_files# stanzas allow to specify that files from another directory could be copied if needed to the current directory.

The syntax is as follows:

```
(copy_files <glob>)
```

<glob> represents the set of files to copy, see the *glob* for details.

The difference between copy_files and copy_files# is the same as the difference between the copy and copy# action. See the *User actions* section for more details.

## Common items

### Ordered set language

A few fields takes as argument an ordered set and can be specified using a small DSL.

This DSL is interpreted by jbuilder into an ordered set of strings using the following rules:

- :standard denotes the standard value of the field when it is absent
- an atom not starting with a : is a singleton containing only this atom
- a list of sets is the concatenation of its inner sets
- (<sets1> \ <sets2>) is the set composed of elements of <sets1> that do not appear in <sets2>

In addition, some fields support the inclusion of an external file using the syntax (:include <filename>). This is useful for instance when you need to run a script to figure out some compilation flags. <filename> is expected to contain a single S-expression and cannot contain (:include ...) forms.

Most fields using the ordered set language also support *Variables expansion*. Variables are expanded after the set language is interpreted.

## Variables expansion

Some fields can contains variables of the form `$(var)` or `${var}` that are expanded by Jbuilder.

Jbuilder supports the following variables:

- `ROOT` is the relative path to the root of the build context. Note that `ROOT` depends on the worksace configuration. As such you shouldn't use `ROOT` to denote the root of your project. Use `SCOPE_ROOT` instead for this purpose
- `SCOPE_ROOT` is the root of the current scope. It is typically the toplevel directory of your project and as long as you have at least one `<package>.opam` file there, `SCOPE_ROOT` is independant of the workspace configuration
- `CC` is the C compiler command line being used in the current build context
- `CXX` is the C++ compiler command line being used in the current build context
- `ocaml_bin` is the path where `ocamlc` lives
- `OCAML` is the `ocaml` binary
- `OCAMLC` is the `ocamlc` binary
- `OCAMLOPT` is the `ocamlopt` binary
- `ocaml_version` is the version of the compiler used in the current build context
- `ocaml_where` is the output of `ocamlc -where`
- `ARCH_SIXTYFOUR` is `true` if using a compiler targeting a 64 bit architecture and `false` otherwise
- `null` is `/dev/null` on Unix or `nul` on Windows

In addition, `(action ...)` fields support the following special variables:

- `@` expands to the list of target
- `<` expands to the first dependency, or the empty string if there are no dependencies
- `^` expands to the list of dependencies, separated by spaces
- `path:<path>` expands to `<path>`
- `exe:<path>` is the same as `<path>`, except when cross-compiling, in which case it will expand to `<path>` from the host build context
- `bin:<program>` expands to a path to `program`. If `program` is installed by a package in the workspace (see *install* stanzas), the locally built binary will be used, otherwise it will be searched in the `PATH` of the current build context. Note that `(run ${bin:program} ...)` and `(run program ...)` behave in the same way. `${bin:...}` is only necessary when you are using `(bash ...)` or `(system ...)`
- `lib:<public-library-name>:<file>` expands to a path to file `<file>` of library `<public-library-name>`. If `<public-library-name>` is available in the current workspace, the local file will be used, otherwise the one from the installed world will be used
- `libexec:<public-library-name>:<file>` is the same as `lib:...` except when cross-compiling, in which case it will expand to the file from the host build context
- `lib-available:<library-name>` expands to `true` or `false` depending on wether the library is available or not. A library is available iff at least one of the following condition holds:
  - it is part the installed worlds

  - it is available locally and is not optional
  - it is available locally and all its library dependencies are available

- `version:<package>` expands to the version of the given package. Note that this is only supported for packages that are being defined in the current scope
- `read:<path>` expands to the contents of the given file
- `read-lines:<path>` expands to the list of lines in the given file
- `read-strings:<path>` expands to the list of lines in the given file, unescaped using OCaml lexical convention

The `${<kind>:...}` forms are what allows you to write custom rules that work transparently whether things are installed or not.

Note that aliases are ignored by both `${<}` and `${^}`.

The intent of this last form is to reliably read a list of strings generated by an OCaml program via:

```
List.iter (fun s -> print_string (String.escaped s)) l
```

1. Expansion of lists

Forms that expands to list of items, such as `${^}`, `${@}` or `${read-lines:...}` will always expand to a single string where elements are separated by spaces. Inside `(run <prog> <arguments>)` forms you can however split the items as several arguments by prefixing the variable with `!`. Such forms can only be used as a whole atom, i.e. they can't be used inside a quoted atom.

For instance in:

```
(run foo ${^})
```

even if there are two dependencies `a` and `b`, the produced command will be equivalent to the shell command:

```
$ foo "a b"
```

However, if you replace `${^}` by `${!^}` in the previous example the command produced would be equivalent to this shell command:

```
$ foo "a" "b"
```

You can also use `${!^}` as program name, for instance:

```
(rule
 ((targets (result.txt))
  (deps    (foo.exe (glob_files *.txt)))
  (action  (run ${!^}))))
```

## Library dependencies

Dependencies on libraries are specified using `(libraries ...)` fields in `library` and `executables` stanzas.

For libraries defined in the current scope, you can use either the real name or the public name. For libraries that are part of the installed world, or for libraries that are part of the current workspace but in another scope, you need to use the public name. For instance: `(libraries (base re))`.

When resolving libraries, libraries that are part of the workspace are always prefered to ones that are part of the installed world.

### Alternative dependencies

In addition to direct dependencies you can specify alternative dependencies. This is described in the *Alternative dependencies* section

It is sometimes the case that one wants to not depend on a specific library, but instead on whatever is already installed. For instance to use a different backend depending on the target.

Jbuilder allows this by using a `(select ... from ...)` form inside the list of library dependencies.

Select forms are specified as follows:

```
(select <target-filename> from
  (<literals> -> <filename>)
  (<literals> -> <filename>)
   ...)
```

`<literals>` are lists of literals, where each literal is one of:

- `<library-name>`, which will evaluate to true if `<library-name>` is available, either in the workspace or in the installed world
- `!<library-name>`, which will evaluate to true if `<library-name>` is not available in the workspace or in the installed world

When evaluating a select form, Jbuilder will create `<target-filename>` by copying the file given by the first `(<literals> -> <filename>)` case where all the literals evaluate to true. It is an error if none of the clauses are selectable. You can add a fallback by adding a clause of the form `(-> <file>)` at the end of the list.

## Preprocessing specification

Jbuilder accepts three kinds of preprocessing:

- `no_preprocessing`, meaning that files are given as it to the compiler, this is the default
- `(action <action>)` to preprocess files using the given action
- `(pps (<ppx-rewriters-and-flags>))` to preprocess files using the given list of ppx rewriters

Note that in any cases, files are preprocessed only once. Jbuilder doesn't use the `-pp` or `-ppx` of the various OCaml tools.

### Preprocessing with actions

`<action>` uses the same DSL as described in the *User actions* section, and for the same reason given in that section, it will be executed from the root of the current build context. It is expected to be an action that reads the file given as only dependency and outputs the preprocessed file on its standard output.

More precisely, `(preprocess (action <action>))` acts as if you had setup a rule for every file of the form:

```
(rule
 ((targets (file.pp.ml))
  (deps    (file.ml))
  (action  (with-stdout-to ${@} (chdir ${ROOT} <action>)))))
```

The equivalent of a `-pp <command>` option passed to the OCaml compiler is `(system "<command> ${<}")`.

### Preprocessing with ppx rewriters

`<ppx-rewriters-and-flags>` is expected to be a list where each element is either a command line flag if starting with a `-` or the name of a library. Additionnally, any sub-list will be treated as a list of command line arguments. So for instance from the following `preprocess` field:

```
(preprocess (pps (ppx1 -foo ppx2 (-bar 42))))
```

The list of libraries will be `ppx1` and `ppx2` and the command line arguments will be: `-foo -bar 42`.

Libraries listed here should be libraries implementing an OCaml AST rewriter and registering themselves using the ocaml-migrate-parsetree.driver API.

Jbuilder will build a single executable by linking all these libraries and their dependencies. Note that it is important that all these libraries are linked with `-linkall`. Jbuilder automatically uses `-linkall` when the `(kind ...)` field is set to `ppx_rewriter` or `ppx_deriver`.

It is guaranteed that the last library in the list will be linked last. You can use this feature to use a custom ppx driver. By default Jbuilder will use `ocaml-migrate-parsetree.driver-main`. See the section about *Using a custom ppx driver* for more details.

### Per module preprocessing specification

By default a preprocessing specification will apply to all modules in the library/set of executables. It is possible to select the preprocessing on a module-by-module basis by using the following syntax:

```
(preprocess (per_module
               (<spec1> (<module-list1>))
               (<spec2> (<module-list2>))
               ...))
```

Where `<spec1>`, `<spec2>`, ... are preprocessing specifications and `<module-list1>`, `<module-list2>`, ... are list of module names.

For instance:

```
(preprocess (per_module
               (((action (run ./pp.sh X=1 ${<})) (foo bar)))
               (((action (run ./pp.sh X=2 ${<})) (baz)))))
```

## Dependency specification

Dependencies in `jbuild` files can be specified using one of the following syntax:

- `(file <filename>)` or simply `<filename>`: depend on this file
- `(alias <alias-name>)`: depend on the construction of this alias, for instance: `(alias src/runtest)`
- `(glob_files <glob>)`: depend on all files matched by `<glob>`, see the *glob* for details
- `(files_recursively_in <dir>)`: depend on all files in the subtree with root `<dir>`

In all these cases, the argument supports *Variables expansion*.

### Glob

You can use globs to declare dependencies on a set of files. Note that globs will match files that exist in the source tree as well as buildable targets, so for instance you can depend on `*.cmi`.

Currently jbuilder only support globbing files in a single directory. And in particular the glob is interpreted as follows:

- anything before the last `/` is taken as a literal path
- anything after the last `/`, or everything if the glob contains no `/`, is interpreted using the glob syntax

The glob syntax is interpreted as follows:

- `\<char>` matches exactly `<char>`, even if it is a special character (`*`, `?`, ...)
- `*` matches any sequence of characters, except if it comes first in which case it matches any character that is not `.` followed by anything
- `**` matches any character that is not `.` followed by anything, except if it comes first in which case it matches anything
- `?` matches any single character
- `[<set>]` matches any character that is part of `<set>`
- `[!<set>]` matches any character that is not part of `<set>`
- `{<glob1>,<glob2>,...,<globn>}` matches any string that is matched by one of `<glob1>`, `<glob2>`, ...

## OCaml flags

In `library` and `executables` stanzas, you can specify OCaml compilation flags using the following fields:

- `(flags <flags>)` to specify flags passed to both `ocamlc` and `ocamlopt`
- `(ocamlc_flags <flags>)` to specify flags passed to `ocamlc` only
- `(ocamlopt_flags <flags>)` to specify flags passed to `ocamlopt` only

For all these fields, `<flags>` is specified in the *Ordered set language*. These fields all support `(:include ...)` forms.

The default value for `(flags ...)` includes some `-w` options to set warnings. The exact set depends on whether `--dev` is passed to Jbuilder. As a result it is recommended to write `(flags ...)` fields as follows:

```
(flags (:standard <my options>))
```

## js_of_ocaml

In `library` and `executables` stanzas, you can specify js_of_ocaml options using `(js_of_ocaml (<js_of_ocaml-options>))`.

`<js_of_ocaml-options>` are all optional:

- `(flags <flags>)` to specify flags passed to `js_of_ocaml`. This field supports `(:include ...)` forms
- `(javascript_files (<files-list>))` to specify `js_of_ocaml` JavaScript runtime files.

=<flags>= is specified in the *Ordered set language*.

The default value for `(flags ...)` depends on whether `--dev` is passed to Jbuilder. `--dev` will enable sourcemap and the pretty JavaScript output.

## User actions

`(action ...)` fields describe user actions.

User actions are always run from the same subdirectory of the current build context as the jbuild they are defined in. So for instance an action defined in `src/foo/jbuild` will be run from `_build/<context>/src/foo`.

The argument of `(action ...)` fields is a small DSL that is interpreted by jbuilder directly and doesn't require an external shell. All atoms in the DSL support *Variables expansion*. Moreover, you don't need to specify dependencies explicitly for the special `${<kind>:...}` forms, these are recognized and automatically handled by Jbuilder.

The DSL is currently quite limited, so if you want to do something complicated it is recommended to write a small OCaml program and use the DSL to invoke it. You can use shexp to write portable scripts or configurator for configuration related tasks.

The following constructions are available:

- `(run <prog> <args>)` to execute a program. `<prog>` is resolved locally if it is available in the current workspace, otherwise it is resolved using the `PATH`
- `(chdir <dir> <DSL>)` to change the current directory
- `(setenv <var> <value> <DSL>)` to set an environment variable
- `(with-<outputs>-to <file> <DSL>)` to redirect the output to a file, where `<outputs>` is one of: `stdout`, `stderr` or `outputs` (for both `stdout` and `stderr`)
- `(ignore-<outputs> <DSL)` to ignore the output, where `<outputs>` is one of: `stdout`, `stderr` or `outputs`
- `(progn <DSL>...)` to execute several commands in sequence
- `(echo <string>)` to output a string on stdout
- `(write-file <file> <string>)` writes `<string>` to `<file>`
- `(cat <file>)` to print the contents of a file to stdout
- `(copy <src> <dst>)` to copy a file
- `(copy# <src> <dst>)` to copy a file and add a line directive at the beginning
- `(system <cmd>)` to execute a command using the system shell: `sh` on Unix and `cmd` on Windows
- `(bash <cmd>)` to execute a command using `/bin/bash`. This is obviously not very portable

As mentioned `copy#` inserts a line directive at the beginning of the destination file. More precisely, it inserts the following line:

```
# 1 "<source file name>"
```

Most languages recognize such lines and update their current location, in order to report errors in the original file rather than the copy. This is important as the copy exists only under the `_build` directory and in order for editors to jump to errors when parsing the output of the build system, errors must point to files that exist in the source tree. In the beta versions of jbuilder, `copy#` was called `copy-and-add-line-directive`. However, most of time one wants this behavior rather than a bare copy, so it was renamed to something shorter.

Note: expansion of the special `${<kind>:...}` is done relative to the current working directory of the part of the DSL being executed. So for instance if you have this action in a `src/foo/jbuild`:

```
(action (chdir ../../.. (echo ${path:jbuild})))
```

Then `${path:jbuild}` will expand to `src/foo/jbuild`. When you run various tools, they often use the filename given on the command line in error messages. As a result, if you execute the command from the original directory, it will only see the basename.

To understand why this is important, let's consider this jbuild living in `src/foo`:

```
(rule
 ((targets (blah.ml))
  (deps    (blah.mll))
  (action  (run ocamllex -o ${@} ${<}))))
```

Here the command that will be executed is:

```
ocamllex -o blah.ml blah.mll
```

And it will be executed in `_build/<context>/src/foo`. As a result, if there is an error in the generated `blah.ml` file it will be reported as:

```
File "blah.ml", line 42, characters 5-10:
Error: ...
```

Which can be a problem as you editor might think that `blah.ml` is at the root of your project. What you should write instead is:

```
(rule
 ((targets (blah.ml))
  (deps    (blah.mll))
  (action  (chdir ${ROOT} (run ocamllex -o ${@} ${<})))))
```

## OCaml syntax

If a `jbuild` file starts with `(* -*- tuareg -*- *)`, then it is interpreted as an OCaml script that generates the `jbuild` file as described in the rest of this section. The code in the script will have access to a Jbuild_plugin module containing details about the build context it is executed in.

The OCaml syntax gives you an escape hatch for when the S-expression syntax is not enough. It is not clear whether the OCaml syntax will be supported in the long term as it doesn't work well with incremental builds. It is possible that it will be replaced by just an `include` stanza where one can include a generated file.

Consequently **you must not** build complex systems based on it.

CHAPTER 6

# API documentation

Jbuilder supports generating API documentation for libraries using the odoc tool in HTML format.

For this to work you need to have odoc installed and have documentation comments in your OCaml source files following the syntax described in the the section `Text formatting` of the OCaml manual.

## Generated pages

Jbuilder stores the generated HTML pages in `_build/<context>/_doc`. It creates one sub-directory per public library and generates an `index.html` file in each sub-directory.

The documentation is never installed on the system by Jbuilder. It is meant to be read locally while developing and then published on the www when releasing packages.

## Building the documentation

To build the documentaion, you can simply use the `doc` alias, which depends on the generated HTML pages for all the public libraries.

For instance:

```
$ jbuilder build @doc
```

## Custom library indexes

If the directory where a library lives contains a file named `<lib-name>.mld`, Jbuilder will generate the library index from this file. `<lib-name>` is what you put in the `(name ...)` field of the library's jbuild file.

Such a file must contains text using the same syntax as ocamldoc comments.

CHAPTER 7

# Usage

This section describe usage of Jbuilder from the shell.

## Finding the root

### jbuild-workspace

The root of the current workspace is determined by looking up a `jbuild-workspace` file in the current directory and parent directories.

`jbuilder` prints out the root when starting if it is not the current directory:

```
$ jbuilder runtest
Entering directory '/home/jdimino/code/jbuilder'
...
```

More precisely, it will choose the outermost ancestor directory containing a `jbuild-workspace` file as root. For instance if you are in `/home/me/code/myproject/src`, then jbuilder will look for all these files in order:

- `/jbuild-workspace`
- `/home/jbuild-workspace`
- `/home/me/jbuild-workspace`
- `/home/me/code/jbuild-workspace`
- `/home/me/code/myproject/jbuild-workspace`
- `/home/me/code/myproject/src/jbuild-workspace`

The first entry to match in this list will determine the root. In practice this means that if you nest your workspaces, Jbuilder will always use the outermost one.

In addition to determining the root, `jbuilder` will read this file as to setup the configuration of the workspace unless the `--workspace` command line option is used. See the section *Workspace configuration* for the syntax of this file.

### jbuild-workspace*

In addition to the previous rule, if no `jbuild-workspace` file is found, `jbuilder` will look for any file whose name starts with `jbuild-workspace` in ancestor directories. For instance `jbuild-workspace.dev`. If such a file is found, it will mark the root of the workspace. `jbuilder` will however not read its contents.

The rationale for this rule is that it is good practice to have a

`jbuild-workspace.dev` file at the root of your project.

For quick experiments, simply do this to mark the root:

```
$ touch jbuild-workspace.here
```

### Current directory

If none of the two previous rules appies, i.e. no ancestor directories have a file whose name starts with `jbuild-workspace`, then the current directory will be used as root.

### Forcing the root (for scripts)

You can pass the `--root` option to `jbuilder` to select the root explicitly. This option is intended for scripts to disable the automatic lookup.

Note that when using the `--root` option, targets given on the command line will be interpreted relative to the given root, not relative to the current directory as this is normally the case.

## Interpretation of targets

This section describes how `jbuilder` interprets the targets given on the command line.

### Resolution

Most targets that Jbuilder knows how to build lives in the `_build` directory, except for a few:

= `.merlin` files

- **`<package>.install` files; for the `default` context Jbuilder knows how** generate the install file both in `_build/default` and in the source tree so that `opam` can find it

As a result, if you want to ask `jbuilder` to produce a particular `.exe` file you would have to type:

```
$ jbuilder build _build/default/bin/prog.exe
```

However, for convenience when a target on the command line doesn't start with `_build`, `jbuilder` will expand it to the corresponding target in all the build contexts where it knows how to build it. It prints out the actual set of targets when starting so that you know what is happening:

```
$ jbuilder build bin/prog.exe
...
Actual targets:
- _build/default/bin/prog.exe
- _build/4.03.0/bin/prog.exe
- _build/4.04.0/bin/prog.exe
```

### Aliases

Targets starting with a `@` are interpreted as aliases. For instance `@src/runtest` means the alias `src/runtest`. If you want to refer to a target starting with a `@`, simply write: `./@foo`.

Note that an alias not pointing to the `_build` directory always depends on all the corresponding aliases in build contexts.

So for instance:

- `jbuilder build @_build/foo/runtest` will run the tests only for the `foo` build context
- `jbuilder build @runtest` will run the tests for all build contexts

## Finding external libraries

When a library is not available in the workspace, jbuilder will look it up in the installed world, and expect it to be already compiled.

It looks up external libraries using a specific list of search pathes. A list of search pathes is specific to a given build context and is determined as follow:

1. if the `ocamlfind` is present in the `PATH` of the context, use each line in the output of `ocamlfind printconf path` as a search path
2. otherwise, if `opam` is present in the `PATH`, use the outout of `opam config var lib`
3. otherwise, take the directory where `ocamlc` was found, and append `../lib` to it. For instance if `ocamlc` is found in `/usr/bin`, use `/usr/lib`

### Running tests

There are two ways to run tests:

- `jbuilder build @runtest`
- `jbuilder runtest`

The two commands are equivalent. They will run all the tests defined in the current directory and its children recursively. You can also run the tests in a specific sub-directory and its children by using:

- `jbuilder build @foo/bar/runtest`
- `jbuilder runtest foo/bar`

## Restricting the set of packages

You can restrict the set of packages from your workspace that Jbuilder can see with the `--only-packages` option:

```
$ jbuilder build --only-packages pkg1,pkg2,... @install
```

This option acts as if you went through all the jbuild files and commented out the stanzas refering to a package that is not in the list given to `jbuilder`.

## Invocation from opam

You should set the `build:` field of your `<package>.opam` file as follows:

```
build: [["jbuilder" "build" "-p" name "-j" jobs]]
```

`-p pkg` is a shorthand for `--root . --only-packages pkg`. `-p` is the short version of `--for-release-of-packages`.

This has the following effects:

- it tells jbuilder to build everything that is installable and to ignore packages other than `name` defined in your project
- it sets the root to prevent jbuilder from looking it up
- it uses whatever concurrency option opam provides

Note that `name` and `jobs` are variables expanded by opam. `name` expands to the package name and `jobs` to the number of jobs available to build the package.

## Tests

To setup the building and running of tests in opam, add this line to your `<package>.opam` file:

```
build-test: [["jbuilder" "runtest" "-p" name "-j" jobs]]
```

## Installation

Installing a package means copying the build artifacts from the build directory to the installed word.

When installing via opam, you don't need to worry about this step: jbuilder generates a `<package>.install` file that opam will automatically read to handle installation.

However, when not using opam or doing local development, you can use jbuilder to install the artifacts by hands. To do that, use the `install` command:

```
$ jbuilder install [PACKAGE]...
```

without an argument, it will install all the packages available in the workspace. With a specific list of packages, it will only install these packages. If several build contexts are configured, the installation will be performed for all of them.

Note that `jbuilder install` is a thin wrapper around the `opam-installer` tool, so you will need to install this tool in order to be able to use `jbuilder install`.

### Destination

The place where the build artifacts are copied, usually referred as **prefix**, is determined as follow for a given build context:

1. if an explicit `--prefix <path>` argument is passed, use this path
2. if `opam` is present in the `PATH` and is configured, use the output of `opam config var prefix`
3. otherwise, take the parent of the directory where `ocamlc` was found.

As an exception to this rule, library files might be copied to a different location. The reason for this is that they often need to be copied to a particular location for the various build system used in OCaml projects to find them and this location might be different from `<prefix>/lib` on some systems.

Historically, the location where to store OCaml library files was configured through findlib and the `ocamlfind` command line tool was used to both install these files and locate them. Many Linux distributions or other packaging systems are using this mechanism to setup where OCaml library files should be copied.

As a result, if none of `--libdir` and `--prefix` is passed to `jbuilder install` and `ocamlfind` is present in the `PATH`, then library files will be copied to the directory reported by `ocamlfind printconf destdir`. This ensures that `jbuilder install` can be used without opam. When using opam, `ocamlfind` is configured to point to the opam directory, so this rule makes no difference.

Note that `--prefix` and `--libdir` are only supported if a single build context is in use.

## Workspace configuration

By default, a workspace has only one build context named `default` which correspond to the environment in which `jbuilder` is run. You can define more contexts by writing a `jbuild-workspace` file.

You can point `jbuilder` to an explicit `jbuild-workspace` file with the `--workspace` option. For instance it is good practice to write a `jbuild-workspace.dev` in your project with all the version of OCaml your projects support. This way developers can tests that the code builds with all version of OCaml by simply running:

```
$ jbuilder build --workspace jbuild-workspace.dev @install @runtest
```

### jbuild-workspace

The `jbuild-workspace` file uses the S-expression syntax. This is what a typical `jbuild-workspace` file looks like:

```
(context ((switch 4.02.3)))
(context ((switch 4.03.0)))
(context ((switch 4.04.0)))
```

The rest of this section describe the stanzas available.

Note that an empty `jbuild-workspace` file is interpreted the same as one containing exactly:

```
(context default)
```

This allows you to use an empty `jbuild-workspace` file to mark the root of your project.

#### context

The `(context ...)` stanza declares a build context. The argument can be either `default` for the default build context or can be the description of an opam switch, as follows:

```
(context ((switch <opam-switch-name>)
          <optional-fields>))
```

`<optional-fields>` are:

- `(name <name>)` is the name of the subdirectory of `_build` where the artifacts for this build context will be stored

- `(root <opam-root>)` is the opam root. By default it will take the opam root defined by the environment in which `jbuilder` is run which is usually `~/.opam`
- **`(merlin)` instructs Jbuilder to use this build context for** merlin

Merlin reads compilation artifacts and it can only read the compilation artifacts of a single context. Usually, you should use the artifacts from the `default` context, and if you have the `(context default)` stanza in your `jbuild-workspace` file, that is the one Jbuilder will use.

For rare cases where this is not what you want, you can force Jbuilder to use a different build contexts for merlin by adding the field `(merlin)` to this context.

## Building JavaScript with js_of_ocaml

Jbuilder knows how to generate a JavaScript version of an executable (`<name>.bc.js`) using the js_of_ocaml compiler (the `js_of_ocaml-compiler` opam package must be installed).

It supports two modes of compilation:

- Direct compilation of a bytecode program to JavaScript. This mode allows js_of_ocaml to perform whole program deadcode elimination and whole program inlining.
- Separate compilation, where compilation units are compiled to JavaScript separately and then linked together. This mode is useful during development as it builds more quickly.

The separate compilation mode will be selected when passing `--dev` to jbuilder. There is currently no other way to control this behaviour.

See the section about *js_of_ocaml* for passing custom flags to the js_of_ocaml compiler

## Using topkg with jbuilder

Jbuilder provides suport for building and installing your project. However it doesn't provides helpers for distributing it. It is recommemded to use Topkg for this purpose.

The topkg-jbuilder project provides helpers for using Topkg in a Jbuilder project. In particular, as long as your project uses the common defaults, just write this `pkg/pkg.ml` file and you are all set:

```
#use "topfind"
#require "topkg-jbuilder.auto"
```

It is planned that this file won't be necessary at all soon and topkg will work out of the box on jbuilder projects.

The common defaults are that your projects include the following files:

- `README.md`
- `CHANGES.md`
- `LICENSE.md`

And that if your project contains several packages, then all the package names must be prefixed by the shortest one.

One of the feature topkg provides is watermarking; it replaces various strings of the form `%%ID%%` in all files of your project before creating a release tarball or when the package is pinned by the user using opam.

This is especially interesting for the `VERSION` watermark, which gets replaced by the version obtained from the vcs. For instance if you are using git, topkg invokes this command to find out the version:

```
$ git describe --always --dirty
1.0+beta9-79-g29e9b37
```

Projects using jbuilder usually only need topkg for creating and publishing releases. However they might still want to substitute the watermarks when the package is pinned by the user. To help with this, jbuilder provides the `subst` sub-command.

`jbuilder subst` performs the same substitution `topkg` does with the default configuration. i.e. calling `jbuilder subst` at the root of your project will rewrite in place all the files in your project.

More precisely, it replaces all the following watermarks in source files:

- `NAME`, the name of the project
- `VERSION`, output of `git describe --always --dirty`
- `VERSION_NUM`, same as `VERSION` but with a potential leading `v` or `V` dropped
- `VCS_COMMIT_ID`, commit hash from the vcs
- `PKG_MAINTAINER`, contents of the `maintainer` field from the opam file
- `PKG_AUTHORS`, contents of the `authors` field from the opam file
- `PKG_HOMEPAGE`, contents of the `homepage` field from the opam file
- `PKG_ISSUES`, contents of the `issues` field from the opam file
- `PKG_DOC`, contents of the `doc` field from the opam file
- `PKG_LICENSE`, contents of the `license` field from the opam file
- `PKG_REPO`, contents of the `repo` field from the opam file

Note that if your project contains several packages, `NAME` will be replaced by the shorted package name as long as it is a prefix of all the package names. If your package names don't follow this rule, you need to specify the name explicitly via the `-n` flag:

```
$ jbuilder subst -n myproject
```

Finally, note that jbuilder doesn't allow you to customize the list of substituted watermarks. If you which to do so, you need to configure topkg and use it instead of `jbuilder subst`.

CHAPTER 8

# Advanced topics

This section describes some details of Jbuilder for advanced users.

## META file generation

Jbuilder uses `META` files from the findlib library manager in order to interoperate with the rest of the world when installing libraries. It is able to generate them automatically. However, for the rare cases where you would need a specific `META` file, or to ease the transition of a project to Jbuilder, it is allowed to write/generate a specific one.

In order to do that, write or setup a rule to generate a `META.<package>` file in the same directory as the `<package>.opam` file. If you do that, Jbuilder will still generate a `META` file but it will be called `META.<package>.from-jbuilder`. So for instance if you want to extend the `META` file generated by Jbuilder you can write:

```
(rule
 ((targets (META.foo))
  (deps    (META.foo.from-jbuilder))
  (action  (with-stdout-to ${@}
            (progn
             (cat ${<})
             (echo blah))))))
```

Additionally, Jbuilder provides a simpler mechanism for this scheme: just write or generate a `META.<package>.template` file containing a line of the form `# JBUILDER_GEN`. Jbuilder will automatically insert its generated `META` contents in place of this line.

## Using a custom ppx driver

You can use a custom ppx driver by putting it as the last library in `(pps ...)` forms. An example of alternative driver is ppx_driver. To use it instead of `ocaml-migrate-parsetree.driver-main`, simply write `ppx_driver.runner` as the last library:

```
(preprocess (pps (ppx_sexp_conv ppx_bin_prot ppx_driver.runner)))
```

### Driver expectation

Jbuilder will invoke the executable resulting from linking the libraries given in the `(pps ...)` form as follows:

```
ppx.exe <flags-written-by-user> --dump-ast -o <output-file> \
  [--cookie library-name="<name>"] [--impl|--intf] <source-file>
```

Where `<source-file>` is either an implementation (`.ml`) or interface (`.mli`) OCaml source file. The command is expected to write a binary OCaml AST in `<output-file>`.

Additionally, it is expected that if the executable is invoked with `--as-ppx` as its first argument, then it will behave as a standard ppx rewirter as passed to `-ppx` option of OCaml. This is for two reasons:

- to improve interoperability with build systems other than Jbuilder
- so that it can be used with merlin

## Findlib integration and limitations

Jbuilder uses `META` files to support external libraries. However, it doesn't export the full power of findlib to the user, and especially it doesn't let the user specify *predicates*.

The reason for this limitation is that so far they haven't been needed, and adding full support for them would complicate things quite a lot. In particular, complex `META` files are often hand-written and the various features they offer are only available once the package is installed, which goes against the root ideas jbuilder is built on.

In practice, jbuilder interpret `META` files assuming the following set of predicates:

- `mt`: what this means is that using a library that can be used with or without threads with jbuilder will force the threaded version
- `mt_posix`: forces the use of posix threads rather than VM threads. VM threadws are deprecated and are likely to go away soon
- `ppx_driver`: when a library acts differently depending on whether it is linked as part of a driver or meant to add a `-ppx` argument to the compiler, choose the former behavior