
jbuilder Documentation

Release

Jérémie Dimino

May 15, 2017

Contents

1	Quickstart	1
1.1	Building a hello world program	1
1.2	Building a hello world program using Lwt	1
1.3	Defining a library using Lwt and ocaml-re	2
1.4	Using cppo	2
1.4.1	Using the .cppo.ml style like the ocamlbuild plugin	2
1.5	Defining a library with C stubs	2
1.6	Defining a library with C stubs using pkg-config	3
1.7	Using a custom code generator	4
1.8	Defining tests	4
2	Overview	5
3	Terminology	7
4	Project Layout and Metadata Specification	9
4.1	Metadata format	9
4.2	<package>.opam files	9
4.2.1	Package version	10
4.2.2	Odig conventions	10
4.3	jbuild-ignore	10
5	jbuild specification	11
5.1	Stanzas	11
5.1.1	jbuild_version	11
5.1.2	library	11
5.1.3	executable	13
5.1.4	executables	14
5.1.5	rule	14
5.1.6	ocamllex	15
5.1.7	ocamlyacc	15
5.1.8	menhir	15
5.1.9	alias	15
5.1.10	install	16
5.2	Common items	17
5.2.1	Ordered set language	17
5.2.2	Variables expansion	17

5.2.3	Library dependencies	18
5.2.4	Preprocessing specification	19
5.2.5	Dependency specification	20
5.2.6	OCaml flags	21
5.2.7	js_of_ocaml	21
5.2.8	User actions	22
5.3	OCaml syntax	23
6	Usage	25
6.1	Finding the root	25
6.1.1	jbuild-workspace	25
6.1.2	jbuild-workspace*	26
6.1.3	Current directory	26
6.1.4	Forcing the root (for scripts)	26
6.2	Interpretation of targets	26
6.2.1	Resolution	26
6.2.2	Aliases	27
6.3	Finding external libraries	27
6.3.1	Running tests	27
6.4	Restricting the set of packages	27
6.5	Invocation from opam	28
6.6	Tests	28
6.7	Installation	28
6.7.1	Destination	28
6.8	Workspace configuration	29
6.8.1	jbuild-workspace	29
6.9	Building JavaScript with js_of_ocaml	29
6.10	Using topkg with jbuilder	30
7	Advanced topics	31
7.1	META file generation	31
7.2	Using a custom ppx driver	31
7.2.1	Driver expectation	32

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)

(executables
  ((names (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)

(executables
  ((names (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`

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 `executables` stanzas:

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

Additionally, if you 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)

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

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
```


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 turns 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 it automatically. In particular this mean 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 mean that you can always drop in the source for a dependency of your project in your working copy and Jbuilder will start using 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 *this section*.
- **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
- **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

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. Jane Street packages use a special `jane_street` version which correspond to a rolling and unstable version that follows the internal Jane Street development. You shouldn't use this in your project, it is only intended to make the publication of Jane Street packages easier.

Except for the special `jane_street` version, 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.

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, documentations, ... to your package by declaring them in `jbuild` files.

Jbuilder will only register the existence of `<package>` in the subtree starting where the `<package>.opam` file lives, so you can only declare parts of the packages in this subtree. 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.

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 include 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. 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.

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.

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 specifying 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 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.

<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
- `(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 *libraries*
- `(preprocess <preprocess-spec>)` is the same as the `(preprocess ...)` field of *libraries*
- `(preprocessor_deps (<deps-conf list>))` is the same as the `(preprocessor_deps ...)` field of *libraries*
- `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>))
    (deps    (<deps-conf list>))
    (action  <action>)))
```

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

<deps-conf list> specifies the dependencies of the rule. See the *dependency specification section* for more details.

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

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.

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 *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 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.

`<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

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
- `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, separated by spaces
- `<` 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
- `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 whether 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

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

Library dependencies

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

For libraries that are present in the workspace, you can use either the real name (with some restrictions, see below) or the public name. For libraries that are part of the installed world, you need to use the public name. For instance: `(libraries (base re))`.

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

1. Scope of internal library names

The scope of internal library names is not the whole workspace. It is restricted to the subtree starting from the closest parent containing a `<package>.opam` file, or the whole workspace if no such directory exist. Moreover, a subtree containing `<package>.opam` doesn't inherit the internal names available in its parent scope.

The idea behind this rule is that public library names must be universally unique, but internal ones don't need to. In particular you might have private libraries that are only used for tests or building an executable.

As a result, when you create a workspace including several projects there might be a name clash between internal library names.

This scoping rule ensure that this won't be a problem.

2. 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.

1. 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> ${<>}")`.

2. 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. Additionally, 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_driver`.

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.

3. 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_file
  (<spec1> (<module-list1>))
  (<spec2> (<module-list2>))
  ...))
```

Where `<spec1>`, `<spec2>`, ... are preprocessing specifications and `<module-list1>`, `<module-list2>`, ... are list of module names. It is currently not possible to distinguish between `.ml/.mli` files, however it wouldn't be hard to support if needed.

For instance:

```
(preprocess (per_file
  ((command "./pp.sh X=1" (foo bar)))
  ((command "./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 section* 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*.

1. 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*.

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`
- `(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
- `(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`
- `(cat <file>)` to print the contents of a file to `stdout`
- `(copy <src> <dst>)` to copy a file
- **`(copy-and-add-line-directive <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

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 file-name 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 your 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 script can use the directive `#require` to access libraries:

```
#require "base,re";;
```

Note that any library required by a jbuild file must be part of the installed world.

If you don't like the S-expression syntax, then this method gives you a way to use whatever else you want. For instance you could have an API to describe your project in OCaml directly:

```
(* -- tuareg -- *)
#require "my_jbuild_api"
open My_jbuild_api

let () =
  library "foo" ~modules:["plop"; "bidule"]
```

Currently the `Jbuild_plugin` module is only available inside plugins. It is however planned to make it a proper library, see the roadmap for details.

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:

```
$ jbuilder runtest
Workspace root: /usr/local/home/jdimino/workspaces/public-jane/+share+
...
```

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 about 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 applies, 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.

Notet 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`
- `jbuidler 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`, use the output of `opam config var prefix`

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`

Note that `--prefix` is 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.

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 generate the `.merlin` files from this context. There can be at most one build context with a `(merlin)` field. If no build context has a `(merlin)` field, the selected context for merlin will be `(context default)` if present. Otherwise Jbuilder won't generate `.merlin` files

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 support for building and installing your project. However it doesn't provide helpers for distributing it. It is recommended to use `Topkg` for this purpose.

The `topkg-jbuilder` project provides helpers for using `Topkg` in a `Jbuilder` project.

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` rewriter as passed to `-ppx` option of OCaml. This is for two reason:

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