
jbuilder Documentation

Release

Jérémie Dimino

Apr 19, 2018

1	Quickstart	1
1.1	Building a hello world program	1
1.2	Building a hello world program using Lwt	1
1.3	Building a hello world program using Core and Jane Street PPXs	2
1.4	Defining a library using Lwt and ocaml-re	2
1.5	Using cppo	3
1.5.1	Using the .cppo.ml style like the ocamlbuild plugin	3
1.6	Defining a library with C stubs	3
1.7	Defining a library with C stubs using pkg-config	3
1.8	Using a custom code generator	4
1.9	Defining tests	5
1.10	Building a custom toplevel	5
2	Overview	7
3	Terminology	9
4	Project Layout and Metadata Specification	11
4.1	Metadata format	11
4.1.1	Lexical conventions of s-expressions	11
4.1.2	Comments	12
4.1.3	Grammar of s-expressions	12
4.1.4	Examples	12
4.2	<package>.opam files	12
4.2.1	Scopes	13
4.2.2	Package version	13
4.2.3	Odig conventions	13
4.3	jbuild-ignore	13
5	jbuild specification	15
5.1	Stanzas	15
5.1.1	jbuild_version	15
5.1.2	library	15
5.1.3	executable	18
5.1.4	executables	20
5.1.5	rule	20
5.1.6	ocamllex	21

5.1.7	ocamlyacc	22
5.1.8	menhir	22
5.1.9	alias	22
5.1.10	install	23
5.1.11	copy_files	24
5.1.12	include	24
5.2	Common items	25
5.2.1	Ordered set language	25
5.2.2	Variables expansion	25
5.2.3	Library dependencies	27
5.2.4	Preprocessing specification	28
5.2.5	Dependency specification	29
5.2.6	OCaml flags	30
5.2.7	js_of_ocaml	30
5.2.8	User actions	31
5.2.9	Locks	32
5.2.10	Diffing and promotion	33
5.3	OCaml syntax	34
6	Writing and running tests	35
6.1	Running tests	35
6.2	Inline tests	35
6.2.1	Inline expectation tests	36
6.2.2	Specifying inline test dependencies	37
6.2.3	Passing special arguments to the test runner	37
6.2.4	Using additional libraries in the test runner	38
6.2.5	Defining your own inline test backend	38
6.3	Custom tests	39
6.3.1	Diffing the result	39
7	Generating Documentation	41
7.1	Prerequisites	41
7.2	Writing Documentation	41
7.3	Building Documentation	41
7.4	Documentation Stanza	42
7.4.1	Examples	42
8	Usage	43
8.1	Finding the root	43
8.1.1	jbuild-workspace	43
8.1.2	jbuild-workspace*	44
8.1.3	Current directory	44
8.1.4	Forcing the root (for scripts)	44
8.2	Interpretation of targets	44
8.2.1	Resolution	44
8.2.2	Aliases	45
8.3	Finding external libraries	45
8.4	Running tests	45
8.5	Launching the Toplevel (REPL)	45
8.5.1	Requirements & Limitations	46
8.6	Restricting the set of packages	46
8.7	Invocation from opam	46
8.8	Tests	46
8.9	Installation	46

8.9.1	Destination	47
8.10	Workspace configuration	47
8.10.1	jbuild-workspace	47
8.11	Building JavaScript with js_of_ocaml	48
8.12	Using topkg with jbuilder	49
9	Advanced topics	51
9.1	META file generation	51
9.2	Using a custom ppx driver	51
9.2.1	Driver expectation	52
9.3	Findlib integration and limitations	52
9.4	Cross Compilation	52
9.4.1	How does it work?	53
9.5	Classical ppx	54
10	Configurator	55
10.1	Usage	55
10.2	Upgrading from the old Configurator	56
11	FAQ	59
11.1	Why do many Jbuilder projects contain a Makefile?	59
11.2	How to add a configure step to a jbuilder project?	59
11.3	Can I use topkg with jbuilder?	59
11.4	here can I find some examples of projects using Jbuilder?	59
12	Known Issues	61
12.1	mli only modules	61

This document gives simple usage examples of Jbuilder. You can also look at [examples](#) for complete examples of projects using Jbuilder.

1.1 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`. Note that native code executables will have the `.exe` extension on all platforms (including non-Windows systems).

1.2 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`

1.3 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`

1.4 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.

1.5 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))
```

1.5.1 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 ${@})))
```

1.6 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))))
```

1.7 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))
```

1.8 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 ${{@}})))
```

1.9 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
```

1.10 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 ()
```


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.

- **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. Aliases are per-directory. However, on the command line, asking for an alias to be built in a given directory will trigger the construction of the alias in all children directories recursively. Jbuilder defines the following standard aliases:
 - `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

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.

4.1 Metadata format

Most configuration files read by Jbuilder are using the S-expression syntax, which is very simple. It is described below.

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

4.1.1 Lexical conventions of s-expressions

Whitespace, which consists of space, newline, horizontal tab, and form feed, is ignored unless within an OCaml-string, where it is treated according to OCaml-conventions. The left parenthesis opens a new list, the right one closes it. Lists can be empty.

The double quote denotes the beginning and end of a string using similar lexing conventions to the ones of OCaml (see the OCaml-manual for details). Differences are:

- octal escape sequences (`\o123`) are not supported;
- backslash that's not a part of any escape sequence is kept as it is instead of resulting in parse error;

- a backslash followed by a space does not form an escape sequence, so it's interpreted as is, while it is interpreted as just a space by OCaml.

All characters other than double quotes, left- and right parentheses, whitespace, carriage return, and comment-introducing characters or sequences (see next paragraph) are considered part of a contiguous string.

4.1.2 Comments

There are three kinds of comments:

- line comments are introduced with `;`, and end at the newline;
- sexp comments are introduced with `#;`, and end at the end of the following s-expression;
- block comments are introduced with `#|` and end with `|#`. These can be nested, and double-quotes within them must be balanced and be lexically correct OCaml strings.

4.1.3 Grammar of s-expressions

S-expressions are either sequences of non-whitespace characters (= atoms), doubly quoted strings or lists. The lists can recursively contain further s-expressions or be empty, and must be balanced, i.e. parentheses must match.

4.1.4 Examples

```
this_is_an_atom_123'&^%! ; this is a comment
"another atom in an OCaml-string \"string in a string\" \123"

; empty list follows below
()

; a more complex example
(
  (
    list in a list ; comment within a list
    (list in a list in a list)
    42 is the answer to all questions
    #; (this S-expression
      (has been commented out)
    )
    #| Block comments #| can be "nested"|#|#
  )
)
)
```

4.2 <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.

4.2.1 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-directories.

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 wish 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.

4.2.2 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 used 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 one of the file listed above. See the section about *Using topkg with jbuilder* for more details.

4.2.3 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.

4.3 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 `jbuidl-ignore` file in the parent directory containing the name of the sub-directories to ignore.

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

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

`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.

5.1 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.

5.1.1 `jbuild_version`

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

5.1.2 `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 which should be built by default. The most common use for this feature is to disable native compilation when writing libraries for the OCaml toplevel. The following modes are available: `byte`, `native` and `best`. `best` is `native` or `byte` when native compilation is not available
- (`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 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
- (`modules_without_implementation <modules>`) specifies a list of modules that have only a `.mli` or `.rei` but no `.ml` or `.re` file. Such modules are usually referred as *mli only modules*. They are not officially supported by the OCaml compiler, however they are commonly used. Such modules must only define types. Since it is not reasonably possible for Jbuilder to check that this is the case, Jbuilder requires the user to explicitly list such modules to avoid surprises. `<modules>` must be a subset of the modules listed in the `(modules ...)` field.
- (`allow_overlapping_dependencies`) allows external dependencies to overlap with libraries that are present in the workspace

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 ...))`.

5.1.3 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.

Executables can also be linked as object or shared object files. See *linking modes* for more information.

<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>)) sets the *linking modes*. The default is (byte exe)
- (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*
- (modules_without_implementation <modules>) is the same as the corresponding field of *library*

- (`allow_overlapping_dependencies`) is the same as the corresponding field of *library*

Linking modes

The `modes` field allows to select what linking modes should be used to link executables. Each mode is a pair (`<compilation-mode>` `<binary-kind>`) where `<compilation-mode>` describes whether the byte code or native code backend of the OCaml compiler should be used and `<binary-kind>` describes what kind of file should be produced.

`<compilation-mode>` must be `byte`, `native` or `best`, where `best` is `native` with a fallback to `byte-code` when `native` compilation is not available.

`<binary-kind>` is one of:

- `exe` for normal executables
- `object` for producing static object files that can be manually linked into C applications
- `shared_object` for producing object files that can be dynamically loaded into an application. This mode can be used to write a plugin in OCaml for a non-OCaml application.

For instance the following `executables` stanza will produce byte code executables and native shared objects:

```
(executables
  (names (a b c))
  (modes ((byte exe) (native shared_object))))
```

Additionally, you can use the following short-hands:

- `exe` for `(best exe)`
- `object` for `(best object)`
- `shared_object` for `(best shared_object)`
- `byte` for `(byte exe)`
- `native` for `(native exe)`

For instance the following `modes` fields are all equivalent:

```
(modes (exe object shared_object))
(modes ((best exe)
        (best object)
        (best shared_object)))
```

The extensions for the various linking modes are choosen as follow:

compilation mode	binary kind	extensions
byte	exe	.bc and .bc.js
native/best	exe	.exe
byte	object	.bc\${ext_obj}
native/best	object	.exe\${ext_obj}
byte	shared_object	.bc\${ext_dll}
native/best	shared_object	\${ext_dll}

Where `${ext_obj}` and `${ext_dll}` are the extensions for object and shared object files. Their value depends on the OS, for instance on Unix `${ext_obj}` is usually `.o` and `${ext_dll}` is usually `.so` while on Windows `${ext_obj}` is `.obj` and `${ext_dll}` is `.dll`.

Note that when `(byte exe)` is specified but neither `(best exe)` nor `(native exe)` are specified, Jbuildk still knows how to build an executable with the extension `.exe`. In such case, the `.exe` version is the same as the `.bc` one except that it is linked with the `-custom` option of the compiler. You should always use the `.exe` rather than the `.bc` inside build rules.

5.1.4 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

5.1.5 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.
- `(mode <mode>)` to specify how to handle the targets, see *modes* for details
- `(fallback)` is deprecated and is the same as `(mode fallback)`
- `(locks (<lock-names>))` specify that the action must be run while holding the following locks. See the *Locks* 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.

modes

By default, the target of a rule must not exist in the source tree and Jbuilder will error out when this is the case.

However, it is possible to change this behavior using the `mode` field. The following modes are available:

- `standard`, this is the standard mode
- `fallback`, in this mode, when 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.
- `promote`, in this mode, the files in the source tree will be ignored. Once the rule has been executed, the targets will be copied back to the source tree
- `promote-until-clean` is the same as `promote` except than `jbuilder clean` will remove the promoted files from the source tree

There are two use cases for promote rules. The first one is when the generated code is easier to review than the generator, so it's easier to commit the generated code and review it. The second is to cut down dependencies during releases: by passing `--ignore-promoted-rules` to jbuilder, rules will (mode `promote`) will be ignored and the source files will be used instead. The `-p/--for-release-of-packages` flag implies `--ignore-promote-rules`.

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}))
```

5.1.6 ocamllex

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

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

To use a different rule mode, use the long form:

```
(ocamllex
  (modules (<names>))
  (mode <mode>))
```

5.1.7 ocaml yacc

(ocaml yacc (<names>)) is essentially a shorthand for:

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

To use a different rule mode, use the long form:

```
(ocaml yacc
  ((modules (<names>))
   (mode    <mode>)))
```

5.1.8 menhir

The basic form for defining menhir parsers (analogous to ocaml yacc) 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> ...))))
```

5.1.9 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>`
- `(locks (<lock-names>))` specify that the action must be run while holding the following locks. See the [Locks](#) section for more details.

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.

5.1.10 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.

5.1.11 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.

5.1.12 include

The `include` stanza allows to include the contents of another file into the current jbuild file. Currently, the included file cannot be generated and must be present in the source tree. This feature is intended to be used in conjunction with promotion, when parts of a jbuild file are to be generated.

For instance:

```
(include jbuild.inc)

(rule (with-stdout-to jbuild.inc.gen (run ./gen-jbuild.exe)))

(alias
  ((name  runtest)
   (action (diff jbuild.inc jbuild.inc.gen))))
```

With this jbuild file, running jbuilder as follow will replace the `jbuild.inc` file in the source tree by the generated one:

```
$ jbuilder build @runtest --auto-promote
```

5.2 Common items

5.2.1 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.

5.2.2 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 workspace 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 independent of the workspace configuration
- **CC is the C compiler command line (list made of the compiler name followed by its flags)** that was used to compile OCaml 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
- **ext_obj, ext_asm, ext_lib, ext_dll and ext_exe** are the file extension used for various artifacts

- `ocaml-config:v` for every variable `v` in the output of `ocamlc -config`. Note that output Jbuilder processes the output of `ocamlc -config` in order to make it a bit more stable across versions, so the exact set of variables accessible this way might not be exactly the same as what you can see in the output of `ocamlc -config`. In particular, variables added in new versions of OCaml needs to be registered in Jbuilder before they can be used

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>`
- `path-no-dep:<path>` is the same as `path:<path>`, except that `<path>` is not considered as a dependency of the action. For instance `(chdir ${ROOT} (run foo --base ${path-no-dep:bar}))` in `src/blah/jbuild` will expand to `(chdir ../../ (run foo --base src/blah/bar))` where `src/blah/bar` doesn't have to be an existing or buildable file
- `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 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
- `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 `${CC}`, `${^}`, `${@}` or `${read-lines: ...}`, are suitable to be used in, say, `(run <prog> <arguments>)`. For instance in:

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

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

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

If you want the two dependencies to be passed as a single argument, you have to quote the variable as in:

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

which is equivalent to the following shell command:

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

(the items of the list are concatenated with space). Note that, since `${^}` is a list of items, the first one may be used as a program name, for instance:

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

Here is another example:

```
(rule
  ((targets (foo.exe))
   (deps   (foo.c))
   (action (run ${CC} -o ${@} ${<} -lfoolib)))
```

5.2.3 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 preferred 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.

5.2.4 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. 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_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)))))
```

5.2.5 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)`
- `(alias_rec <alias-name>)`: depend on the construction of this alias recursively in all children directories wherever it is defined. For instance: `(alias_rec src/runtest)` might depend on `(alias src/runtest)`, `(alias src/foo/bar/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>`
- `(universe)`: depend on everything in the universe. This is for cases where dependencies are too hard to specify. Note that Jbuilder will not be able to cache the result of actions that depend on the universe. In any case, this is only for dependencies in the installed world, you must still specify all dependencies that come from the workspace.
- `(package <pkg>)` depend on all files installed by `<package>`, as well as on the transitive package dependencies of `<package>`. This can be used to test a command against the files that will be installed

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>`, ...

5.2.6 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>))
```

5.2.7 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.

5.2.8 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
- `(diff <file1> <file2>)` is similar to `(run diff <file1> <file2>)` but is better and allows promotion. See *Diffing and promotion* for more details
- `(diff? <file1> <file2>)` is the same as `(diff <file1> <file2>)` except that it is ignored when `<file1>` or `<file2>` doesn't exist

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 ${@} ${<}))))
```

5.2.9 Locks

Given two rules that are independent, Jbuilder will assume that their associated action can be run concurrently. Two rules are considered independent if none of them depend on the other, either directly or through a chain of dependencies. This basic assumption allows to parallelize the build.

However, it is sometimes the case that two independent rules cannot be executed concurrently. For instance this can happen for more complicated tests. In order to prevent jbuilder from running the actions at the same time, you can specify that both actions take the same lock:

```
(alias
  ((name  runtest)
   (deps  (foo))
   (locks (m))
   (action (run test.exe ${<})))

(alias
```

```
((name  runtest)
 (deps  (bar))
 (locks (m))
 (action (run test.exe ${<})))
```

Jbuilder will make sure that the executions of `test.exe foo` and `test.exe bar` are serialized.

Although they don't live in the filesystem, lock names are interpreted as file names. So for instance `(with-lock m ...)` in `src/jbuild` and `(with-lock ../src/m)` in `test/jbuild` refer to the same lock.

Note also that locks are per build context. So if your workspace has two build contexts setup, the same rule might still be executed concurrently between the two build contexts. If you want a lock that is global to all build contexts, simply use an absolute filename:

```
(alias
 ((name  runtest)
 (deps  (foo))
 (locks (/tcp-port/1042))
 (action (run test.exe ${<})))
```

5.2.10 Diffing and promotion

`(diff <file1> <file2>)` is very similar to `(run diff <file1> <file2>)`. In particular it behaves in the same way:

- when `<file1>` and `<file2>` are equal, it doesn't do anything
- when they are not, the differences are shown and the action fails

However, it is different for the following reason:

- the exact command used to diff files can be configured via the `--diff-command` command line argument. Note that it is only called when the files are not byte equals
- by default, it will use `patdiff` if it is installed. `patdiff` is a better diffing program. You can install it via `opam` with:

```
$ opam install patdiff
```

- since `(diff a b)` is a builtin action, Jbuilder knows that `a` and `b` are needed and so you don't need to specify them explicitly as dependencies
- you can use `(diff? a b)` after a command that might or might not produce `b`. For cases where commands optionally produce a *corrected* file
- it allows promotion. See below

Promotion

Whenever an action `(diff <file1> <file2>)` or `(diff? <file1> <file2>)` fails because the two files are different, jbuilder allows you to promote `<file2>` as `<file1>` if `<file1>` is a source file and `<file2>` is a generated file.

More precisely, let's consider the following jbuild file:

```
(rule
 (with-stdout-to data.out (run ./test.exe)))
```

```
(alias
  ((name  runtest)
   (action (diff data.expected data.out))))
```

Where `data.expected` is a file committed in the source repository. You can use the following workflow to update your test:

- update the code of your test
- run `jbuilder runtest`. The `diff` action will fail and a diff will be printed
- check the diff to make sure it is what you expect
- run `jbuilder promote`. This will copy the generated `data.out` file to `data.expected` directly in the source tree

You can also use `jbuilder runtest --auto-promote` which will automatically do the promotion.

5.3 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.

Writing and running tests

Jbuilder tries to streamline the testing story as much as possible, so that you can focus on the tests themselves and not bother with setting up with various test frameworks.

In this section, we will explain the workflow to deal with tests in Jbuilder. In particular we will see how to run the testsuite of a project, how to describe your tests to Jbuilder and how to promote tests result as expectation.

We distinguish two kinds of tests: inline tests and custom tests. Inline tests are usually written directly inside the ml files of a library. They are the easiest to work with and usually requires nothing more than writing `(inline_tests)` inside your library stanza. Custom tests consist on executing an executable and sometimes do something afterwards, such as diffing its output.

6.1 Running tests

Whatever the tests of a project are, the usual way to run tests with Jbuilder is to call `jbuilder runtest` from the shell. This will run all the tests defined in the current directory and any sub-directory recursively. You can also pass a directory argument to run the tests from a sub-tree. For instance `jbuilder runtest test` will only run the tests from the `test` directory and any sub-directory of `test` recursively.

Note that in any case, `jbuilder runtest` is simply a short-hand for building the `runtest` alias, so you can always ask Jbuilder to run the tests in conjunction with other targets by passing `@runtest` to `jbuilder build`. For instance:

```
$ jbuilder build @install @runtest
$ jbuilder build @install @test/runtest
```

6.2 Inline tests

There are several inline tests framework available for OCaml, such as `ppx_inline_test` and `qtest`. We will use `ppx_inline_test` as an example as at the time of writing this document it has the necessary setup to be used with Jbuilder out of the box.

`ppx_inline_test` allows to write tests directly inside ml files as follow:

```
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%test _ = fact 5 = 120
```

The file has to be preprocessed with the `ppx_inline_test` ppx rewriter, so for instance the `jbuild` file might look like this:

```
(library
  (name foo)
  (preprocess (pps (ppx_inline_test))))
```

In order to instruct Jbuilder that our library contains inline tests, all we have to do is add an `inline_tests` field:

```
(library
  (name foo)
  (inline_tests)
  (preprocess (pps (ppx_inline_test))))
```

We can now build and execute this test by running `jbuilder runtest`. For instance, if we make the test fail by replacing 120 by 0 we get:

```
$ jbuilder runtest
[...]
File "src/fact.ml", line 3, characters 0-25: <<(fact 5) = 0>> is false.

FAILED 1 / 1 tests
```

Note that in this case Jbuild knew how to build and run the tests without any special configuration. This is because `ppx_inline_test` defines an inline tests backend and it is used by the library. Some other frameworks, such as `qtest` don't have any special library or ppx rewriter. To use such a framework, you must tell Jbuilder about it since it cannot guess it. You can do that by adding a `backend` field:

```
(library
  (name foo)
  (inline_tests ((backend qtest))))
```

6.2.1 Inline expectation tests

Inline expectation tests are a special case of inline tests where you write a bit of OCaml code that prints something followed by what you expect this code to print. For instance, using `ppx_expect`:

```
let%expect_test _ =
  print_endline "Hello, world!";
  [%expect{|
    Hello, world!
  |}]
```

The test procedure consist of executing the OCaml code and replacing the contents of the `[%expect]` extension point by the real output. You then get a new file that you can compare to the original source file. Expectation tests are a neat way to write tests as the following test elements are clearly identified:

- the code of the test
- the test expectation

- the test outcome

You can have a look at [this blob post](#) to find out more about expectation tests. But Back to Jbuilder, the workflow for expectation tests is always as follow:

- write the test with some empty expect nodes in it
- run the tests
- check the suggested correction and promote it as the original source file if you are happy with it

Jbuilder makes this workflow very easy, simply add `ppx_expect` to your list of ppx rewriters as follow:

```
(library
  ((name foo)
   (inline_tests)
   (preprocess (pps (ppx_expect)))))
```

Then calling `jbuilder runtest` will run these tests and in case of mismatch Jbuilder will print a diff of the original source file and the suggested correction. For instance:

```
$ jbuilder runtest
[...]
-src/fact.ml
+src/fact.ml.corrected
File "src/fact.ml", line 5, characters 0-1:
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%expect_test _ =
  print_int (fact 5);
- [%expect]
+ [%expect{| 120 |}]
```

In order to accept the correction, simply run:

```
$ jbuilder promote
```

You can also make Jbuilder automatically accept the correction after running the tests by typing:

```
$ jbuilder runtest --auto-promote
```

Finally, some editor integration is possible to make the editor do the promotion and make the workflow even smoother.

6.2.2 Specifying inline test dependencies

If your tests are reading files, you must say it to Jbuilder by adding a `deps` field to the `inline_tests` field. The argument of this `deps` field follows the usual *Dependency specification*. For instance:

```
(library
  ((name foo)
   (inline_tests ((deps (data.txt))))
   (preprocess (pps (ppx_expect)))))
```

6.2.3 Passing special arguments to the test runner

Under the hood, a test executable is built by Jbuilder. Depending on the backend used this runner might take useful command line arguments. You can specify such flags by using a `flags` field, such as:

```
(library
  (name foo)
  (inline_tests ((flags (-foo bar))))
  (preprocess (pps (ppx_expect))))
```

The argument of the `flags` field follows the *Ordered set language*.

6.2.4 Using additional libraries in the test runner

When tests are not part of the library code, it is possible that tests require additional libraries than the library being tested. This is the case with `qtest` as tests are written in comments. You can specify such libraries using a `libraries` field, such as:

```
(library
  (name foo)
  (inline_tests ((backend qtest)
                (libraries (bar)))))
```

6.2.5 Defining your own inline test backend

If you are writing a test framework, or for specific cases, you might want to define your own inline tests backend. If your framework is naturally implemented by a library or ppx rewriter that the user must use when they want to write tests, then you should define this library has a backend. Otherwise simply create an empty library with the name you want to give for your backend.

In order to define a library as an inline tests backend, simply add an `inline_tests.backend` field to the library stanza. An inline tests backend is specified by three parameters:

1. How to create the test runner
2. How to build the test runner
3. How to run the test runner

These three parameters can be specified inside the `inline_tests.backend` field, which accepts the following fields:

```
(generate_runner <action>)
(runner_libraries (<ocaml-libraries>))
(flags <flags>)
(extends (<backends>))
```

For instance:

`<action>` follows the *User actions* specification. It describe an action that should be executed in the directory of libraries using this backend for their tests. It is expected that the action produces some OCaml code on its standard output. This code will constitute the test runner. The action can use the following additional variables:

- `#{library-name}` which is the name of the library being tested
- `#{impl-files}` which is the list of implementation files in the library, i.e. all the `.ml` and `.re` files
- `#{intf-files}` which is the list of interface files in the library, i.e. all the `.mli` and `.rei` files

The `runner_libraries` field specifies what OCaml libraries the test runner uses. For instance, if the `generate_runner` actions generates something like `My_test_framework.runtests ()`, the you should probably put `my_test_framework` in the `runner_libraries` field.

If your test runner needs specific flags, you should pass them in the `flags` field. You can use the `${library-name}` variable in this field.

Finally, a backend can be an extension of another backend. In this case you must specify by in the `extends` field. For instance, `ppx_expect` is an extension of `ppx_inline_test`. It is possible to use a backend with several extensions in a library, however there must be exactly one *root backend*, i.e. exactly one backend that is not an extension of another one.

When using a backend with extensions, the various fields are simply concatenated. The order in which they are concatenated is unspecified, however if a backend `b` extends of a backend `a`, then `a` will always come before `b`.

Example of backend

In this example, we put tests in comments of the form:

```
(*TEST: assert (fact 5 = 120) *)
```

The backend for such a framework looks like this:

```
(library
  ((name simple_tests)
    (inline_tests.backend
      ((generate_runner (run sed "s/(\\*TEST:\\\\(.*)\\*)/let () = \\1;;/" ${impl-files}
→)))
    )))
```

Now all you have to do is write `(inline_tests ((backend simple_tests)))` wherever you want to write such tests. Note that this is only an example, we do not recommend using `sed` in your build as this would cause portability problems.

6.3 Custom tests

We said in *Running tests* that to run tests Jbuilder simply builds the `runtest` alias. As a result, to define custom tests, you simply need to add an action to this alias in any directory. For instance if you have a binary `tests.exe` that you want to run as part of running your testsuite, simply add this to a `jbuild` file:

```
(alias
  ((name runtest)
    (action (run ./tests.exe))))
```

6.3.1 Diffing the result

It is often the case that we want to compare the output of a test to some expected one. For that, Jbuilder offers the `diff` command, which in essence is the same as running the `diff` tool, except that it is more integrated in Jbuilder and especially with the `promote` command. For instance let's consider this test:

```
(rule
  (with-stdout-to tests.output (run ./tests.exe))

(alias
  ((name runtest)
    (action (diff tests.expected test.output))))
```

After having run `tests.exe` and dumping its output to `tests.output`, Jbuilder will compare the latter to `tests.expected`. In case of mismatch, Jbuilder will print a diff and then the `jbuilder promote` command can be used to copy over the generated `test.output` file to `tests.expected` in the source tree. This provides a nice way of dealing with the usual *write code, run, promote* cycle of testing. For instance:

```
$ jbuilder runtest
[...]
-tests.expected
+tests.output
File "tests.expected", line 1, characters 0-1:
-Hello, world!
+Good bye!
$ jbuilder promote
Promoting _build/default/tests.output to tests.expected.
```

Note that if available, the diffing is done using the `patdiff` tool, which displays nicer looking diffs than the standard `diff` tool. You can change that by passing `--diff-command CMD` to Jbuilder.

Generating Documentation

7.1 Prerequisites

Documentation in jbuilder is done courtesy of the `odoc` tool. Therefore, to generate documentation in jbuilder, you will need to install this tool. This should likely be done with `opam`:

```
$ opam install odoc
```

7.2 Writing Documentation

Documentation comments will be automatically extracted from your OCaml source files following the syntax described in the section `Text formatting of the OCaml manual`.

Additional documentation pages may be attached to a package can be attached using the *Documentation Stanza*.

7.3 Building Documentation

Building the documentation using the `@doc` alias. Hence, all that is required to generate documentation for your project is building this alias:

```
$ jbuilder build @doc
```

An index page containing links to all the `opam` packages in your project can be found in:

```
$ open _build/default/_doc/_html/index.html
```

Documentation for private libraries may also be built with:

```
$ jbuilder build @doc-private
```

But this libraries will not be in the main html listing above, since they do not belong to any particular package. But the generated html will still be found in `_build/default/_doc/_html/<library>`.

7.4 Documentation Stanza

Documentation pages will be automatically generated for from `.ml` and `.mli` files that include `ocaml doc` fragments. Additional manual pages may be attached to packages using the `documentation` stanza. These `.mld` files must contain text in the same syntax as `ocaml doc` comments.

```
(documentation (<optional-fields>))
```

Where `<optional-fields>` are:

- `(package <name>)` the package this documentation should be attached to. If this absent, `jbuilder` will try to infer it based on the location of the stanza.
- `(mld_files <arg>)` where `<arg>` field follows the *Ordered set language*. This is a set of extension-less, `mld` file base names that are attached to the package. Where `:standard` refers to all the `.mld` files in the stanza's directory.

The `index.mld` file (specified as `index` in `mld_files`) is treated specially by `jbulder`. This will be the file used to generate the entry page for the package. This is the page that will be linked from the main package listing. If you omit writing an `index.mld`, `jbuilder` will generate one with the entry modules for your package. But this generated will not be installed.

All `mld` files attached to a package will be included in the generated `.install` file for that package, and hence will be installed by `opam`.

7.4.1 Examples

This stanza use attach all the `.mld` files in the current directory in a project with a single package.

```
(documentation ())
```

This stanza will attach three `mld` files to package `foo`. The `mld` files should be named `foo.mld`, `bar.mld`, and `baz.mld`

```
(documentation
  ((package foo)
   (mld_files (foo bar baz))))
```

This stanza will attach all `mld` files excluding `wip.mld` in the current directory to the inferred package:

```
(documentation
  ((mld_files (:standard \ wip))))
```

This section describe usage of Jbuilder from the shell.

8.1 Finding the root

8.1.1 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 runttest
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.

8.1.2 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
```

8.1.3 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.

8.1.4 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.

8.2 Interpretation of targets

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

8.2.1 Resolution

All targets that `Jbuilder` knows how to build live in the `_build` directory. Although, some are sometimes copied to the source tree for the need of external tools. These includes:

- `.merlin` files
- `<package>.install` files

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. When using `--verbose`, It prints out the actual set of targets when starting:

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

8.2.2 Aliases

Targets starting with a `@` are interpreted as aliases. For instance `@src/runtest` means the alias `runtest` in all descendant of `src` in all build contexts where it is defined. If you want to refer to a target starting with a `@`, simply write: `./@foo`.

To build and run the tests for a particular build context, use `@_build/default/runtest` instead.

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

8.3 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`

8.4 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`

8.5 Launching the Toplevel (REPL)

jbuilder supports launching a `utop` instance with locally defined libraries loaded.

```
$ jbuilder utop <dir> -- <args>
```

Where `<dir>` is a directory containing a `jbuild` file defining all the libraries that will be loaded (using the `library` stanza). `<args>` will be passed as arguments to the `utop` command itself. For example, to launch it in `emacs` mode.

8.5.1 Requirements & Limitations

- utop version ≥ 2.0 is required for this to work.
- This subcommand only supports loading libraries. Executables aren't supported.
- Libraries that are dependencies of utop itself cannot be loaded. For example [Camomile](#).
- Loading libraries that are defined in different directories into one utop instance isn't possible.

8.6 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 referring to a package that is not in the list given to jbuilder.

8.7 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 --promote ignore`. `-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 ignores promotion to cut down dependencies and speed up the build
- 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.

8.8 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]]
```

8.9 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`.

8.9.1 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.

8.10 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
```

8.10.1 jbuild-workspace

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

```
(context (opam (switch 4.02.3)))
(context (opam (switch 4.03.0)))
(context (opam (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` or `(default)` for the default build context or can be the description of an opam switch, as follows:

```
(context (opam (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

Both `(default ...)` and `(opam ...)` accept a `targets` field in order to setup cross compilation. See ‘**Cross Compilation**’ for more information.

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.

Note that the following syntax is still accepted but is deprecated:

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

it is interpreted the same as `(context (opam (switch ...) ...))`.

8.11 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

8.12 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. 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 features `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 wish to do so, you need to configure `topkg` and use it instead of `jbuilder subst`.

This section describes some details of Jbuilder for advanced users.

9.1 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>.template` file in the same directory as the `<package>.opam` file. Jbuilder will generate a `META.<package>` file from the `META.<package>.template` file by replacing lines of the form `# JBUILDER_GEN` by the contents of the `META` it would normally generate.

For instance if you want to extend the `META` file generated by Jbuilder you can write the following `META.foo.template` file:

```
# JBUILDER_GEN
blah = "..."
```

9.2 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)))
```

9.2.1 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 reasons:

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

9.3 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 interprets `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 threads 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

9.4 Cross Compilation

Jbuilder allows for cross compilation by defining build contexts with multiple targets. Targets are specified by adding a `targets` field to the definition of a build context.

`targets` takes a list of target name. It can be either:

- `native` which means using the native tools that can build binaries that run on the machine doing the build
- the name of an alternative toolchain

Note that at the moment, there is no official support for cross-compilation in OCaml. Jbuilder supports the `opam-cross-x` repositories from the [ocaml-cross organization on github](#), such as:

- `opam-cross-windows`
- `opam-cross-android`
- `opam-cross-ios`

In particular:

- to build Windows binaries using `opam-cross-windows`, write `windows` in the list of targets
- to build Android binaries using `opam-cross-android`, write `android` in the list of targets
- to build Android binaries using `opam-cross-ios`, write `ios` in the list of targets

For example, the following workspace file defines three different targets for the `default` build context:

```
(context (default (targets (native windows android))))
```

This configuration defines three build contexts:

- `default`
- `default.windows`
- `default.android`

Note that the `native` target is always implicitly added when not present. However, when implicitly added `jbuilder build @install` will skip this context, i.e. `default` will only be used for building executables needed by the other contexts.

With such a setup, calling `jbuilder build @install` will build all the packages three times.

Note that instead of writing a `jbuild-workspace` file, you can also use the `-x` command line option. Passing `-x foo` to `jbuilder` without having a `jbuild-workspace` file is the same as writing the following `jbuild-workspace` file:

```
(context (default (targets (foo))))
```

If you have a `jbuild-workspace` and pass a `-x foo` option, `foo` will be added as target of all context stanzas.

9.4.1 How does it work?

In such a setup, binaries that need to be built and executed in the `default.windows` or `default.android` contexts as part of the build, will no longer be executed. Instead, all the binaries that will be executed will come from the `default` context. One consequence of this is that all preprocessing (ppx or otherwise) will be done using binaries built in the `default` context.

To clarify this with an example, let's assume that you have the following `src/jbuild` file:

```
(executable ((name foo)))
(rule (with-stdout-to blah (run ./foo.exe)))
```

When building `_build/default/src/blah`, `jbuilder` will resolve `./foo.exe` to `_build/default/src/foo.exe` as expected. However, for `_build/default.windows/src/blah` `jbuilder` will resolve `./foo.exe` to `_build/default/src/foo.exe`

Assuming that the right packages are installed or that your workspace has no external dependencies, `jbuilder` will be able to cross-compile a given package without doing anything special.

Some packages might still have to be updated to support cross-compilation. For instance if the `foo.exe` program in the previous example was using `Sys.os_type`, it should instead take it as a command line argument:

```
(rule (with-stdout-to blah (run ./foo.exe -os-type ${os_type})))
```

9.5 Classical ppx

classical ppx refers to running ppx using the `-ppx` compiler option, which is composed using Findlib. Even though this is useful to run some (usually old) ppx's which don't support drivers, Jbuilder does not support preprocessing with ppx this way. but a workaround exists using the `ppxfind` tool.

CHAPTER 10

Configurator

Configurator is a small library designed to query features available on the system, in order to generate configuration for dune builds. Such generated configuration is usually in the form of command line flags, generated headers, stubs, but there are no limitations on this.

Configurator allows you to query for the following features:

- Variables defined in `ocamlc - config`,
- `pkg-config` flags for packages,
- Test features by compiling C code,
- Extract compile time information such as `#define` variables.

Configurator is designed to be cross compilation friendly and avoids `_running_` any compiled code to extract any of the information above.

Configurator started as an [independent library](#), but now lives in dune. You do not need to install anything to use configurator.

10.1 Usage

We'll describe configurator with a simple example. Everything else can be easily learned by studying [configurator's API](#).

To use configurator, we write an executable that will query the system using configurator's API and output a set of targets reflecting the results. For example:

```
module C = Configurator.V1

let clock_gettime_code = {}
#include <time.h>

int main()
{
```

```

struct timespec ts;
clock_gettime(CLOCK_REALTIME, &ts);
return 0;
}
|}

let () =
  C.main ~name:"foo" (fun c ->
    let has_clock_gettime = C.c_test c clock_gettime_code ~link_flags:["-lrt"] in

    C.C_define.gen_header_file c ~fname:"config.h"
    [ "HAS_CKOCK_GETTIME", Switch has_ckock_gettime ]);

```

Usually, the module above would be named `discover.ml`. The next step is to invoke it as an executable and tell dune about the targets that it produces:

```

(executable
  ((name discover)
   (libraries (jbuilder.configurator))))

(rule
  ((targets (config.h))
   (action (run ./discover.exe))))

```

Another common pattern is to produce a flags file with configurator and then use this flag file using `:include`:

```

(library
  ((name mylib)
   (c_names (foo))
   (c_library_flags (:include (flags.sexp)))))

```

For this, generate the list of flags for your library — for example using `Configurator.V1.Pkg_config` — and then write them to a file, in the above example `flags.sexp`, with `Configurator.V1.write_flags "flags.sexp" flags`.

10.2 Upgrading from the old Configurator

The old configurator is the independent `configurator` opam package. It is deprecated and users are encouraged to migrate to dune's own configurator. The advantage of the transition include:

- No extra dependencies,
- No need to manually pass `-ocamlc` flag,
- New configurator is cross compilation compatible.

The following steps must be taken to transition from the old configurator:

- Mentions of the `configurator` opam package should be removed.
- The library name `configurator` should be changed `jbuilder.configurator`.
- The `-ocamlc` flag in rules that run configurator scripts should be removed. This information is now passed automatically by dune.
- The new configurator API is versioned explicitly. The version that is compatible with old configurator is under the `V1` module. Hence, to transition one's code it's enough to add this module alias:

```
module Configurator = Configurator.V1
```


11.1 Why do many Jbuilder projects contain a Makefile?

Many Jbuilder project contain a toplevel *Makefile*. It is often only there for convenience, for the following reasons:

1. there are many different build systems out there, all with a different CLI. If you have been hacking for a long time, the one true invocation you know is *make && make install*, possibly preceded by *./configure*
2. you often have a few common operations that are not part of the build and *make <blah>* is a good way to provide them
3. *make* is shorter to type than *jbuilder build @install*

11.2 How to add a configure step to a jbuilder project?

example/sample-projects/with-configure-step shows one way to do it which preserves composability; i.e. it doesn't require manually running *./configure* script when working on multiple projects at the same time.

11.3 Can I use topkg with jbuilder?

Yes, have a look at the [topkg-jbuilder][topkg-jbuilder] project for more details.

11.4 here can I find some examples of projects using Jbuilder?

The [dune-universe](https://github.com/dune-universe/dune-universe) repository contains a snapshot of the latest versions of all opam packages depending on jbuilder. It is therefore a useful reference to search through to find different approaches to constructing build rules.

12.1 mli only modules

These are supported, however using them might cause make it impossible for non-jbuilder users to use your library. We tried to use them for some internal module generated by Jbuilder and it broke the build of projects not using Jbuilder:

<https://github.com/ocaml/dune/issues/567>

So, while they are supported, you should be careful where you use them. Using a *.ml* only module is still preferable.