PNPM Docs - English

Table of contents

Aliases

Aliases let you install packages with custom names.

Let's assume you use `lodash` all over your project. There is a bug in `lodash` that breaks your project. You have a fix but `lodash` won't merge it. Normally you would either install `lodash` from your fork directly (as a git-hosted dependency) or publish it with a different name. If you use the second solution you have to replace all the requires in your project with the new dependency name (`require('lodash')` => `require('awesome-lodash')`). With aliases, you have a third option.

Publish a new package called `awesome-lodash` and install it using `lodash` as its alias:

```
pnpm add lodash@npm:awesome-lodash
```

No changes in code are needed. All the requires of `lodash` will now resolve to `awesome-lodash`.

Sometimes you'll want to use two different versions of a package in your project. Easy:

```
pnpm add lodash1@npm:lodash@1
pnpm add lodash2@npm:lodash@2
```

Now you can require the first version of lodash via `require('lodash1')` and the second via `require('lodash2')`.

This gets even more powerful when combined with hooks. Maybe you want to replace `lodash` with `awesome-lodash` in all the packages in `node_modules`. You can easily achieve that with the following `.pnpmfile.cjs`:

```
function readPackage(pkg) {
  if (pkg.dependencies && pkg.dependencies.lodash) {
    pkg.dependencies.lodash = 'npm:awesome-lodash@^1.0.0'
  }
  return pkg
}

module.exports = {
  hooks: {
    readPackage
  }
}
```

Go to TOC

pnpm add

Installs a package and any packages that it depends on. By default, any new package is installed as a production dependency.

TL;DR

Command	Meaning
`pnpm add sax`	Save to `dependencies`
`pnpm add -D sax`	Save to `devDependencies`
`pnpm add -O sax`	Save to `optionalDependencies`
`pnpm add -g sax`	Install package globally
`pnpm add sax@next`	Install from the `next` tag
`pnpm add sax@3.0.0`	Specify version `3.0.0`

Supported package locations

Install from npm registry

`pnpm add package-name` will install the latest version of `package-name` from the npm registry by default.)

If executed in a workspace, the command will first try to check whether other projects in the workspace use the specified package. If so, the already used version range will be installed.

You may also install packages by:

- tag: `pnpm add express@nightly`
- version: `pnpm add express@1.0.0`
- version range: `pnpm add express@2 react@>=0.1.0 <0.2.0`

Install from the workspace

Note that when adding dependencies and working within a workspace, packages

will be installed from the configured sources, depending on whether or not `link-workspace-packages` is set, and use of the `workspace: range protocol`.

Install from local file system

There are two ways to install from the local file system:

1. from a tarball file (`.tar` , `.tar.gz` , or `.tgz`)
2. from a directory

Examples:

```
pnpm add ./package.tar.gz
pnpm add ./some-directory
```

When you install from a directory, a symlink will be created in the current project's `node_modules`, so it is the same as running `pnpm link`.

Install from remote tarball

The argument must be a fetchable URL starting with http:// or https://.

Example:

```
pnpm add https://github.com/indexzero/forever/tarball/v0.5.6
```

Install from Git repository

```
pnpm add <git remote url>
```

Installs the package from the hosted Git provider, cloning it with Git. You can use a protocol for certain Git providers. For example, `pnpm add github:user/repo`

You may install from Git by:

- latest commit from master: `pnpm add kevva/is-positive`
- commit: `pnpm add kevva/is-positive#97edff6f525f192a3f83cea1944765f769ae2678`
- branch: `pnpm add kevva/is-positive#master`
- version range: `pnpm add kevva/is-positive#semver:^2.0.0`

Options

--save-prod, -P

Install the specified packages as regular `dependencies`.

--save-dev, -D

Install the specified packages as `devDependencies`.

--save-optional, -O

Install the specified packages as `optionalDependencies`.

--save-exact, -E

Saved dependencies will be configured with an exact version rather than using pnpm's default semver range operator.

--save-peer

Using `--save-peer` will add one or more packages to `peerDependencies` and install them as dev dependencies.

--ignore-workspace-root-check

Adding a new dependency to the root workspace package fails, unless the `--ignore-workspace-root-check` or `-w` flag is used.

For instance, `pnpm add debug -w`.

--global, -g

Install a package globally.

--workspace

Only adds the new dependency if it is found in the workspace.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm audit

Checks for known security issues with the installed packages.

If security issues are found, try to update your dependencies via `pnpm update`. If a simple update does not fix all the issues, use overrides to force versions that are not vulnerable. For instance, if `lodash@<2.1.0` is vulnerable, use this overrides to force `lodash@^2.1.0`:

package.json

```
{
    pnpm: {
        overrides: {
            lodash@<2.1.0: ^2.1.0
        }
    }
}
```

Or alternatively, run `pnpm audit --fix`.

If you want to tolerate some vulnerabilities as they don't affect your project, you may use the `pnpm.auditConfig.ignoreCves` setting.

Options

--audit-level <severity>

- Type: **low**, **moderate**, **high**, **critical**
- Default: **low**

Only print advisories with severity greater than or equal to `<severity>`.

--fix

Add overrides to the `package.json` file in order to force non-vulnerable versions of the dependencies.

--json

Output audit report in JSON format.

--dev, -D

Only audit dev dependencies.

--prod, -P

Only audit production dependencies.

--no-optional

Don't audit `optionalDependencies`.

--ignore-registry-errors

If the registry responds with a non-200 status code, the process should exit with 0. So the process will fail only if the registry actually successfully responds with found vulnerabilities.

Go to TOC

pnpm bin

Prints the directory into which the executables of dependencies are linked.

Options

--global, -g

Prints the location of the globally installed executables.

Go to TOC

pnpm config

Aliases: `c`

Sets configuration values. Alias for `npm config`.

pnpm create

Create a project from a `create-*` or `@foo/create-*` starter kit.

Examples

```
pnpm create react-app my-app
```

pnpm deploy

Added in: v7.4.0

Deploy a package from a workspace.

Usage:

```
pnpm --filter=<deployed project name> deploy <target directory>
```

In case you build your project before deployment, also use the `--prod` option to skip `devDependencies` installation.

```
pnpm --filter=<deployed project name> --prod deploy <target directory>
```

Usage in a docker image. After building everything in your monorepo, do this in a second image that uses your monorepo base image as a build context or in an additional build stage:

```
# syntax=docker/dockerfile:1.4

FROM workspace as pruned
RUN pnpm --filter <your package name> --prod deploy pruned

FROM node:18-alpine
WORKDIR /app

ENV NODE_ENV=production

COPY --from=pruned /app/pruned .

ENTRYPOINT [node, index.js]
```

Files included in the deployed project

By default, all the files of the project are copied during deployment. The project's `package.json` may contain a files field to list the files and directories that should be copied.

Go to TOC

pnpm dlx

Fetches a package from the registry without installing it as a dependency, hotloads it, and runs whatever default command binary it exposes.

For example, to use `create-react-app` anywhere to bootstrap a react app without needing to install it under another project, you can run:

```
pnpm dlx create-react-app ./my-app
```

This will fetch `create-react-app` from the registry and run it with the given arguments.

You may also specify which exact version of the package you'd like to use:

```
pnpm dlx create-react-app@next ./my-app
```

Options

--package <name>

The package to install before running the command.

Example:

```
pnpm --package=@pnpm/meta-updater dlx meta-updater --help
pnpm --package=@pnpm/meta-updater@0 dlx meta-updater --help
```

Multiple packages can be provided for installation:

```
pnpm --package=yo --package=generator-webapp dlx yo webapp --skip-install
```

--silent, -s

Only the output of the executed command is printed.

Go to TOC

Go to TOC

pnpm doctor

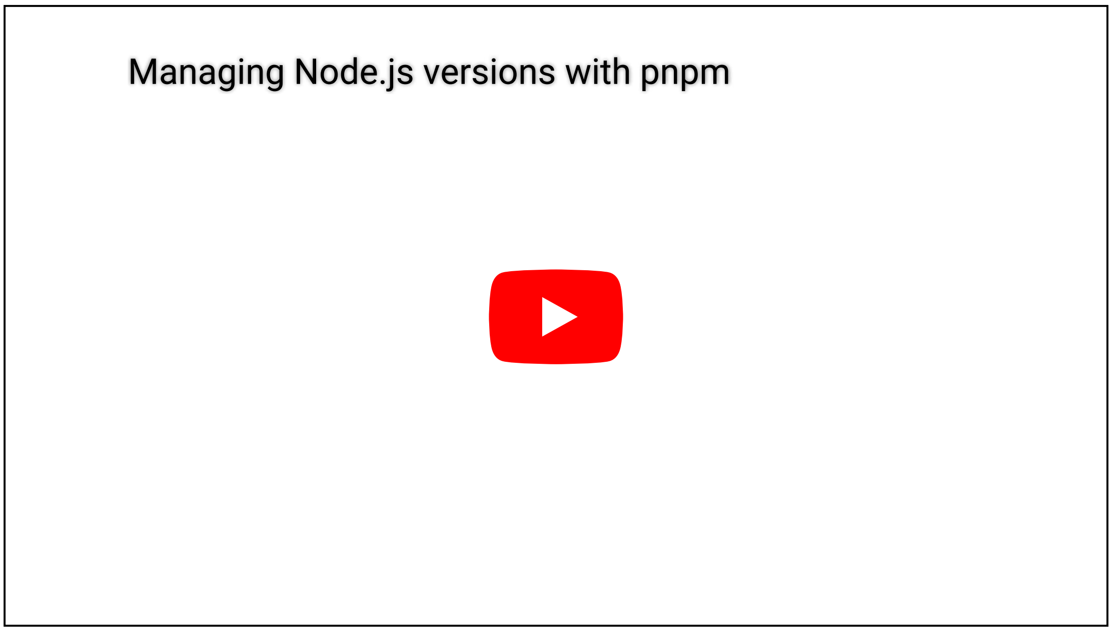
Added in: v7.14.0

Checks for known common issues with pnpm configuration.

Go to TOC

pnpm env <.html)

Manages the Node.js environment.



Commands

use

Install and use the specified version of Node.js

Install the LTS version of Node.js:

```
pnpm env use --global lts
pnpm env use --global argon
```

Install Node.js v16:

```
pnpm env use --global 16
```

Install a prerelease version of Node.js:

```
pnpm env use --global nightly
pnpm env use --global rc
pnpm env use --global 16.0.0-rc.0
pnpm env use --global rc/14
```

Install the latest version of Node.js:

```
pnpm env use --global latest
```

remove, rm

Added in: v7.10.0

Removes the specified version of Node.JS.

Usage example:

```
pnpm env remove --global 14.0.0
```

list, ls

Added in: v7.16.0

List Node.js versions available locally or remotely.

Print locally installed versions:

```
pnpm env list
```

Print remotely available Node.js versions:

```
pnpm env list --remote
```

Print remotely available Node.js v16 versions:

```
pnpm env list --remote 16
```

Options

--global, -g

The changes are made systemwide.

Go to TOC

pnpm exec

Execute a shell command in scope of a project.

`node_modules/.bin` is added to the `PATH`, so `pnpm exec` allows executing commands of dependencies.

Examples

If you have Jest as a dependency of your project, there is no need to install Jest globally, just run it with `pnpm exec`:

```
pnpm exec jest
```

The `exec` part is actually optional when the command is not in conflict with a builtin pnpm command, so you may also just run:

```
pnpm jest
```

Options

Any options for the `exec` command should be listed before the `exec` keyword. Options listed after the `exec` keyword are passed to the executed command.

Good. pnpm will run recursively:

```
pnpm -r exec jest
```

Bad, pnpm will not run recursively but `jest` will be executed with the `-r` option:

```
pnpm exec jest -r
```

--recursive, -r

Execute the shell command in every project of the workspace.

The name of the current package is available through the environment variable `PNPM_PACKAGE_NAME`.

Examples

Prune `node_modules` installations for all packages:

```
pnpm -r exec rm -rf node_modules
```

View package information for all packages. This should be used with the `--shell-mode` (or `-c`) option for the environment variable to work.

```
pnpm -rc exec pnpm view $PNPM_PACKAGE_NAME
```

--parallel

Completely disregard concurrency and topological sorting, running a given script immediately in all matching packages with prefixed streaming output. This is the preferred flag for long-running processes over many packages, for instance, a lengthy build process.

--shell-mode, -c

Runs the command inside of a shell. Uses `/bin/sh` on UNIX and `.html) on Windows.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm fetch

Fetch packages from a lockfile into virtual store, package manifest is ignored.

Usage scenario

This command is specifically designed to boost building a docker image.

You may have read the [official guide] to writing a Dockerfile for a Node.js app, if you didn't read it yet, you may want to read it first.

From that guide, we learn to write an optimized Dockerfile for projects using pnpm, which shall look like

```
FROM node:14

RUN curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm

# Files required by pnpm install
COPY .npmrc package.json pnpm-lock.yaml .pnpmfile.cjs ./

RUN pnpm install --frozen-lockfile --prod

# Bundle app source
COPY . .

EXPOSE 8080
CMD [ node, server.js ]
```

As long as there is no change to `.npmrc`, `package.json`, `pnpm-lock.yaml`, `.pnpmfile.cjs`, docker build cache is still valid up to the layer of `RUN pnpm install --frozen-lockfile --prod`, which cost most of the time when building a docker image.

However, modification to `package.json` may happen much more frequently than we expected, because it does not only contain dependencies, but may also contain the version number, scripts, and arbitrary configuration for any other tool.

It's also hard to maintain a Dockerfile that build a monorepo project, it may look like

```
FROM node:14

RUN curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm

# Files required by pnpm install
COPY .npmrc package.json pnpm-lock.yaml .pnpmfile.cjs ./

# for each sub-package, we have to add one extra step to copy its manifest
# to the right place, as docker have no way to filter out only package.json with
# single instruction
COPY packages/foo/package.json packages/foo/
COPY packages/bar/package.json packages/bar/

RUN pnpm install --frozen-lockfile --prod
```

```
# Bundle app source
COPY . .

EXPOSE 8080
CMD [ node, server.js ]
```

As you can see, the Dockerfile has to be updated when you add or remove sub-packages.

`pnpm fetch` will solve the above problem perfectly by providing the ability to fetch package to virtual store with information only from a lockfile.

```
FROM node:14

RUN curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm

# pnpm fetch does require only lockfile
COPY pnpm-lock.yaml ./

RUN pnpm fetch --prod


ADD . ./
RUN pnpm install -r --offline --prod


EXPOSE 8080
CMD [ node, server.js ]
```

It works for both simple project and monorepo project, `--offline` enforces pnpm not to communicate with package registry as all needed packages shall be already presented in the virtual store.

As long as the lockfile is not changed, the build cache is valid up to the layer `RUN pnpm install -r --offline --prod`, which will save you much time.

Options

--dev

Only development packages will be fetched

--prod

Development packages will not be fetched

official guide

Go to TOC

pnpm import

`pnpm import` generates a `pnpm-lock.yaml` from another package manager's lockfile. Supported source files:

- `package-lock.json`
- `npm-shrinkwrap.json`
- `yarn.lock`

Note that if you have workspaces you wish to import dependencies for, they will need to be declared in a pnpm-workspace.yaml file beforehand.

Go to TOC

pnpm init

Create a `package.json` file.

Go to TOC

pnpm install-test

Aliases: `it`

Runs `pnpm install` followed immediately by `pnpm test`. It takes exactly the same arguments as `pnpm install`

Go to TOC

pnpm install

Aliases: `i`

`pnpm install` is used to install all dependencies for a project.

In a CI environment, installation fails if a lockfile is present but needs an update.

Inside a workspace, `pnpm install` installs all dependencies in all the projects. If you want to disable this behavior, set the `recursive-install` setting to `false`.

TL;DR

Command	Meaning
`pnpm i --offline`	Install offline from the store only
`pnpm i --frozen-lockfile`	`pnpm-lock.yaml` is not updated
`pnpm i --lockfile-only`	Only `pnpm-lock.yaml` is updated

Options

--offline

- Default: **false**
- Type: **Boolean**

If `true`, pnpm will use only packages already available in the store. If a package won't be found locally, the installation will fail.

--prefer-offline

- Default: **false**
- Type: **Boolean**

If `true`, staleness checks for cached data will be bypassed, but missing data will be requested from the server. To force full offline mode, use `--offline`.

--prod, -P

pnpm will not install any package listed in `devDependencies` and will remove those insofar they were already installed, if the `NODE_ENV` environment variable is set to production. Use this flag to instruct pnpm to ignore `NODE_ENV` and take its production status from this flag instead.

--dev, -D

Only `devDependencies` are installed and `dependencies` are removed insofar they were already installed, regardless of the `NODE_ENV`.

--no-optional

`optionalDependencies` are not installed.

--lockfile-only

- Default: **false**
- Type: **Boolean**

When used, only updates `pnpm-lock.yaml` and `package.json`. Nothing gets written to the `node_modules` directory.

--fix-lockfile

Fix broken lockfile entries automatically.

--frozen-lockfile

- Default:
  - For non-CI: **false**
  - For CI: **true**, if a lockfile is present
- Type: **Boolean**

If `true`, pnpm doesn't generate a lockfile and fails to install if the lockfile is out of sync with the manifest / an update is needed or no lockfile is present.

This setting is `true` by default in [CI environments]. The following code is used to detect CI environments:

https://github.com/watson/ci-info/blob/44e98cebcdf4403f162195fbcf90b1f69fc6e047/index.js#L54-L61

```
exports.isCI = !!(
  env.CI || // Travis CI, CircleCI, Cirrus CI, GitLab CI, Appveyor, CodeShip,
dsari
  env.CONTINUOUS_INTEGRATION || // Travis CI, Cirrus CI
  env.BUILD_NUMBER || // Jenkins, TeamCity
  env.RUN_ID || // TaskCluster, dsari
  exports.name ||
  false
)
```

CI environments

--reporter=<name>

- Default:
  - For TTY stdout: **default**

- For non-TTY stdout: **append-only**
- Type: **default**, **append-only**, **ndjson**, **silent**

Allows you to choose the reporter that will log debug info to the terminal about

the installation progress.

- **silent** - no output is logged to the console, not even fatal errors
- **default** - the default reporter when the stdout is TTY
- **append-only** - the output is always appended to the end. No cursor manipulations are performed
- **ndjson** - the most verbose reporter. Prints all logs in ndjson format

If you want to change what type of information is printed, use the loglevel setting.

--use-store-server

- Default: **false**
- Type: **Boolean**

Starts a store server in the background. The store server will keep running after installation is done. To stop the store server, run `pnpm server stop`

--shamefully-hoist

- Default: **false**
- Type: **Boolean**

Creates a flat `node_modules` structure, similar to that of `npm` or `yarn`. **WARNING**: This is highly discouraged.

--ignore-scripts

- Default: **false**
- Type: **Boolean**

Do not execute any scripts defined in the project `package.json` and its dependencies.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm licenses

Added in: v7.17.0

> This command is experimental

Commands

list

List licenses for installed packages.

Go to TOC

pnpm link

Aliases: `ln`

Makes the current local package accessible system-wide, or in another location.

```
pnpm link <dir>
pnpm link --global
pnpm link --global <pkg>
```

Options

--dir <dir>, -C

- **Default**: Current working directory
- **Type**: Path string

Changes the link location to `<dir>`.

`pnpm link <dir>`

Links package from `<dir>` folder to node_modules of package from where you're executing this command or specified via `--dir` option.

`pnpm link --global`

Links package from location where this command was executed or specified via `--dir` option to global `node_modules`, so it can be referred from another package with `pnpm link --global <pkg>`.

`pnpm link --global <pkg>`

Links the specified package (`<pkg>`) from global `node_modules` to the `node_modules` of package from where this command was executed or specified via `--dir` option.

Go to TOC

pnpm list

Aliases: `ls`

This command will output all the versions of packages that are installed, as well as their dependencies, in a tree-structure.

Positional arguments are `name-pattern@version-range` identifiers, which will limit the results to only the packages named. For example, `pnpm list babel-* eslint-* semver@5`.

Options

--recursive, -r

Perform command on every package in subdirectories or on every workspace package, when executed inside a workspace.

--json

Log output in JSON format.

--long

Show extended information.

--parseable

Outputs package directories in a parseable format instead of their tree view.

--global, -g

List packages in the global install directory instead of in the current project.

--depth <number>

Max display depth of the dependency tree.

`pnpm ls --depth 0` will list direct dependencies only. `pnpm ls --depth -1` will list projects only. Useful inside a workspace when used with the `-r` option.

--prod, -P

Display only the dependency graph for packages in `dependencies` and `optionalDependencies`.

--dev, -D

Display only the dependency graph for packages in `devDependencies`.

--no-optional

Don't display packages from `optionalDependencies`.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm outdated

Checks for outdated packages. The check can be limited to a subset of the installed packages by providing arguments (patterns are supported).

Examples:

```
pnpm outdated
pnpm outdated *gulp-* @babel/core
```

Options

--recursive, -r

Check for outdated dependencies in every package found in subdirectories, or in every workspace package when executed inside a workspace.

--filter <package_selector>

Read more about filtering.

--global, -g

List outdated global packages.

--long

Print details.

--format <format>

Added in: v7.15.0

- Default: **table**
- Type: **table**, **list**, **json**

Prints the outdtaed dependencies in the given format.

--compatible

Prints only versions that satisfy specifications in `package.json`.

--dev, -D

Checks only `devDependencies`.

--prod, -P

Checks only `dependencies` and `optionalDependencies`.

--no-optional

Doesn't check `optionalDependencies`.

Go to TOC

pnpm pack

Create a tarball from a package.

Options

--pack-destination <dir>

Directory in which `pnpm pack` will save tarballs. The default is the current working directory.

Go to TOC

pnpm patch-commit

Added in: v7.4.0

Generate a patch out of a directory (inspired by a similar command in Yarn).

```
pnpm patch-commit <patchDir>
```

Go to TOC

pnpm patch

Added in: v7.4.0

Prepare a package for patching (inspired by a similar command in Yarn).

This command will cause a package to be extracted in a temporary directory intended to be editable at will.

Once you're done with your changes, run `pnpm patch-commit <path>` (with `<path>` being the temporary directory you received) to generate a patchfile and register it into your top-level manifest via the `patched-Dependencies` field.

Usage:

```
pnpm patch <pkg name>@<version>
```

Options

--edit-dir <dir>

Added in: v7.11.0

The package that needs to be patched will be extracted to this directory.

Go to TOC

pnpm prune

Removes unnecessary packages.

Options

--prod

Remove the packages specified in `devDependencies`.

--no-optional

Remove the packages specified in `optionalDependencies`.

> The prune command does not support recursive execution on a monorepo currently. To only install production-dependencies in a monorepo `node_modules` folders can be deleted and then re-installed with `pnpm install --prod`.

pnpm publish

Publishes a package to the registry.

```
pnpm [-r] publish [<tarball|folder>] [--tag <tag>]
     [--access <public|restricted>] [options]
```

When publishing a package inside a workspace the LICENSE file from the root of the workspace is packed with the package (unless the package has a license of its own).

You may override some fields before publish, using the publishConfig field in `package.json`. You also can use the `publishConfig.directory` to customize the published subdirectory (usually using third party build tools).

When running this command recursively (`pnpm -r publish`), pnpm will publish all the packages that have versions not yet published to the registry.

Options

--tag <tag>

Publishes the package with the given tag. By default, `pnpm publish` updates the `latest` tag.

For example:

```
# inside the foo package directory
pnpm publish --tag next
# in a project where you want to use the next version of foo
pnpm add foo@next
```

--access <public|restricted>

Tells the registry whether the published package should be public or restricted.

--no-git-checks

Don't check if current branch is your publish branch, clean, and up-to-date with remote.

--publish-branch

- Default: **master** and **main**
- Types: **String**

The primary branch of the repository which is used for publishing the latest changes.

--force

Try to publish packages even if their current version is already found in the registry.

--report-summary

Save the list of published packages to `pnpm-publish-summary.json`. Useful when some other tooling is used to report the list of published packages.

An example of a `pnpm-publish-summary.json` file:

```
{
  publishedPackages: [
    {
      name: foo,
      version: 1.0.0
    },
    {
      name: bar,
      version: 2.0.0
    }
  }
]
```

--dry-run

Does everything a publish would do except actually publishing to the registry.

--otp

When publishing packages that require two-factor authentication, this option can specify a one-time password.

--filter <package_selector>

Read more about filtering.

Configuration

You can also set `git-checks`, `publish-branch` options in the `.npmrc` file.

For example:

.npmrc

```
git-checks=false
publish-branch=production
```

Go to TOC

pnpm rebuild

Aliases: `rb`

Rebuild a package.

Options

--recursive, -r

This command runs the **pnpm rebuild** command in every package of the monorepo.

--filter <package_selector>

Read more about filtering.

pnpm -r, --recursive

Aliases: `m`, `multi`, `recursive`, `<command> -r`

Runs a command in every project of a workspace, when used with the following commands:

- `install`
- `list`
- `outdated`
- `publish`
- `rebuild`
- `remove`
- `unlink`
- `update`
- `why`

Runs a command in every project of a workspace, excluding the root project, when used with the following commands:

- `exec`
- `run`
- `test`
- `add`

If you want the root project be included even when running scripts, set the include-workspace-root setting to `true`.

Usage example:

```
pnpm -r publish
```

Options

--link-workspace-packages

- Default: **true**
- Type: **true, false, deep**

Link locally available packages in workspaces of a monorepo into `node_modules` instead of re-downloading them from the registry. This emulates functionality similar to `yarn workspaces`.

When this is set to deep, local packages can also be linked to subdependencies.

Be advised that it is encouraged instead to use npmrc for this setting, to enforce the same behaviour in all environments. This option exists solely so you may override that if necessary.

--workspace-concurrency

- Default: **4**
- Type: **Number**

Set the maximum number of tasks to run simultaneously. For unlimited concurrency use `Infinity`.

You can set the `workpace-concurrency` as `<= 0` and it will use amount of cores of the host as: `max(1, (number of cores) - abs(workspace-concurrency))`

--[no-]bail

- Default: **true**
- Type: **Boolean**

If true, stops when a task throws an error.

This config does not affect the exit code. Even if `--no-bail` is used, all tasks will finish but if any of the tasks fail, the command will exit with a non-zero code.

Example (run tests in every package, continue if tests fail in one of them):

```
pnpm -r --no-bail test
```

--[no-]sort

- Default: **true**
- Type: **Boolean**

When `true`, packages are sorted topologically (dependencies before dependents). Pass `--no-sort` to disable.

Example:

```
pnpm -r --no-sort test
```

--reverse

- Default: **false**
- Type: **boolean**

When `true`, the order of packages is reversed.

```
pnpm -r --reverse run clean
```

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm remove

Aliases: `rm`, `uninstall`, `un`

Removes packages from `node_modules` and from the project's `package.json`.

Options

--recursive, -r

When used inside a workspace removes a dependency (or dependencies) from every workspace package.

When used not inside a workspace, removes a dependency (or dependencies) from every package found in subdirectories.

--global, -g

Remove a global package.

--save-dev, -D

Only remove the dependency from `devDependencies`.

--save-optional, -O

Only remove the dependency from `optionalDependencies`.

--save-prod, -P

Only remove the dependency from `dependencies`.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm root

Prints the effective modules directory.

Options

--global, -g

The global package's modules directory is printed.

Go to TOC

pnpm run

Aliases: `run-script`

Runs a script defined in the package's manifest file.

Examples

Let's say you have a `watch` script configured in your `package.json`, like so:

```
scripts: {
    watch: webpack --watch
}
```

You can now run that script by using `pnpm run watch`! Simple, right? Another thing to note for those that like to save keystrokes and time is that

all scripts get aliased in as pnpm commands, so ultimately `pnpm watch` is just shorthand for `pnpm run watch` (ONLY for scripts that do not share the same name as already existing pnpm commands).

Details

In addition to the shell's pre-existing `PATH`, `pnpm run` includes `node_modules/.bin` in the `PATH` provided to `scripts`. This means that so long as you have a package installed, you can use it in a script like a regular command. For example, if you have `eslint` installed, you can write up a script like so:

```
lint: eslint src --fix
```

And even though `eslint` is not installed globally in your shell, it will run.

For workspaces, `<workspace root>/node_modules/.bin` is also added to the `PATH`, so if a tool is installed in the workspace root, it may be called in any workspace package's `scripts`.

Differences with `npm run`

By default, pnpm doesn't run arbitrary `pre` and `post` hooks for user-defined scripts (such as `prestart`). This behavior, inherited from npm, caused scripts to be implicit rather than explicit, obfuscating the execution flow. It also led to surprising executions with `pnpm serve` also running `pnpm preserve`.

If for some reason you need the pre/post scripts behavior of npm, use the `enable-pre-post-scripts` option.

Options

Any options for the `run` command should be listed before the script's name. Options listed after the script's name are passed to the executed script.

All these will run pnpm CLI with the `--silent` option:

```
pnpm run --silent watch
pnpm --silent run watch
pnpm --silent watch
```

Any arguments after the command's name are added to the executed script. So if `watch` runs `webpack --watch`, then this command:

```
pnpm run watch --no-color
```

will run:

```
webpack --watch --no-color
```

script-shell

- Default: **null**
- Type: **path**

The shell to use for scripts run with the `pnpm run` command.

For instance, to force usage of Git Bash on Windows:

```
pnpm config set script-shell C:\\Program Files\\git\\bin\\bash.exe
```

shell-emulator

- Default: **false**
- Type: **Boolean**

When `true`, pnpm will use a JavaScript implementation of a [bash-like shell] to execute scripts.

This option simplifies cross-platform scripting. For instance, by default, the next script will fail on non-POSIX-compliant systems:

```
scripts: {
  test: NODE_ENV=test node test.js
}
```

But if the `shell-emulator` setting is set to `true`, it will work on all platforms.

bash-like shell

--recursive, -r

This runs an arbitrary command from each package's scripts object. If a package doesn't have the command, it is skipped. If none of the packages have the command, the command fails.

--if-present

You can use the `--if-present` flag to avoid exiting with a non-zero exit code when the script is undefined. This lets you run potentially undefined scripts without breaking the execution chain.

--parallel

Completely disregard concurrency and topological sorting, running a given script immediately in all matching packages with prefixed streaming output. This is the preferred flag for long-running processes over many packages, for instance, a lengthy build process.

--stream

Stream output from child processes immediately, prefixed with the originating package directory. This allows output from different packages to be interleaved.

--aggregate-output

Aggregate output from child processes that are run in parallel, and only print output when the child process is finished. It makes reading large logs after running `pnpm -r <command>` with `--parallel` or with `--workspace-concurrency=<number>` much easier (especially on CI). Only `--reporter=append-only` is supported.

enable-pre-post-scripts

- Default: **false**
- Type: **Boolean**

When `true`, pnpm will run any pre/post scripts automatically. So running `pnpm foo` will be like running `pnpm prefoo && pnpm foo && pnpm postfoo`.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm server

Manage a store server.

Commands

pnpm server start

Starts a server that performs all interactions with the store. Other commands will delegate any store-related tasks to this server.

pnpm server stop

Stops the store server.

pnpm server status

Prints information about the running server.

Options

--background

- Default: **false**
- Type: **Boolean**

Runs the server in the background, similar to daemonizing on UNIX systems.

--network-concurrency

- Default: **null**
- Type: **Number**

The maximum number of network requests to process simultaneously.

--protocol

- Default: **auto**
- Type: **auto**, **tcp**, **ipc**

The communication protocol used by the server. When this is set to `auto`, IPC is used on all systems except for Windows, which uses TCP.

--port

- Default: **5813**
- Type: **port number**

The port number to use when TCP is used for communication. If a port is specified and the protocol is set to `auto`, regardless of system type, the protocol is automatically set to use TCP.

--store-dir

- Default: **<home>/.pnpm-store**
- Type: **Path**

The directory to use for the content addressable store.

--[no-]lock

- Default: **false**
- Type: **Boolean**

Set whether to make the package store immutable to external processes while the server is running or not.

--ignore-stop-requests

- Default: **false**
- Type: **Boolean**

Prevents you from stopping the server using `pnpm server stop`.

--ignore-upload-requests

- Default: **false**
- Type: **Boolean**

Prevents creating a new side effect cache during install.

Go to TOC

pnpm setup

This command is used by the standalone installation scripts of pnpm. For instance, in https://get.pnpm.io/install.sh.

Setup does the following actions:

- creates a home directory for the pnpm CLI
- adds the pnpm home directory to the `PATH` by updating the shell configuration file
- copies the pnpm executable to the pnpm home directory

Go to TOC

pnpm start

Aliases: `run start`

Runs an arbitrary command specified in the package's `start` property of its `scripts` object. If no `start` property is specified on the `scripts` object, it will attempt to run `node server.js` as a default, failing if neither are present.

The intended usage of the property is to specify a command that starts your program.

Go to TOC

pnpm store

Managing the package store.

Commands

status

Checks for modified packages in the store.

Returns exit code 0 if the content of the package is the same as it was at the time of unpacking.

add

Functionally equivalent to [`pnpm add`], except this adds new packages to the store directly without modifying any projects or files outside of the store.

`pnpm add`

prune

Removes *unreferenced packages* from the store.

Unreferenced packages are packages that are not used by any projects on the system. Packages can become unreferenced after most installation operations, for instance when dependencies are made redundant.

For example, during `pnpm install`, package `foo@1.0.0` is updated to `foo@1.0.1`. pnpm will keep `foo@1.0.0` in the store, as it does not automatically remove packages. If package `foo@1.0.0` is not used by any other project on the system, it becomes unreferenced. Running `pnpm store prune` would remove `foo@1.0.0` from the store.

Running `pnpm store prune` is not harmful and has no side effects on your projects. If future installations require removed packages, pnpm will download them again.

It is best practice to run `pnpm store prune` occasionally to clean up the store, but not too frequently. Sometimes, unreferenced packages become required again. This could occur when switching branches and installing older dependencies, in which case pnpm would need to re-download all removed packages, briefly slowing down the installation process.

Please note that this command is prohibited when a [store server] is running.

store server

path

Returns the path to the active store directory.

Go to TOC

pnpm test

Aliases: `run test`, `t`, `tst`

Runs an arbitrary command specified in the package's `test` property of its `scripts` object.

The intended usage of the property is to specify a command that runs unit or integration testing for your program.

Go to TOC

pnpm unlink

Unlinks a system-wide package (inverse of `pnpm link`

If called without arguments, all linked dependencies will be unlinked.

This is similar to `yarn unlink`, except pnpm re-installs the dependency after removing the external link.

Options

--recursive, -r

Unlink in every package found in subdirectories or in every workspace package, when executed inside a workspace

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm update

Aliases: `up`, `upgrade`

`pnpm update` updates packages to their latest version based on the specified range.

When used without arguments, updates all dependencies.

TL;DR

Command	Meaning
`pnpm up`	Updates all dependencies, adhering to ranges specified in `package.json`
`pnpm up --latest`	Updates all dependencies, ignoring ranges specified in `package.json`
`pnpm up foo@2`	Updates `foo` to the latest version on v2
`pnpm up @babel/*`	Updates all dependencies under the `@babel` scope
`pnpm up @babel/*`	Updates all dependencies under the `@babel` scope

Selecting dependencies with patterns

You can use patterns to update specific dependencies.

Update all `babel` packages:

```
pnpm update @babel/*
```

Update all dependencies, except `webpack`:

```
pnpm update !webpack
```

Patterns may also be combined, so the next command will update all `babel` packages, except `core`:

```
pnpm update @babel/* !@babel/core
```

Options

--recursive, -r

Concurrently runs update in all subdirectories with a `package.json` (excluding node_modules).

Usage examples:

```
pnpm --recursive update
# updates all packages up to 100 subdirectories in depth
pnpm --recursive update --depth 100
# update typescript to the latest version in every package
pnpm --recursive update typescript@latest
```

--latest, -L

Ignores the version range specified in `package.json`. Instead, the version specified by the `latest` tag will be used (potentially upgrading the packages across major versions).

--global, -g

Update global packages.

--workspace

Tries to link all packages from the workspace. Versions are updated to match the versions of packages inside the workspace.

If specific packages are updated, the command will fail if any of the updated dependencies are not found inside the workspace. For instance, the following command fails if `express` is not a workspace package:

```
pnpm up -r --workspace express
```

--prod, -P

Only update packages in `dependencies` and `optionalDependencies`.

--dev, -D

Only update packages in `devDependencies`.

--no-optional

Don't update packages in `optionalDependencies`.

--interactive, -i

Show outdated dependencies and select which ones to update.

--filter <package_selector>

Read more about filtering.

Go to TOC

pnpm why

Shows all packages that depend on the specified package.

Options

--recursive, -r

Show the dependency tree for the specified package on every package in subdirectories or on every workspace package when executed inside a workspace.

--json

Show information in JSON format.

--long

Show verbose output.

--parseable

Show parseable output instead of tree view.

--global, -g

List packages in the global install directory instead of in the current project.

--prod, -P

Only display the dependency tree for packages in `dependencies` .

--dev, -D

Only display the dependency tree for packages in `devDependencies` .

--filter <package_selector>

Read more about filtering.

Command line tab-completion

Unlike other popular package managers, which usually require plugins, pnpm supports command line tab-completion for Bash, Zsh, Fish, and similar shells.

To setup autocompletion, run:

```
pnpm install-completion
```

The CLI will ask for which shell to generate the autocompletion script. Alternatively, the target shell may be specified in the command line:

```
pnpm install-completion zsh
```

To see examples of completion, read [this article].

this article

Fig (on macOS only)

You can get IDE-style autocompletions for pnpm with [Fig]. It works in Bash, Zsh, and Fish.

To install, run:

```
brew install fig
```

Fig

Go to TOC

Configuring

pnpm uses [npm's configuration] formats. Hence, you should set configuration the same way you would for npm. For example,

```
pnpm config set store-dir /path/to/.pnpm-store
```

If no store is configured, then pnpm will automatically create a store on the same drive. If you need pnpm to work across multiple hard drives or filesystems, please read [the FAQ].

Furthermore, pnpm uses the same configuration that npm uses for doing installations. If you have a private registry and npm is configured to work with it, pnpm should be able to authorize requests as well, with no additional configuration.

In addition to those options, pnpm also allows you to use all parameters that are flags (for example `--filter` or `--workspace-concurrency`) as options:

```
workspace-concurrency = 1
filter = @my-scope/*
```

See the [`config` command] for more information.

npm's configuration the FAQ `config` command

Continuous Integration

pnpm can easily be used in various continuous integration systems.

Travis

On [Travis CI], you can use pnpm for installing your dependencies by adding this to your `.travis.yml` file:

.travis.yml

```yaml
cache:
  npm: false
  directories:
    - ~/.pnpm-store
before_install:
  - curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
  - pnpm config set store-dir ~/.pnpm-store
install:
  - pnpm install
```

Travis CI

Semaphore

On [Semaphore], you can use pnpm for installing and caching your dependencies by adding this to your `.semaphore/semaphore.yml` file:

.semaphore/semaphore.yml

```yaml
version: v1.0
name: Semaphore CI pnpm example
agent:
  machine:
    type: e1-standard-2
    os_image: ubuntu1804
blocks:
  - name: Install dependencies
    task:
      jobs:
        - name: pnpm install
          commands:
            - curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
            - checkout
            - cache restore node-$(checksum pnpm-lock.yaml)
            - pnpm install
            - cache store node-$(checksum pnpm-lock.yaml) $(pnpm store path)
```

Semaphore

AppVeyor

On [AppVeyor], you can use pnpm for installing your dependencies by adding this to your `appveyor.yml`:

appveyor.yml

```yaml
install:
  - ps: Install-Product node $env:nodejs_version
  - curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
  - pnpm install
```

AppVeyor

GitHub Actions

On GitHub Actions, you can use pnpm for installing and caching your dependencies like so (belongs in `.github/workflows/NAME.yml`):

.github/workflows/NAME.yml

```yaml
name: pnpm Example Workflow
on:
  push:
jobs:
  build:
    runs-on: ubuntu-20.04
    strategy:
      matrix:
        node-version: [15]
    steps:
    - uses: actions/checkout@v3
    - uses: pnpm/action-setup@v2
      with:
        version: 7
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'pnpm'
    - name: Install dependencies
      run: pnpm install
```

Caching packages dependencies with `actions/setup-node@v2` requires you to install pnpm with version **6.10+**.

GitLab CI

On GitLab, you can use pnpm for installing and caching your dependencies like so (belongs in `.gitlab-ci.yml`):

.gitlab-ci.yml

```yaml
stages:
  - build

build:
  stage: build
  image: node:14.16.0-buster
  before_script:
    - curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
    - pnpm config set store-dir .pnpm-store
  script:
    - pnpm install # install dependencies
  cache:
    key:
      files:
        - pnpm-lock.yaml
    paths:
      - .pnpm-store
```

Bitbucket Pipelines

You can use pnpm for installing and caching your dependencies:

.bitbucket-pipelines.yml

```yaml
definitions:
  caches:
    pnpm: $BITBUCKET_CLONE_DIR/.pnpm-store

pipelines:
  pull-requests:
    "**":
      - step:
          name: Build and test
          image: node:14.16.0
          script:
            - curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
            - pnpm install
            - pnpm run build # Replace with your build/test…etc. commands
          caches:
            - pnpm
```

Azure Pipelines

On Azure Pipelines, you can use pnpm for installing and caching your dependencies by adding this to your `azure-pipelines.yml`:

azure-pipelines.yml

```yaml
variables:
  pnpm_config_cache: $(Pipeline.Workspace)/.pnpm-store

steps:
  - task: Cache@2
    inputs:
      key: 'pnpm | $(Agent.OS) | pnpm-lock.yaml'
      path: $(pnpm_config_cache)
    displayName: Cache pnpm
  - script: |
      curl -f https://get.pnpm.io/v6.16.js | node - add --global pnpm@7
      pnpm config set store-dir $(pnpm_config_cache)
    displayName: Setup pnpm

  - script: |
      pnpm install
      pnpm run build
    displayName: pnpm install and build
```

CircleCI

On CircleCI, you can use pnpm for installing and caching your dependencies by adding this to your `.circleci/config.yml`:

.circleci/config.yml

```yaml
version: 2.1

jobs:
  build: # this can be any name you choose
    docker:
      - image: node:18
    resource_class: large
    parallelism: 10

    steps:
      - checkout
      - restore_cache:
          name: Restore pnpm Package Cache
          keys:
            - pnpm-packages-{{ checksum pnpm-lock.yaml }}
      - run:
          name: Install pnpm package manager
          command: |
            curl -L https://pnpm.js.org/pnpm.js | node - add --global pnpm@7
      - run:
          name: Install Dependencies
          command: |
            pnpm install
      - save_cache:
          name: Save pnpm Package Cache
          key: pnpm-packages-{{ checksum pnpm-lock.yaml }}
          paths:
            - node_modules
```

Go to TOC

Error Codes

ERR_PNPM_UNEXPECTED_STORE

A modules directory is present and is linked to a different store directory.

If you changed the store directory intentionally, run `pnpm install` and pnpm will reinstall the dependencies using the new store.

ERR_PNPM_NO_MATCHING_VERSION_INSIDE_WORKSPAC

A project has a workspace dependency that does not exist in the workspace.

For instance, package `foo` has `bar@1.0.0` in the `dependencies`:

```
{
  name: foo,
  version: 1.0.0,
  dependencies: {
    bar: workspace:1.0.0
  }
}
```

However, there is only `bar@2.0.0` in the workspace, so `pnpm install` will fail.

To fix this error, all dependencies that use the [workspace protocol] should be updated to use versions of packages that are present in the workspace. This can be done either manually or using the `pnpm -r update` command.

workspace protocol

ERR_PNPM_PEER_DEP_ISSUES

`pnpm install` will fail if the project has unresolved peer dependencies or the peer dependencies are not matching the wanted ranges. To fix this, install the missing peer dependencies.

You may also selectively ignore these errors using the pnpm.peerDependencyRules.ignoreMissing and pnpm.peerDependencyRules.allowedVersions fields in `package.json`.

ERR_PNPM_OUTDATED_LOCKFILE

This error happens when installation cannot be performed without changes to the lockfile. This might happen in a CI environment if someone has changed a `package.json` file in the repository without running `pnpm install` afterwards. Or someone forgot to commit the changes to the lockfile.

To fix this error, just run `pnpm install` and commit the changes to the lockfile.

Go to TOC

Frequently Asked Questions

Why does my `node_modules` folder use disk space if packages are stored in a global store?

pnpm creates [hard links] from the global store to the project's `node_modules` folders. Hard links point to the same place on the disk where the original files are. So, for example, if you have `foo` in your project as a dependency and it occupies 1MB of space, then it will look like it occupies 1MB of space in the project's `node_modules` folder and the same amount of space in the global store. However, that 1MB is *the same space* on the disk addressed from two different locations. So in total `foo` occupies 1MB, not 2MB.

hard links

For more on this subject:

- Why do hard links seem to take the same space as the originals?)
- A thread from the pnpm chat room)
- An issue in the pnpm repo)

Does it work on Windows?

Short answer: Yes. Long answer: Using symbolic linking on Windows is problematic to say the least, however, pnpm has a workaround. For Windows, we use [junctions] instead.

junctions

But the nested `node_modules` approach is incompatible with Windows?

Early versions of npm had issues because of nesting all `node_modules` (see [this issue]). However, pnpm does not create deep folders, it stores all packages flatly and uses symbolic links to create the dependency tree structure.

this issue

What about circular symlinks?

Although pnpm uses linking to put dependencies into `node_modules` folders, circular symlinks are avoided because parent packages are placed into the same `node_modules` folder in which their dependencies are. So `foo`'s dependencies are not in `foo/node_modules`, but `foo` is in `node_modules` together with its own dependencies.

Why have hard links at all? Why not symlink directly to the global store?

One package can have different sets of dependencies on one machine.

In project **A** `foo@1.0.0` can have a dependency resolved to `bar@1.0.0`, but in project **B** the same dependency of `foo` might resolve to `bar@1.1.0`; so, pnpm hard links `foo@1.0.0` to every project where it is used, in order to create different sets of dependencies for it.

Direct symlinking to the global store would work with Node's `--preserve-symlinks` flag, however, that approach comes with a plethora of its own issues, so we decided to stick with hard links. For more details about why this decision was made, see [this issue][eps-issue].

eps-issue

Does pnpm work across multiple drives or filesystems?

The package store should be on the same drive and filesystem as installations, otherwise packages will be copied, not linked. This is due to a limitation in how hard linking works, in that a file on one filesystem cannot address a location in another. See [Issue #712] for more details.

pnpm functions differently in the 2 cases below:

Issue #712

Store path is specified

If the store path is specified via the store config then copying occurs between the store and any projects that are on a different disk.

If you run `pnpm install` on disk `A`, then the pnpm store must be on disk `A`. If the pnpm store is located on disk `B`, then all required packages will be directly copied to the project location instead of being linked. This severely inhibits the storage and performance benefits of pnpm.

Store path is NOT specified

If the store path is not set, then multiple stores are created (one per drive or filesystem).

If installation is run on disk `A`, the store will be created on `A` `.pnpm-store` under the filesystem root. If later the installation is run on disk `B`, an independent store will be created on `B` at `.pnpm-store`. The projects would still maintain the benefits of pnpm, but each drive may have redundant packages.

What does `pnpm` stand for?

`pnpm` stands for `performant npm`. @rstacruz came up with the name.)

`pnpm` does not work with <YOUR-PROJECT-HERE>?

In most cases it means that one of the dependencies require packages not declared in `package.json`. It is a common mistake caused by flat `node_modules`. If this happens, this is an error in the dependency and the dependency should be fixed. That might take time though, so pnpm supports workarounds to make the buggy packages work.

Solution 1

In case there are issues, you can use the `node-linker=hoisted` setting. This creates a flat `node_modules` structure similar to the one created by `npm`.

Solution 2

In the following example, a dependency does **not** have the `iterall` module in its own list of deps.

The easiest solution to resolve missing dependencies of the buggy packages is to **add `iterall` as a dependency to our project's `package.json`**.

You can do so, by installing it via `pnpm add iterall`, and will be automatically added to your project's `package.json`.

```
  dependencies: {
    ...
    iterall: ^1.2.2,
    ...
  }
```

Solution 3

One of the solutions is to use hooks for adding the missing dependencies to the package's `package.json`.

An example was [Webpack Dashboard] which wasn't working with `pnpm`. It has since been resolved such that it works with `pnpm` now.

It used to throw an error:

```
Error: Cannot find module 'babel-traverse'
  at /node_modules/inspectpack@2.2.3/node_modules/inspectpack/lib/actions/parse
```

The problem was that `babel-traverse` was used in `inspectpack` which was used by `webpack-dashboard`, but `babel-traverse` wasn't specified in `inspectpack`'s `package.json`. It still worked with `npm` and `yarn` because they create flat `node_modules`.

The solution was to create a `.pnpmfile.cjs` with the following contents:

```
module.exports = {
  hooks: {
    readPackage: (pkg) => {
      if (pkg.name === inspectpack) {
        pkg.dependencies['babel-traverse'] = '^6.26.0';
      }
```

```
        return pkg;
      }
    }
  };
```

After creating a `.pnpmfile.cjs`, delete `pnpm-lock.yaml` only - there is no need to delete `node_modules`, as pnpm hooks only affect module resolution. Then, rebuild the dependencies & it should be working.

Webpack Dashboard

Feature Comparison

Feature	pnpm	Yarn	npm
Workspace support	✔	✔	✔
Isolated `node_modules`	✔ - The default	✔	❌
Hoisted `node_modules`	✔	✔	✔ - The default
Autoinstalling peers	✔ - Via auto-install-peers=true	❌	✔
Plug'n'Play	✔	✔ - The default	❌
Zero-Installs	❌	✔	❌
Patching dependencies	✔	✔	❌
Managing Node.js versions	✔	❌	❌
Has a lockfile	✔ - `pnpm-lock.yaml`	✔ - `yarn.lock`	✔ - `package-lock.json`
Overrides support	✔	✔ - Via resolutions	✔
Content-addressable storage	✔	❌	❌
Dynamic package execution	✔ - Via `pnpm dlx`	✔ - Via `yarn dlx`	✔ - Via `npx`
Side-effects cache	✔	❌	❌
Listing licenses	✔ - Via `pnpm licenses list`	✔ - Via a plugin	❌

Go to TOC

Filtering

Filtering allows you to restrict commands to specific subsets of packages.

pnpm supports a rich selector syntax for picking packages by name or by relation.

Selectors may be specified via the `--filter` (or `-F`) flag:

```
pnpm --filter <package_selector> <command>
```

Matching

--filter <package_name>

To select an exact package, just specify its name (`@scope/pkg`) or use a pattern to select a set of packages (`@scope/*`).

Examples:

```
pnpm --filter @babel/core test
pnpm --filter @babel/* test
pnpm --filter *core test
```

Specifying the scope of the package is optional, so `--filter=core` will pick `@babel/core` if `core` is not found. However, if the workspace has multiple packages with the same name (for instance, `@babel/core` and `@types/core`), then filtering without scope will pick nothing.

--filter <package_name>...

To select a package and its dependencies (direct and non-direct), suffix the package name with an ellipsis: `<package_name>...`. For instance, the next command will run tests of `foo` and all of its dependencies:

```
pnpm --filter foo... test
```

You may use a pattern to select a set of root packages:

```
pnpm --filter @babel/preset-*... test
```

--filter <package_name>^...

To ONLY select the dependencies of a package (both direct and non-direct), suffix the name with the afore-mentioned ellipsis preceded by a chevron. For instance, the next command will run tests for all of `foo`'s dependencies:

```
pnpm --filter foo^... test
```

--filter ...<package_name>

To select a package and its dependent packages (direct and non-direct), prefix the package name with an ellipsis: `...<package_name>`. For instance, this will run the tests of `foo` and all packages dependent on it:

```
pnpm --filter ...foo test
```

--filter ...^<package_name>

To ONLY select a package's dependents (both direct and non-direct), prefix the package name with an ellipsis followed by a chevron. For instance, this will run tests for all packages dependent on `foo`:

```
pnpm --filter ...^foo test
```

--filter ./<glob>, --filter {<glob>}

A glob pattern relative to the current working directory matching projects.

```
pnpm --filter ./packages/** <.html)
```

Includes all projects that are under the specified directory.

It may be used with the ellipsis and chevron operators to select dependents/dependencies as well:

```
pnpm --filter ...{<directory>} <.html)
pnpm --filter {<directory>}... <.html)
pnpm --filter ...{<directory>}... <.html)
```

It may also be combined with `[<since>]`. For instance, to select all changed projects inside a directory:

```
pnpm --filter {packages/**}[origin/master] <.html)
pnpm --filter ...{packages/**}[origin/master] <.html)
pnpm --filter {packages/**}[origin/master]... <.html)
pnpm --filter ...{packages/**}[origin/master]... <.html)
```

Or you may select all packages from a directory with names matching the given pattern:

```
pnpm --filter @babel/*{components/**} <.html)
pnpm --filter @babel/*{components/**}[origin/master] <.html)
pnpm --filter ...@babel/*{components/**}[origin/master] <.html)
```

--filter [<since>]

Selects all the packages changed since the specified commit/branch. May be suffixed or prefixed with `...` to include dependencies/dependents.

For example, the next command will run tests in all changed packages since `master` and on any dependent packages:

```
pnpm --filter ...[origin/master] test
```

Excluding

Any of the filter selectors may work as exclusion operators when they have a leading !. In zsh (and possibly other shells), ! should be escaped: `\!`.

For instance, this will run a command in all projects except for `foo`:

```
pnpm --filter=!foo <.html)
```

And this will run a command in all projects that are not under the `lib` directory:

```
pnpm --filter=!./lib <.html)
```

Multiplicity

When packages are filtered, every package is taken that matches at least one of the selectors. You can use as many filters as you want:

```
pnpm --filter ...foo --filter bar --filter baz... test
```

--filter-prod <filtering_pattern>

Acts the same a `--filter` but omits `devDependencies` when selecting dependency projects from the workspace.

--test-pattern <glob>

`test-pattern` allows detecting whether the modified files are related to tests. If they are, the dependent packages of such modified packages are not included.

This option is useful with the changed since filter. For instance, the next command will run tests in all changed packages, and if the changes are in the source code of the package, tests will run in the dependent packages as well:

```
pnpm --filter=...[origin/master] --test-pattern=test/* test
```

--changed-files-ignore-pattern <glob>

Allows to ignore changed files by glob patterns when filtering for changed projects since the specified commit/branch.

Usage example:

```
pnpm --filter=...[origin/master] --changed-files-ignore-pattern=**/README.html)
run build
```

Go to TOC

Working with Git

Lockfiles

You should always commit the lockfile (`pnpm-lock.yaml`). This is for a multitude of reasons, the primary of which being:

- it enables faster installation for CI and production environments, due to being able to skip package resolution
- it enforces consistent installations and resolution between development, testing, and production environments, meaning the packages used in testing and production will be exactly the same as when you developed your project

Merge conflicts

pnpm can automatically resolve merge conflicts in `pnpm-lock.yaml`. If you have conflicts, just run `pnpm install` and commit the changes.

Be warned, however. It is advised that you review the changes prior to staging a commit, because we cannot guarantee that pnpm will choose the correct head - it instead builds with the most updated of lockfiles, which is ideal in most cases.

Go to TOC

How peers are resolved

One of the best features of pnpm is that in one project, a specific version of a package will always have one set of dependencies. There is one exception from this rule, though - packages with [peer dependencies].

peer dependencies

Peer dependencies are resolved from dependencies installed higher in the dependency graph, since they share the same version as their parent. That means that if `foo@1.0.0` has two peers (`bar@^1` and `baz@^1`) then it might have multiple different sets of dependencies in the same project.

```
- foo-parent-1
  - bar@1.0.0
  - baz@1.0.0
  - foo@1.0.0
- foo-parent-2
  - bar@1.0.0
  - baz@1.1.0
  - foo@1.0.0
```

In the example above, `foo@1.0.0` is installed for `foo-parent-1` and `foo-parent-2`. Both packages have `bar` and `baz` as well, but they depend on different versions of `baz`. As a result, `foo@1.0.0` has two different sets of dependencies: one with `baz@1.0.0` and the other one with `baz@1.1.0`. To support these use cases, pnpm has to hard link `foo@1.0.0` as many times as there are different dependency sets.

Normally, if a package does not have peer dependencies, it is hard linked to a `node_modules` folder next to symlinks of its dependencies, like so:

```
node_modules
└── .pnpm
    ├── foo@1.0.0
    │   └── node_modules
    │       ├── foo
    │       ├── qux   -> ../../qux@1.0.0/node_modules/qux
    │       └── plugh -> ../../plugh@1.0.0/node_modules/plugh
    ├── qux@1.0.0
    ├── plugh@1.0.0
```

However, if `foo` has peer dependencies, there may be multiple sets of dependencies for it, so we create different sets for different peer dependency resolutions:

```
node_modules
└── .pnpm
    ├── foo@1.0.0_bar@1.0.0+baz@1.0.0
    │   └── node_modules
    │       ├── foo
    │       ├── bar   -> ../../bar@1.0.0/node_modules/bar
    │       ├── baz   -> ../../baz@1.0.0/node_modules/baz
    │       ├── qux   -> ../../qux@1.0.0/node_modules/qux
    │       └── plugh -> ../../plugh@1.0.0/node_modules/plugh
    ├── foo@1.0.0_bar@1.0.0+baz@1.1.0
    │   └── node_modules
    │       ├── foo
    |       ├── bar   -> ../../bar@1.0.0/node_modules/bar
    |       ├── baz   -> ../../baz@1.1.0/node_modules/baz
    |       ├── qux   -> ../../qux@1.0.0/node_modules/qux
    |       └── plugh -> ../../plugh@1.0.0/node_modules/plugh
    ├── bar@1.0.0
    ├── baz@1.0.0
    ├── baz@1.1.0
    ├── qux@1.0.0
    ├── plugh@1.0.0
```

We create symlinks either to the `foo` that is inside `foo@1.0.0_bar@1.0.0+baz@1.0.0` or to the one in `foo@1.0.0_bar@1.0.0+baz@1.1.0`. As a consequence, the Node.js module resolver will find the correct peers.

If a package has no peer dependencies but has dependencies with peers that are resolved higher in the graph, then that transitive package can appear in the project with different sets of dependencies. For instance, there's package `a@1.0.0` with a single dependency `b@1.0.0`. `b@1.0.0` has a peer dependency `c@^1`. `a@1.0.0` will never resolve the peers of `b@1.0.0`, so it becomes dependent from the peers of `b@1.0.0` as well.

Here's how that structure will look in `node_modules`. In this example, `a@1.0.0` will need to appear twice in the project's `node_modules` - resolved once with `c@1.0.0` and again with `c@1.1.0`.

```
node_modules
└── .pnpm
    ├── a@1.0.0_c@1.0.0
    |   └── node_modules
    |       ├── a
    |       └── b -> ../../b@1.0.0_c@1.0.0/node_modules/b
    ├── a@1.0.0_c@1.1.0
    |   └── node_modules
    |       ├── a
    |       └── b -> ../../b@1.0.0_c@1.1.0/node_modules/b
    ├── b@1.0.0_c@1.0.0
    |   └── node_modules
    |       ├── b
    |       └── c -> ../../c@1.0.0/node_modules/c
    ├── b@1.0.0_c@1.1.0
    |   └── node_modules
    |       ├── b
    |       └── c -> ../../c@1.1.0/node_modules/c
    ├── c@1.0.0
    ├── c@1.1.0
```

Go to TOC

Installation

Using a standalone script

You may install pnpm even if you don't have Node.js installed, using the following scripts.

On Windows

Using PowerShell:

```
iwr https://get.pnpm.io/install.ps1 -useb | iex
```

On POSIX systems

```
curl -fsSL https://get.pnpm.io/install.sh | sh -
```

If you don't have curl installed, you would like to use wget:

```
wget -qO- https://get.pnpm.io/install.sh | sh -
```

On Alpine Linux

```
curl -fsSL https://github.com/pnpm/pnpm/releases/latest/download/pnpm-linuxstatic-
x64 -o /bin/pnpm; chmod +x /bin/pnpm;
```

Prerequisites

If you don't use the standalone script to install pnpm, then you need to have Node.js (at least v14) to be installed on your system.

Installing a specific version

Prior to running the install script, you may optionally set an env variable `PNPM_VERSION` to install a specific version of pnpm:

```
curl -fsSL https://get.pnpm.io/install.sh | env PNPM_VERSION=<version> sh -
```

> You may use the [pnpm env] command then to install Node.js.

Using Corepack

Since v16.13, Node.js is shipping Corepack for managing package managers. This is an experimental feature, so you need to enable it by running:)

```
corepack enable
```

This will automatically install pnpm on your system. However, it probably won't be the latest version of pnpm. To upgrade it, check what is the latest pnpm version and run:)

```
corepack prepare pnpm@<version> --activate
```

With Node.js v16.17 or newer, you may install the `latest` version of pnpm by just specifying the tag:

```
corepack prepare pnpm@latest --activate
```

Using npm

```
npm install -g pnpm
```

Using Homebrew

If you have the package manager installed, you can install pnpm using the following command:

```
brew install pnpm
```

Using Scoop

If you have Scoop installed, you can install pnpm using the following command:

```
scoop install nodejs-lts pnpm
```

Do you wanna use pnpm on CI servers? See: Continuous Integration

Compatibility

Here is a list of past pnpm versions with respective Node.js version support.

Node.js	pnpm 4	pnpm 5	pnpm 6	pnpm 7
Node.js 10	✓	✓	✗	✗
Node.js 12	✓	✓	✓	✗
Node.js 14	✓	✓	✓	✓
Node.js 16	?	?	✓	✓
Node.js 18	?	?	✓	✓

Troubleshooting

If pnpm is broken and you cannot fix it by reinstalling, you might need to remove it manually from the PATH.

Let's assume you have the following error when running `pnpm install`:

```
C:\src>pnpm install
internal/modules/cjs/loader.js:883
  throw err;
  ^



Error: Cannot find module 'C:\Users\Bence\AppData\Roaming\npm\pnpm-
global\4\node_modules\pnpm\bin\pnpm.js'
←[90m    at Function.Module._resolveFilename
(internal/modules/cjs/loader.js:880:15)←[39m
←[90m    at Function.Module._load (internal/modules/cjs/loader.js:725:27)←[39m
←[90m    at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:72:12)←[39m
←[90m    at internal/main/run_main_module.js:17:47←[39m {
  code: ←[32m'MODULE_NOT_FOUND'←[39m,
  requireStack: []
}
```

First, try to find the location of pnpm by running: `which pnpm`. If you're on Windows, run this command in Git Bash. You'll get the location of the pnpm command, for instance:

```
$ which pnpm
/c/Program Files/nodejs/pnpm
```

Now that you know where the pnpm CLI is, open that directory and remove any pnpm-related files (`pnpm..html)` pnpx..html) `pnpm`, etc). Once done, install pnpm again and it should work as expected.

Using a shorter alias

`pnpm` might be hard to type, so you may use a shorter alias like `pn` instead.

Adding a permanent alias on POSIX systems

Just put the following line to your `.bashrc`, `.zshrc`, or `config.fish`:

```
alias pn=pnpm
```

Adding a permanent alias in Powershell (Windows):

In a Powershell window with admin rights, execute:

```
notepad $profile.AllUsersAllHosts
```

In the `profile.ps1` file that opens, put:

```
set-alias -name pn -value pnpm
```

Save the file and close the window. You may need to close any open Powershell window in order for the alias to take effect.

Uninstalling pnpm

If you need to remove the pnpm CLI from your system and any files it has written to your disk, see [Uninstalling pnpm].

Uninstalling pnpm pnpm env

Go to TOC

Limitations

1. `npm-shrinkwrap.json` and `package-lock.json` are ignored. Unlike pnpm, npm can install the same `name@version` multiple times and with different sets of dependencies. npm's lockfile is designed to reflect the flat `node_modules` layout, however, as pnpm creates an isolated layout by default, it cannot respect npm's lockfile format. See pnpm import if you wish to convert a lockfile to pnpm's format, though.
2. Binstubs (files in `node_modules/.bin`) are always shell files, not symlinks to JS files. The shell files are created to help pluggable CLI apps in finding their plugins in the unusual `node_modules` structure. This is very rarely an issue and if you expect the file to be a JS file, reference the original file directly instead, as described in [#736].

Got an idea for workarounds for these issues? Share them.)

#736

Go to TOC

Logos

Standard logo

SVG:

PNG:

Standard logo with no text

SVG:

PNG:

Standard light logo

SVG:

PNG:

Standard light logo with no text

SVG:

PNG:

Go to TOC

Motivation

Saving disk space and boosting installation speed

)

When using npm, if you have 100 projects using a dependency, you will have 100 copies of that dependency saved on disk. With pnpm, the dependency will be stored in a content-addressable store, so:

1. If you depend on different versions of the dependency, only the files that differ are added to the store. For instance, if it has 100 files, and a new version has a change in only one of those files, `pnpm update` will only add 1 new file to the store, instead of cloning the entire dependency just for the singular change.
2. All the files are saved in a single place on the disk. When packages are installed, their files are hard-linked from that single place, consuming no additional disk space. This allows you to share dependencies of the same version across projects.

As a result, you save a lot of space on your disk proportional to the number of projects and dependencies, and you have a lot faster installations!

Creating a non-flat node_modules directory

)

When installing dependencies with npm or Yarn Classic, all packages are hoisted to the root of the modules directory. As a result, source code has access to dependencies that are not added as dependencies to the project.

By default, pnpm uses symlinks to add only the direct dependencies of the project into the root of the modules directory. If you'd like more details about the unique `node_modules` structure that pnpm creates and why it works fine with the Node.js ecosystem, read:

- Flat node_modules is not the only way
- Symlinked node_modules structure

If your tooling doesn't work well with symlinks, you may still use pnpm and set the node-linker setting to `hoisted`. This will instruct pnpm to create a node_modules directory that is similar to those created by npm and Yarn Classic.

Go to TOC

.npmrc

pnpm gets its configuration from the command line, environment variables, and `.npmrc` files.

The `pnpm config` command can be used to update and edit the contents of the user and global `.npmrc` files.

The four relevant files are:

- per-project configuration file (`/path/to/my/project/.npmrc`)
- per-workspace configuration file (the directory that contains the `pnpm-workspace.yaml` file)
- per-user configuration file (`~/.npmrc`)
- global configuration file (`/etc/npmrc`)

All `.npmrc` files are an [INI-formatted] list of `key = value` parameters.

INI-formatted

Dependency Hoisting Settings

hoist

- Default: **true**
- Type: **boolean**

When `true` , all dependencies are hoisted to `node_modules/.pnpm` . This makes unlisted dependencies accessible to all packages inside `node_modules` .

hoist-pattern

- Default: **['*']**
- Type: **string[]**

Tells pnpm which packages should be hoisted to `node_modules/.pnpm` . By default, all packages are hoisted - however, if you know that only some flawed packages have phantom dependencies, you can use this option to exclusively hoist the phantom dependencies (recommended).

For instance:

```
hoist-pattern[]=*eslint*
hoist-pattern[]=*babel*
```

Since v7.12.0, you may also exclude patterns from hoisting using `!` .

For instance:

```
hoist-pattern[]=*types*
hoist-pattern[]=!@types/react
```

public-hoist-pattern

- Default: **['*eslint*', '*prettier*']**
- Type: **string[]**

Unlike `hoist-pattern`, which hoists dependencies to a hidden modules directory inside the virtual store, `public-hoist-pattern` hoists dependencies matching the pattern to the root modules directory. Hoisting to the root modules directory means that application code will have access to phantom dependencies, even if they modify the resolution strategy improperly.

This setting is useful when dealing with some flawed pluggable tools that don't resolve dependencies properly.

For instance:

```
public-hoist-pattern[]=*plugin*
```

Note: Setting `shamefully-hoist` to `true` is the same as setting

`public-hoist-pattern` to `*`.

Since v7.12.0, you may also exclude patterns from hoisting using `!`.

For instance:

```
public-hoist-pattern[]=*types*
public-hoist-pattern[]=!@types/react
```

shamefully-hoist

- Default: **false**
- Type: **Boolean**

By default, pnpm creates a semistrict `node_modules`, meaning dependencies have access to undeclared dependencies but modules outside of `node_modules` do not. With this layout, most of the packages in the ecosystem work with no issues. However, if some tooling only works when the hoisted dependencies are in the root of `node_modules`, you can set this to `true` to hoist them for you.

Node-Modules Settings

store-dir

- Default:
  - If the **$XDG_DATA_HOME** env variable is set, then **$XDG_DATA_HOME/pnpm/store**
  - On Windows: **~/AppData/Local/pnpm/store**
  - On macOS: **~/Library/pnpm/store**
  - On Linux: **~/.local/share/pnpm/store**
- Type: **path**

The location where all the packages are saved on the disk.

The store should be always on the same disk on which installation is happening, so there will be one store per disk. If there is a home directory on the current disk, then the store is created inside it. If there is no home on the disk, then the store is created at the root of the filesystem. For example, if installation is happening on a filesystem mounted at `/mnt`, then the store will be created at `/mnt/.pnpm-store`. The same goes for Windows systems.

It is possible to set a store from a different disk but in that case pnpm will copy packages from the store instead of hard-linking them, as hard links are only possible on the same filesystem.

modules-dir

- Default: **node_modules**
- Type: **path**

The directory in which dependencies will be installed (instead of `node_modules`).

node-linker

- Default: **isolated**
- Type: **isolated**, **hoisted**, **pnp**

Defines what linker should be used for installing Node packages.

- **isolated** - dependencies are symlinked from a virtual store at `node_modules/.pnpm`.
- **hoisted** - a flat `node_modules` without symlinks is created. Same as the `node_modules` created by npm or Yarn Classic. One of Yarn's libraries is used for hoisting, when this setting is used. Legitimate reasons to use this setting:
  1. Your tooling doesn't work well with symlinks. A React Native project will most probably only work if you use a hoisted `node_modules`.
  2. Your project is deployed to a serverless hosting provider. Some serverless providers (for instance, AWS Lambda) don't support symlinks. An alternative solution for this problem is to bundle your application before deployment.
  3. If you want to publish your package with [`bundledDependencies`].
  4. If you are running Node.js with the [--preserve-symlinks] flag.
- **pnp** - no `node_modules`. Plug'n'Play is an innovative strategy for Node that is [used by Yarn Berry][pnp]. It is recommended to also set `symlink` setting to `false` when using `pnp` as your linker.

pnp --preserve-symlinks `bundledDependencies`

symlink

- Default: **true**
- Type: **Boolean**

When `symlink` is set to `false`, pnpm creates a virtual store directory without any symlinks. It is a useful setting together with `node-linker=pnp`.

enable-modules-dir

- Default: **true**
- Type: **Boolean**

When `false`, pnpm will not write any files to the modules directory (`node_modules`). This is useful for when the modules directory is mounted with filesystem in userspace (FUSE). There is an experimental CLI that allows you to mount a modules directory with FUSE: [@pnpm/mount-modules].

@pnpm/mount-modules

virtual-store-dir

- Default: **node_modules/.pnpm**
- Types: **path**

The directory with links to the store. All direct and indirect dependencies of the project are linked into this directory.

This is a useful setting that can solve issues with long paths on Windows. If you have some dependencies with very long paths, you can select a virtual store in the root of your drive (for instance `C:\my-project-store`).

Or you can set the virtual store to `.pnpm` and add it to `.gitignore`. This will make stacktraces cleaner as paths to dependencies will be one directory higher.

NOTE: the virtual store cannot be shared between several projects. Every project should have its own virtual store (except for in workspaces where the root is shared).

package-import-method

- Default: **auto**
- Type: **auto**, **hardlink**, **copy**, **clone**, **clone-or-copy**

Controls the way packages are imported from the store (if you want to disable symlinks inside `node_modules`, then you need to change the [node-linker] setting, not this one).

- **auto** - try to clone packages from the store. If cloning is not supported then hardlink packages from the store. If neither cloning nor linking is possible, fall back to copying
- **hardlink** - hard link packages from the store
- **clone-or-copy** - try to clone packages from the store. If cloning is not supported then fall back to copying
- **copy** - copy packages from the store
- **clone** - clone (AKA copy-on-write or reference link) packages from the store

Cloning is the best way to write packages to node_modules. It is the fastest way and safest way. When cloning is used, you may edit files in your node_modules and they will not be modified in the central content-addressable store.

Unfortunately, not all file systems support cloning. We recommend using a copy-on-write (CoW) file system (for instance, Btrfs instead of Ext4 on Linux) for the best experience with pnpm.

> Even though macOS supports cloning, there is currently [a bug in Node.js] that prevents us from using it in pnpm. If you have ideas how to fix it, [help us].

a bug in Node.js help us [node-linker]: #node-linker

modules-cache-max-age

- Default: **10080** (7 days in minutes)
- Type: **number**

The time in minutes after which orphan packages from the modules directory should be removed. pnpm keeps a cache of packages in the modules directory. This boosts installation speed when switching branches or downgrading dependencies.

Lockfile Settings

lockfile

- Default: **true**
- Type: **Boolean**

When set to `false`, pnpm won't read or generate a `pnpm-lock.yaml` file.

prefer-frozen-lockfile

- Default: **true**
- Type: **Boolean**

When set to `true` and the available `pnpm-lock.yaml` satisfies the `package.json` dependencies directive, a headless installation is performed. A headless installation skips all dependency resolution as it does not need to modify the lockfile.

lockfile-include-tarball-url

Added in: v7.6.0

- Default: **false**
- Type: **Boolean**

Add the full URL to the package's tarball to every entry in `pnpm-lock.yaml`.

Registry & Authentication Settings

registry

- Default: **https://registry.npmjs.org/**
- Type: **url**

The base URL of the npm package registry (trailing slash included).

<scope>:registry

The npm registry that should be used for packages of the specified scope. For example, setting `@babel:registry=https://example.com/packages/npm/` will enforce that when you use `pnpm add @babel/core`, or any `@babel` scoped package, the package will be fetched from `https://example.com/packages/npm` instead of the default registry.

<URL>:_authToken

Define the authentication bearer token to use when accessing the specified registry. For example:

```
//registry.npmjs.org/:_authToken=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

You may also use an environment variable. For example:

```
//registry.npmjs.org/:_authToken=${NPM_TOKEN}
```

Or you may just use an environment variable directly, without changing `.npmrc` at all:

```
npm_config_//registry.npmjs.org/:_authToken=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

<URL>:tokenHelper

A token helper is an executable which outputs an auth token. This can be used in situations where the authToken is not a constant value but is something that refreshes regularly, where a script or other tool can use an existing refresh token to obtain a new access token.

The configuration for the path to the helper must be an absolute path, with no arguments. In order to be secure, it is only permitted to set this value in the user `.npmrc`. Otherwise a project could place a value in a project's local `.npmrc` and run arbitrary executables.

Setting a token helper for the default registry:

```
tokenHelper=/home/ivan/token-generator
```

Setting a token helper for the specified registry:

```
//registry.corp.com:tokenHelper=/home/ivan/token-generator
```

Request Settings

ca

- Default: **The npm CA certificate**
- Type: **String, Array or null**

The Certificate Authority signing certificate that is trusted for SSL connections to the registry. Values should be in PEM format (AKA Base-64 encoded X.509 (.CER)). For example:

```
ca=--BEGIN CERTIFICATE-----\nXXXX\nXXXX\n-----END CERTIFICATE-----
```

Set to null to only allow known registrars, or to a specific CA cert to trust only that specific signing authority.

Multiple CAs can be trusted by specifying an array of certificates:

```
ca[]=...
ca[]=...
```

See also the `strict-ssl` config.

cafile

- Default: **null**
- Type: **path**

A path to a file containing one or multiple Certificate Authority signing certificates. Similar to the `ca` setting, but allows for multiple CAs, as well as for the CA information to be stored in a file instead of being specified via

CLI.

cert

- Default: **null**
- Type: **String**

A client certificate to pass when accessing the registry. Values should be in PEM format (AKA Base-64 encoded X.509 (.CER)). For example:

```
cert=--BEGIN CERTIFICATE-----\nXXXX\nXXXX\n-----END CERTIFICATE-----
```

It is not the path to a certificate file (and there is no `certfile` option).

key

- Default: **null**
- Type: **String**

A client key to pass when accessing the registry. Values should be in PEM format (AKA Base-64 encoded X.509 (.CER)). For example:

```
key=--BEGIN PRIVATE KEY-----\nXXXX\nXXXX\n-----END PRIVATE KEY-----
```

It is not the path to a key file (and there is no `keyfile` option).

This setting contains sensitive information. Don't write it to a local `.npmrc` file committed to the repository.

git-shallow-hosts

- Default: **['github.com', 'gist.github.com', 'gitlab.com', 'bitbucket.com', 'bitbucket.org']**
- Type: **string[]**

When fetching dependencies that are Git repositories, if the host is listed in this setting, pnpm will use shallow cloning to fetch only the needed commit, not all the history.

https-proxy

- Default: **null**
- Type: **url**

A proxy to use for outgoing HTTPS requests. If the `HTTPS_PROXY`, `https_proxy`, `HTTP_PROXY` or `http_proxy` environment variables are set, their values will be used instead.

local-address

- Default: **undefined**
- Type: **IP Address**

The IP address of the local interface to use when making connections to the npm registry.

proxy

- Default: **null**
- Type: **url**

A proxy to use for outgoing http requests. If the HTTP_PROXY or http_proxy environment variables are set, proxy settings will be honored by the underlying request library.

maxsockets

- Default: **network-concurrency x 3**
- Type: **Number**

The maximum number of connections to use per origin (protocol/host/port combination).

noproxy

- Default: **null**
- Type: **String**

A comma-separated string of domain extensions that a proxy should not be used for.

strict-ssl

- Default: **true**
- Type: **Boolean**

Whether or not to do SSL key validation when making requests to the registry via HTTPS.

See also the `ca` option.

network-concurrency

- Default: **16**
- Type: **Number**

Controls the maximum number of HTTP(S) requests to process simultaneously.

fetch-retries

- Default: **2**
- Type: **Number**

How many times to retry if pnpm fails to fetch from the registry.

fetch-retry-factor

- Default: **10**
- Type: **Number**

The exponential factor for retry backoff.

fetch-retry-mintimeout

- Default: **10000 (10 seconds)**
- Type: **Number**

The minimum (base) timeout for retrying requests.

fetch-retry-maxtimeout

- Default: **60000 (1 minute)**
- Type: **Number**

The maximum fallback timeout to ensure the retry factor does not make requests too long.

fetch-timeout

- Default: **60000 (1 minute)**
- Type: **Number**

The maximum amount of time to wait for HTTP requests to complete.

Peer Dependency Settings

auto-install-peers

- Default: **false**
- Type: **Boolean**

When `true`, any missing non-optional peer dependencies are automatically installed.

strict-peer-dependencies

- Default: **false** (was **true** from v7.0.0 until v7.13.5)
- Type: **Boolean**

If this is enabled, commands will fail if there is a missing or invalid peer dependency in the tree.

CLI Settings

[no-]color

- Default: **auto**
- Type: **auto**, **always**, **never**

Controls colors in the output.

- **auto** - output uses colors when the standard output is a terminal or TTY.
- **always** - ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the pnpm command to force it to use color codes. The default setting is almost always what you'll want.
- **never** - turns off colors. This is the setting used by `--no-color`.

loglevel

- Default: **info**

- Type: **debug**, **info**, **warn**, **error**

Any logs at or higher than the given level will be shown. You can instead pass `--silent` to turn off all output logs.

use-beta-cli

- Default: **false**
- Type: **Boolean**

Experimental option that enables beta features of the CLI. This means that you may get some changes to the CLI functionality that are breaking changes, or potentially bugs.

recursive-install

- Default: **true**
- Type: **Boolean**

If this is enabled, the primary behaviour of `pnpm install` becomes that of `pnpm install -r`, meaning the install is performed on all workspace or subdirectory packages.

Else, `pnpm install` will exclusively build the package in the current directory.

engine-strict

- Default: **false**
- Type: **Boolean**

If this is enabled, pnpm will not install any package that claims to not be compatible with the current Node version.

Regardless of this configuration, installation will always fail if a project (not a dependency) specifies an incompatible version in its `engines` field.

npm-path

- Type: **path**

The location of the npm binary that pnpm uses for some actions, like publishing.

Build Settings

ignore-scripts

- Default: **false**
- Type: **Boolean**

Do not execute any scripts defined in the project `package.json` and its dependencies.

> This flag does not prevent the execution of .pnpmfile.cjs

ignore-dep-scripts

Added in: v7.9.0

- Default: **false**
- Type: **Boolean**

Do not execute any scripts of the installed packages. Scripts of the projects are executed.

child-concurrency

- Default: **5**
- Type: **Number**

The maximum number of child processes to allocate simultaneously to build node_modules.

side-effects-cache

- Default: **true**
- Type: **Boolean**

Use and cache the results of (pre/post)install hooks.

side-effects-cache-readonly

- Default: **false**
- Type: **Boolean**

Only use the side effects cache if present, do not create it for new packages.

unsafe-perm

- Default: **false** IF running as root, ELSE **true**
- Type: **Boolean**

Set to true to enable UID/GID switching when running package scripts. If set explicitly to false, then installing as a non-root user will fail.

Node.js Settings

use-node-version

- Default: **undefined**
- Type: **semver**

Specifies which exact Node.js version should be used for the project's runtime. pnpm will automatically install the specified version of Node.js and use it for running `pnpm run` commands or the `pnpm node` command.

This may be used instead of `.nvmrc` and `nvm`. Instead of the following `.nvmrc` file:

```
16.16.0
```

Use this `.npmrc` file:

```
use-node-version=16.16.0
```

node-version

- Default: the value returned by **node -v**, without the v prefix
- Type: **semver**

The Node.js version to use when checking a package's `engines` setting.

If you want to prevent contributors of your project from adding new incompatible dependencies, use `node-version` and `engine-strict` in a `.npmrc` file at the root of the project:

```
node-version=12.22.0
engine-strict=true
```

This way, even if someone is using Node.js v16, they will not be able to install a new dependency that doesn't support Node.js v12.22.0.

node-mirror:<releaseDir>

- Default: **`https://nodejs.org/download/<releaseDir>/`**
- Type: **URL**

Sets the base URL for downloading Node.js. The `<releaseDir>` portion of this setting can be any directory from https://nodejs.org/download: `release`, `rc`, `nightly`, `v8-canary`, etc.

Here is how pnpm may be configured to download Node.js from Node.js mirror in China:

```
node-mirror:release=https://npmmirror.com/mirrors/node/
node-mirror:rc=https://npmmirror.com/mirrors/node-rc/
node-mirror:nightly=https://npmmirror.com/mirrors/node-nightly/
```

Workspace Settings

link-workspace-packages

- Default: **true**
- Type: **true**, **false**, **deep**

If this is enabled, locally available packages are linked to `node_modules` instead of being downloaded from the registry. This is very convenient in a monorepo. If you need local packages to also be linked to subdependencies, you can use the `deep` setting.

Else, packages are downloaded and installed from the registry. However, workspace packages can still be linked by using the `workspace:` range protocol.

prefer-workspace-packages

- Default: **false**
- Type: **Boolean**

If this is enabled, local packages from the workspace are preferred over packages from the registry, even if there is a newer version of the package in the registry.

This setting is only useful if the workspace doesn't use `save-workspace-protocol`.

shared-workspace-lockfile

- Default: **true**
- Type: **Boolean**

If this is enabled, pnpm creates a single `pnpm-lock.yaml` file in the root of the workspace. This also means that all dependencies of workspace packages will be in a single `node_modules` (and get symlinked to their package `node_modules` folder for Node's module resolution).

Advantages of this option:

- every dependency is a singleton
- faster installations in a monorepo
- fewer changes in code reviews as they are all in one file

> Even though all the dependencies will be hard linked into the root `node_modules`, packages will have access only to those dependencies that are declared in their `package.json`, so pnpm's strictness is preserved. This is a result of the aforementioned symbolic linking.

save-workspace-protocol

- Default: **true**
- Type: **true**, **false**, **rolling**

This setting controls how dependencies that are linked from the workspace are added to `package.json`.

If `foo@1.0.0` is in the workspace and you run `pnpm add foo` in another project of the workspace, below is how `foo` will be added to the dependencies field. The `save-prefix` setting also influences how the spec is created.

save-workspace-protocol	save-prefix	spec
false	`''`	`1.0.0`
false	`'~'`	`~1.0.0`
false	`'^'`	`^1.0.0`
true	`''`	`workspace:1.0.0`
true	`'~'`	`workspace:~1.0.0`
true	`'^'`	`workspace:^1.0.0`
rolling	`''`	`workspace:*`
rolling	`'~'`	`workspace:~`
rolling	`'^'`	`workspace:^`

include-workspace-root

Added in: v7.4.0

- Default: **false**
- Type: **Boolean**

When executing commands recursively in a workspace, execute them on the root workspace project as well.

Other Settings

use-running-store-server

- Default: **false**
- Type: **Boolean**

Only allows installation with a store server. If no store server is running, installation will fail.

save-prefix

- Default: **'^'**
- Type: **String**

Configure how versions of packages installed to a `package.json` file get prefixed.

For example, if a package has version `1.2.3`, by default its version is set to `^1.2.3` which allows minor upgrades for that package, but after `pnpm config set save-prefix='~'` it would be set to `~1.2.3` which only allows patch upgrades.

This setting is ignored when the added package has a range specified. For instance, `pnpm add foo@2` will set the version of `foo` in `package.json` to `2`, regardless of the value of `save-prefix`.

tag

- Default: **latest**
- Type: **String**

If you `pnpm add` a package and you don't provide a specific version, then it will install the package at the version registered under the tag from this setting.

This also sets the tag that is added to the `package@version` specified by the `pnpm tag` command if no explicit tag is given.

global-dir

- Default:
  - If the **$XDG_DATA_HOME** env variable is set, then **$XDG_DATA_HOME/pnpm/global**
  - On Windows: **~/AppData/Local/pnpm/global**
  - On macOS: **~/Library/pnpm/global**
  - On Linux: **~/.local/share/pnpm/global**

- Type: **path**

Specify a custom directory to store global packages.

global-bin-dir

- Default:
  - If the **$XDG_DATA_HOME** env variable is set, then **$XDG_DATA_HOME/pnpm**
  - On Windows: **~/AppData/Local/pnpm**
  - On macOS: **~/Library/pnpm**
  - On Linux: **~/.local/share/pnpm**
- Type: **path**

Allows to set the target directory for the bin files of globally installed packages.

state-dir

- Default:
  - If the **$XDG_STATE_HOME** env variable is set, then **$XDG_STATE_HOME/pnpm**
  - On Windows: **~/AppData/Local/pnpm-state**
  - On macOS: **~/.pnpm-state**
  - On Linux: **~/.local/state/pnpm**
- Type: **path**

The directory where pnpm creates the `pnpm-state.json` file that is currently used only by the update checker.

cache-dir

- Default:
  - If the **$XDG_CACHE_HOME** env variable is set, then **$XDG_CACHE_HOME/pnpm**
  - On Windows: **~/AppData/Local/pnpm-cache**
  - On macOS: **~/Library/Caches/pnpm**
  - On Linux: **~/.cache/pnpm**
- Type: **path**

The location of the package metadata cache.

use-stderr

- Default: **false**
- Type: **Boolean**

When true, all the output is written to stderr.

update-notifier

- Default: **true**
- Type: **Boolean**

Set to `false` to suppress the update notification when using an older version of pnpm than the latest.

prefer-symlinked-executables

Added in: v7.6.0

- Default: **true**, when **node-linker** is set to **hoisted** and the system is POSIX
- Type: **Boolean**

Create symlinks to executables in `node_modules/.bin` instead of command shims. This setting is ignored on Windows, where only command shims work.

verify-store-integrity

Added in: v7.7.0

- Default: **true**
- Type: **Boolean**

By default, if a file in the store has been modified, the content of this file is checked before linking it to a project's `node_modules`. If `verify-store-integrity` is set to `false`, files in the content-addressable store will not be checked during installation.

ignore-compatibility-db

Added in: v7.9.0

- Default: **false**
- Type: **Boolean**

During installation the dependencies of some packages are automatically patched. If you want to disable this, set this config to `false`.

The patches are applied from Yarn's [`@yarnpkg/extensions`] package.

resolution-mode

Added in: v7.10.0

- Default: **highest**
- Type: **highest**, **time-based**

When `resolution-mode` is set to `time-based`, dependencies will be resolved the following way:

1. Direct dependencies will be resolved to their lowest versions. So if there is `foo@^1.1.0` in the dependencies, then `1.1.0` will be installed.
2. Subdependencies will be resolved from versions that were published before the last direct dependency was published.

With this resolution mode installations with warm cache are faster. It also reduces the chance of subdependency hijacking as subdependencies will be updated only if direct dependencies are updated.

This resolution mode works only with npm's [full metadata]. So it is slower in some scenarios. However, if you use [Verdaccio] v5.15.1 or newer, you may set the `registry-supports-time-field` setting to `true`, and it will be really fast.

registry-supports-time-field

Added in: v7.10.0

- Default: **false**
- Type: **Boolean**

Set this to `true` if the registry that you are using returns the time field in the abbreviated metadata. As of now, only [Verdaccio] from v5.15.1 supports this.

`@yarnpkg/extensions` full metadata Verdaccio

Go to TOC

Only allow pnpm

When you use pnpm on a project, you don't want others to accidentally run `npm install` or `yarn`. To prevent devs from using other package managers, you can add the following `preinstall` script to your `package.json`:

```
{
    scripts: {
        preinstall: npx only-allow pnpm
    }
}
```

Now, whenever someone runs `npm install` or `yarn`, they'll get an error instead and installation will not proceed.

If you use npm v7, use `npx -y` instead.

Go to TOC

package.json

The manifest file of a package. It contains all the package's metadata, including dependencies, title, author, et cetera. This is a standard preserved across all major Node.JS package managers, including pnpm.

engines

You can specify the version of Node and pnpm that your software works on:

```
{
    engines: {
        node: >=10,
        pnpm: >=3
    }
}
```

During local development, pnpm will always fail with an error message if its version does not match the one specified in the `engines` field.

Unless the user has set the `engine-strict` config flag (see [.npmrc]), this field is advisory only and will only produce warnings when your package is installed as a dependency.

.npmrc

dependenciesMeta

Additional meta information used for dependencies declared inside `dependencies`, `optionalDependencies`, and `devDependencies`.

dependenciesMeta.*.injected

If this is set to true for a local dependency, the package will be hard linked to the modules directory, not symlinked.

For instance, the following `package.json` in a workspace will create a symlink to `button` in the `node_modules` directory of `card`:

```
{
  name: card,
  dependencies: {
    button: workspace:1.0.0
  }
}
```

But what if `button` has `react` in its peer dependencies? If all projects in the monorepo use the same version of `react`, then no problem. But what if `button` is required by `card` that uses `react@16` and `form` with `react@17`? Without using `inject`, you'd have to choose a single version of `react` and install it as dev dependency of `button`. But using the `injected` field you can inject `button` to a package, and `button` will be installed with the `react` version of that package.

So this will be the `package.json` of `card`:

```
{
  name: card,
  dependencies: {
    button: workspace:1.0.0,
    react: 16
  },
  dependenciesMeta: {
    button: {
      injected: true
    }
  }
}
```

`button` will be hard linked into the dependencies of `card`, and `react@16` will be symlinked to the dependencies of `card/node_modules/button`.

And this will be the `package.json` of `form`:

```
{
  name: form,
  dependencies: {
    button: workspace:1.0.0,
    react: 17
  },
  dependenciesMeta: {
    button: {
      injected: true
    }
  }
}
```

`button` will be hard linked into the dependencies of `form`, and `react@17` will be symlinked to the dependencies of `form/node_modules/button`.

peerDependenciesMeta

This field lists some extra information related to the dependencies listed in

the `peerDependencies` field.

peerDependenciesMeta.*.optional

If this is set to true, the selected peer dependency will be marked as optional by the package manager. Therefore, the consumer omitting it will no longer be reported as an error.

For example:

```
{
    peerDependencies: {
        foo: 1
    },
    peerDependenciesMeta: {
        foo: {
            optional: true
        },
        bar: {
            optional: true
        }
    }
}
```

Note that even though `bar` was not specified in `peerDependencies`, it is

marked as optional. pnpm will therefore assume that any version of bar is fine. However, `foo` is optional, but only to the required version specification.

publishConfig

It is possible to override some fields in the manifest before the package is packed. The following fields may be overridden:

- `bin`)
- `main`)
- `exports`)
- `types` or `typings`)
- `module`)
- `browser`)
- `esnext`)
- `es2015`)
- `unpkg`)
- [`.html)
- `typesVersions`)
- cpu
- os

To override a field, add the publish version of the field to `publishConfig`.

For instance, the following `package.json`:

```
{
    name: foo,
    version: 1.0.0,
    main: src/index.ts,
    publishConfig: {
        main: lib/index.js,
        typings: lib/index.d.ts
    }
}
```

Will be published as:

```
{
    name: foo,
    version: 1.0.0,
    main: lib/index.js,
    typings: lib/index.d.ts
}
```

publishConfig.executableFiles

By default, for portability reasons, no files except those listed in the bin field will be marked as executable in the resulting package archive. The `executableFiles` field lets you declare additional fields that must have the executable flag (+x) set even if they aren't directly accessible through the bin field.

```
{
  publishConfig: {
    executableFiles: [
      ./dist/shim.js
    ]
  }
}
```

publishConfig.directory

You also can use the field `publishConfig.directory` to customize the published subdirectory relative to the current `package.json`.

It is expected to have a modified version of the current package in the specified directory (usually using third party build tools).

In this example the `dist` folder must contain a `package.json`

```
{
  name: foo,
  version: 1.0.0,
  publishConfig: {
    directory: dist
  }
}
```

publishConfig.linkDirectory

Added in: v7.8.0

When set to `true`, the project will be symlinked from the `publishConfig.directory` location during local development.

For example:

```
{
  name: foo,
  version: 1.0.0,
  publishConfig: {
    directory: dist
    linkDirectory: true
  }
}
```

pnpm.overrides

This field allows you to instruct pnpm to override any dependency in the dependency graph. This is useful to enforce all your packages to use a single version of a dependency, backport a fix, or replace a dependency with a fork.

Note that the overrides field can only be set at the root of the project.

An example of the `pnpm.overrides` field:

```
{
  pnpm: {
    overrides: {
      foo: ^1.0.0,
      quux: npm:@myorg/quux@^1.0.0,
      bar@^2.1.0: 3.0.0,
      qar@1>zoo: 2
    }
  }
}
```

You may specify the package the overriden dependency belongs to by separating the package selector from the dependency selector with a >, for example `qar@1>zoo` will only override the `zoo` dependency of `qar@1`, not for any other dependencies.

An override may be defined as a reference to a direct dependency's spec. This is achieved by prefixing the name of the dependency with a `$`:

```
{
  dependencies: {
    foo: ^1.0.0
  },
  overrides: {
    foo: $foo
  }
}
```

The referenced package does not need to match the overridden one:

```
{
  dependencies: {
    foo: ^1.0.0
  },
  overrides: {
    bar: $foo
  }
}
```

pnpm.packageExtensions

The `packageExtensions` fields offer a way to extend the existing package definitions with additional information. For example, if `react-redux` should have `react-dom` in its `peerDependencies` but it has not, it is possible to patch `react-redux` using `packageExtensions`:

```
{
  pnpm: {
    packageExtensions: {
      react-redux: {
        peerDependencies: {
          react-dom: *
        }
      }
    }
  }
}
```

The keys in `packageExtensions` are package names or package names and semver ranges, so it is possible to patch only some versions of a package:

```
{
  pnpm: {
    packageExtensions: {
      react-redux@1: {
        peerDependencies: {
          react-dom: *
        }
      }
    }
  }
}
```

The following fields may be extended using `packageExtensions`: `dependencies`, `optionalDependencies`, `peerDependencies`, and `peerDependenciesMeta`.

A bigger example:

```
{
  pnpm: {
    packageExtensions: {
      express@1: {
        optionalDependencies: {
          typescript: 2
        }
      },
      fork-ts-checker-webpack-plugin: {
        dependencies: {
          @babel/core: 1
        },
        peerDependencies: {
          eslint: >= 6
        },
        peerDependenciesMeta: {
          eslint: {
            optional: true
          }
```

```
            }
          }
        }
      }
    }
```

Together with Yarn, we maintain a database of `packageExtensions` to patch broken packages in the ecosystem. If you use `packageExtensions` , consider sending a PR upstream and contributing your extension to the [`@yarnpkg/extensions`] database.

`@yarnpkg/extensions`

pnpm.peerDependencyRules

pnpm.peerDependencyRules.ignoreMissing

pnpm will not print warnings about missing peer dependencies from this list.

For instance, with the following configuration, pnpm will not print warnings if a dependency needs `react` but `react` is not installed:

```
{
  pnpm: {
    peerDependencyRules: {
      ignoreMissing: [react]
    }
  }
}
```

Package name patterns may also be used:

```
{
  pnpm: {
    peerDependencyRules: {
      ignoreMissing: [@babel/*, @eslint/*]
    }
  }
}
```

pnpm.peerDependencyRules.allowedVersions

Unmet peer dependency warnings will not be printed for peer dependencies of the specified range.

For instance, if you have some dependencies that need `react@16` but you know that they work fine with `react@17` , then you may use the following configuration:

```
{
  pnpm: {
    peerDependencyRules: {
      allowedVersions: {
        react: 17
      }
    }
  }
}
```

This will tell pnpm that any dependency that has react in its peer dependencies should allow `react` v17 to be installed.

pnpm.peerDependencyRules.allowAny

Added in: v7.3.0

`allowAny` is an array of package name patterns, any peer dependency matching the pattern will be resolved from any version, regardless of the range specified in `peerDependencies`. For instance:

```
{
  pnpm: {
    peerDependencyRules: {
      allowAny: [@babel/*, eslint]
    }
  }
}
```

The above setting will mute any warnings about peer dependency version mismatches related to `@babel/` packages or `eslint`.

pnpm.neverBuiltDependencies

This field allows to ignore the builds of specific dependencies. The preinstall, install, and postinstall scripts of the listed packages will not be executed during installation.

An example of the `pnpm.neverBuiltDependencies` field:

```
{
  pnpm: {
    neverBuiltDependencies: [fsevents, level]
  }
}
```

pnpm.onlyBuiltDependencies

A list of package names that are allowed to be executed during installation. If this field exists, only the listed packages will be able to run install scripts.

Example:

```
{
  pnpm: {
    onlyBuiltDependencies: [fsevents]
  }
}
```

pnpm.allowedDeprecatedVersions

Added in: v7.2.0

This setting allows muting deprecation warnings of specific packages.

Example:

```
{
  pnpm: {
    allowedDeprecatedVersions: {
      express: 1,
      request: *
    }
  }
}
```

With the above configuration pnpm will not print deprecation warnings about any version of `request` and about v1 of `express`.

pnpm.patchedDependencies

Added in: v7.4.0

This field is added/updated automatically when you run [pnpm patch-commit]. It is a dictionary where the key should be the package name and exact version. The value should be a relative path to a patch file.

Example:

```
{
  pnpm: {
    patchedDependencies: {
      express@4.18.1: patches/express@4.18.1.patch
    }
  }
}
```

pnpm.allowNonAppliedPatches

Added in: v7.12.0

When `true`, installation won't fail if some of the patches from the `patchedDependencies` field were not applied.

```
{
  pnpm: {
    patchedDependencies: {
      express@4.18.1: patches/express@4.18.1.patch
    }
    allowNonAppliedPatches: true
}
```

pnpm.updateConfig

pnpm.updateConfig.ignoreDependencies

Added in: v7.13.0

Sometimes you can't update a dependency. For instance, the latest version of the dependency started to use ESM but your project is not yet in ESM. Annoyingly, such a package will be always printed out by the `pnpm outdated` command and updated, when running `pnpm update --latest`. However, you may list packages that you don't want to upgrade in the `ignoreDependencies` field:

```
{
  pnpm: {
    updateConfig: {
      ignoreDependencies: [load-json-file]
    }
  }
}
```

Patterns are also supported, so you may ignore any packages from a scope: `@babel/*`.

pnpm.auditConfig

pnpm.auditConfig.ignoreCves

Added in: v7.15.0

A list of CVE IDs that will be ignored by the [`pnpm audit`] command.

```
{
  pnpm: {
    auditConfig: {
      ignoreCves: [
        CVE-2022-36313
      ]
    }
  }
}
```

`pnpm audit`

resolutions

Same as [`pnpm.overrides`]. We read it for easier migration from Yarn.

pnpm patch-commit [`pnpm.overrides`]: #pnpmoverrides

Go to TOC

pnpm CLI

Differences vs npm

Unlike npm, pnpm validates all options. For example, `pnpm install --target_arch x64` will fail as `--target_arch` is not a valid option for `pnpm install`.

However, some dependencies may use the `npm_config_` environment variable, which is populated from the CLI options. In this case, you have the following options:

1. explicitly set the env variable: `npm_config_target_arch=x64 pnpm install`
2. force the unknown option with `--config.`: `pnpm install --config.target_arch=x64`

Options

-C <path>, --dir <path>

Run as if pnpm was started in `<path>` instead of the current working directory.

-w, --workspace-root

Run as if pnpm was started in the root of the workspace) instead of the current working directory.

Commands

For more information, see the documentation for individual CLI commands. Here is

a list of handy npm equivalents to get you started:

npm command	pnpm equivalent
`npm install`	[`pnpm install`]
`npm i <pkg>`	[`pnpm add <pkg>`]
`npm run <.html)	[`pnpm <.html)

When an unknown command is used, pnpm will search for a script with the given name, so `pnpm run lint` is the same as `pnpm lint`. If there is no script with the specified name, then pnpm will execute the command as a shell script, so you can do things like `pnpm eslint` (see [pnpm exec]).

`pnpm install` `pnpm add <pkg>` [`pnpm <.html) pnpm exec

Go to TOC

pnpm vs npm

npm's flat tree

npm maintains a [flattened dependency tree] as of version 3. This leads to less disk space bloat, with a messy `node_modules` directory as a side effect.

On the other hand, pnpm manages `node_modules` by using hard linking and symbolic linking to a global on-disk content-addressable store. This nets you the benefits of far less disk space usage, while also keeping your `node_modules` clean. There is documentation on the [store layout] if you wish to learn more.

The good thing about pnpm's proper `node_modules` structure is that it [helps to avoid silly bugs] by making it impossible to use modules that are not specified in the project's `package.json`.

flattened dependency tree [store layout]: symlinked-node-modules-structure helps to avoid silly bugs

Installation

pnpm does not allow installation of packages without saving them to `package.json`. If no parameters are passed to `pnpm add`, packages are saved as regular dependencies. Like with npm, `--save-dev` and `--save-optional` can be used to install packages as dev or optional dependencies.

As a consequence of this limitation, projects won't have any extraneous packages when they use pnpm unless they remove a dependency and leave it orphaned. That's why pnpm's implementation of the prune command does not allow you to specify packages to prune - it ALWAYS removes all extraneous and orphaned packages.

Directory dependencies

Directory dependencies start with the `file:` prefix and point to a directory in the filesystem. Like npm, pnpm symlinks those dependencies. Unlike npm, pnpm does not perform installation for the file dependencies.

This means that if you have a package called `foo` (`<root>/foo`) that has `bar@file:../bar` as a dependency, pnpm won't perform installation for `<root>/bar` when you run `pnpm install` on `foo`.

If you need to run installations in several packages at the same time, for instance in the case of a monorepo, you should look at the documentation for `pnpm -r`.

Go to TOC

pnpm-workspace.yaml

`pnpm-workspace.yaml` defines the root of the workspace and enables you to include / exclude directories from the workspace. By default, all packages of all subdirectories are included.

For example:

pnpm-workspace.yaml

```yaml
packages:
  # all packages in direct subdirs of packages/
  - 'packages/*'
  # all packages in subdirs of components/
  - 'components/**'
  # exclude packages that are inside test directories
  - '!**/test/**'
```

The root package is always included, even when custom location wildcards are used.

Go to TOC

.pnpmfile.cjs

pnpm lets you hook directly into the installation process via special functions (hooks). Hooks can be declared in a file called `.pnpmfile.cjs`.

By default, `.pnpmfile.cjs` should be located in the same directory as the lockfile. For instance, in a workspace with a shared lockfile, `.pnpmfile.cjs` should be in the root of the monorepo.

Hooks

TL;DR

Hook Function	Process	Uses
`hooks.readPackage(pkg, context): pkg`	Called after pnpm parses the dependency's package manifest	Allows you to mutate a dependency's `package.json`
`hooks.afterAllResolved(lockfile, context): lockfile`	Called after the dependencies have been resolved.	Allows you to mutate the lockfile.

`hooks.readPackage(pkg, context): pkg | Promise<pkg>`

Allows you to mutate a dependency's `package.json` after parsing and prior to resolution. These mutations are not saved to the filesystem, however, they will affect what gets resolved in the lockfile and therefore what gets installed.

Note that you will need to delete the `pnpm-lock.yaml` if you have already

resolved the dependency you want to modify.

> If you need changes to `package.json` saved to the filesystem, you need to use the [`pnpm patch`] command and patch the `package.json` file. This might be useful if you want to remove the `bin` field of a dependency for instance.

Arguments

- `pkg` - The manifest of the package. Either the response from the registry or the `package.json` content.
- `context` - Context object for the step. Method `#log(msg)` allows you to use a debug log for the step.

Usage

Example `.pnpmfile.cjs` (changes the dependencies of a dependency):

```
function readPackage(pkg, context) {
  // Override the manifest of foo@1.x after downloading it from the registry
  if (pkg.name === 'foo' && pkg.version.startsWith('1.')) {
    // Replace bar@x.x.x with bar@2.0.0
    pkg.dependencies = {
      ...pkg.dependencies,
      bar: '^2.0.0'
    }
    context.log('bar@1 => bar@2 in dependencies of foo')
  }

  // This will change any packages using baz@x.x.x to use baz@1.2.3
  if (pkg.dependencies.baz) {
    pkg.dependencies.baz = '1.2.3';
  }

  return pkg
}

module.exports = {
  hooks: {
    readPackage
  }
}
```

Known limitations

Removing the `scripts` field from a dependency's manifest via `readPackage` will not prevent pnpm from building the dependency. When building a dependency, pnpm reads the `package.json` of the package from the package's archive, which is not affected by the hook. In order to ignore a package's build, use the pnpm.neverBuiltDependencies field.

`hooks.afterAllResolved(lockfile, context): lockfile | Promise<lockfile>`

Allows you to mutate the lockfile output before it is serialized.

Arguments

- `lockfile` - The lockfile resolutions object that is serialized to `pnpm-lock.yaml`.
- `context` - Context object for the step. Method `#log(msg)` allows you to use a debug log for the step.

Usage example

.pnpmfile.cjs

```
function afterAllResolved(lockfile, context) {
  // ...
  return lockfile
}

module.exports = {
  hooks: {
    afterAllResolved
  }
}
```

Known Limitations

There are none - anything that can be done with the lockfile can be modified via this function, and you can even extend the lockfile's functionality.

Related Configuration

ignore-pnpmfile

- Default: **false**
- Type: **Boolean**

`.pnpmfile.cjs` will be ignored. Useful together with `--ignore-scripts` when you want to make sure that no script gets executed during install.

pnpmfile

- Default: **.pnpmfile.cjs**
- Type: **path**
- Example: **.pnpm/.pnpmfile.cjs**

The location of the local pnpmfile.

global-pnpmfile

- Default: **null**
- Type: **path**
- Example: **~/.pnpm/global_pnpmfile.cjs**

The location of a global pnpmfile. A global pnpmfile is used by all projects during installation.

> It is recommended to use local pnpmfiles. Only use a global pnpmfile if you use pnpm on projects that don't use pnpm as the primary package manager.

`pnpm patch`

Go to TOC

Production

There are two ways to bootstrap your package in a production environment with pnpm. One of these is to commit the lockfile. Then, in your production environment, run `pnpm install` - this will build the dependency tree using the lockfile, meaning the dependency versions will be consistent with how they were when the lockfile was committed. This is the most effective way (and the one we recommend) to ensure your dependency tree persists across environments.

The other method is to commit the lockfile AND copy the package store to your production environment (you can change where with the store location option). Then, you can run `pnpm install --offline` and pnpm will use the packages from the global store, so it will not make any requests to the registry. This is recommended **ONLY** for environments where external access to the registry is unavailable for whatever reason.

Go to TOC

Scripts

How pnpm handles the `scripts` field of `package.json`.

Lifecycle Scripts

`pnpm:devPreinstall`

Runs only on local `pnpm install`.

Runs before any dependency is installed.

This script is executed only when set in the root project's `package.json`.

Go to TOC

Symlinked `node_modules` structure

> This article only describes how pnpm's `node_modules` are structured when there are no packages with peer dependencies. For the more complex scenario of dependencies with peers, see how peers are resolved

pnpm's `node_modules` layout uses symbolic links to create a nested structure of dependencies.

Every file of every package inside `node_modules` is a hard link to the content-addressable store. Let's say you install `foo@1.0.0` that depends on `bar@1.0.0`. pnpm will hard link both packages to `node_modules` like this:

```
node_modules
└── .pnpm
    ├── bar@1.0.0
    │   └── node_modules
    │       └── bar -> <store>/bar
    │           ├── index.js
    │           └── package.json
    └── foo@1.0.0
        └── node_modules
            └── foo -> <store>/foo
                ├── index.js
                └── package.json
```

These are the only real files in `node_modules`. Once all the packages are hard linked to `node_modules`, symbolic links are created to build the nested dependency graph structure.

As you might have noticed, both packages are hard linked into a subfolder inside a `node_modules` folder (`foo@1.0.0/node_modules/foo`). This is needed to:

1. **allow packages to import themselves.** `foo` should be able to `require('foo/package.json')` or `import * as package from foo/package.json`.
2. **avoid circular symlinks.** Dependencies of packages are placed in the same folder in which the dependent packages are. For Node.js it doesn't make a difference whether dependencies are inside the package's `node_modules` or in any other `node_modules` in the parent directories.

The next stage of installation is symlinking dependencies. `bar` is going to be symlinked to the `foo@1.0.0/node_modules` folder:

```
node_modules
└── .pnpm
    ├── bar@1.0.0
    │   └── node_modules
    │       └── bar -> <store>/bar
    └── foo@1.0.0
        └── node_modules
            ├── foo -> <store>/foo
            └── bar -> ../../bar@1.0.0/node_modules/bar
```

Next, direct dependencies are handled. `foo` is going to be symlinked into the root `node_modules` folder because `foo` is a dependency of the project:

```
node_modules
├── foo -> ./.pnpm/foo@1.0.0/node_modules/foo
└── .pnpm
    ├── bar@1.0.0
    |   └── node_modules
    |       └── bar -> <store>/bar
    └── foo@1.0.0
        └── node_modules
            ├── foo -> <store>/foo
            └── bar -> ../../bar@1.0.0/node_modules/bar
```

This is a very simple example. However, the layout will maintain this structure regardless of the number of dependencies and the depth of the dependency graph.

Let's add `qar@2.0.0` as a dependency of `bar` and `foo`. This is how the new structure will look:

```
node_modules
├── foo -> ./.pnpm/foo@1.0.0/node_modules/foo
└── .pnpm
    ├── bar@1.0.0
    |   └── node_modules
    |       ├── bar -> <store>/bar
    |       └── qar -> ../../qar@2.0.0/node_modules/qar
    ├── foo@1.0.0
    |   └── node_modules
    |       ├── foo -> <store>/foo
    |       ├── bar -> ../../bar@1.0.0/node_modules/bar
    |       └── qar -> ../../qar@2.0.0/node_modules/qar
    └── qar@2.0.0
        └── node_modules
            └── qar -> <store>/qar
```

As you may see, even though the graph is deeper now (`foo > bar > qar`), the directory depth in the file system is still the same.

This layout might look weird at first glance, but it is completely compatible with Node's module resolution algorithm! When resolving modules, Node ignores symlinks, so when `bar` is required from `foo@1.0.0/node_modules/foo/index.js`, Node does not use `bar` at `foo@1.0.0/node_modules/bar`, but instead, `bar` is resolved to its real location (`bar@1.0.0/node_modules/bar`). As a consequence, `bar` can also resolve its dependencies which are in `bar@1.0.0/node_modules`.

A great bonus of this layout is that only packages that are really in the dependencies are accessible. With a flattened `node_modules` structure, all hoisted packages are accessible. To read more about why this is an advantage, see [pnpm's strictness helps to avoid silly bugs][bugs]

bugs

Go to TOC

Uninstalling pnpm

Removing the globally installed packages

Before removing the pnpm CLI, it might make sense to remove all global packages that were installed by pnpm.

To list all the global packages, run `pnpm ls -g`. There are two ways to remove the global packages:

1. Run `pnpm rm -g <pkg>...` with each global package listed.
2. Run `pnpm root -g` to find the location of the global directory and remove it manually.

Removing the pnpm CLI

If you used the standalone script to install pnpm, then you should be able to uninstall the pnpm CLI by removing the pnpm home directory:

```
rm -rf $PNPM_HOME
```

You might also want to clean the `PNPM_HOME` env variable in your shell configuration file (`$HOME/.bashrc`, `$HOME/.zshrc` or `$HOME/.config/fish/config.fish`).

If you used npm to install pnpm, then you should use npm to uninstall pnpm:

```
npm rm -g pnpm
```

Removing the global content-addressable store

```
rm -rf $(pnpm store path)
```

If you used pnpm in non-primary disks, then you must run the above command in every disk, where pnpm was used. pnpm creates one store per disk.

Go to TOC

Using Changesets with pnpm

> At the time of writing this documentation, the latest pnpm version was v6.14. The latest Changesets version was v2.16.0.)

Setup

To setup changesets on a pnpm workspace, install changesets as a dev dependency in the root of the workspace:

```
pnpm add -Dw @changesets/cli
```

Then changesets' init command:

```
pnpm changeset init
```

Adding new changesets

To generate a new changeset, run `pnpm changeset` in the root of the repository. The generated markdown files in the `.changeset` directory should be committed to the repository.

Releasing changes

1. Run `pnpm changeset version`. This will bump the versions of the packages previously specified with `pnpm changeset` (and any dependents of those) and update the changelog files.
2. Run `pnpm install`. This will update the lockfile and rebuild packages.
3. Commit the changes.
4. Run `pnpm publish -r`. This command will publish all packages that have bumped versions not yet present in the registry.

Using GitHub Actions

To automate the process, you can use `changeset version` with GitHub actions.

Bump up package versions

The action will detect when changeset files arrive in the `main` branch, and then open a new PR listing all the packages with bumped versions. Once merged, the packages will be updated and you can decide whether to publish or not by adding the `publish` property.

Publishing

Add a new script `ci:publish` which executes `pnpm publish -r`. It will publish to the registry once the PR is opened by `changeset version`.

package.json

```
{
   scripts: {
      ci:publish: pnpm publish -r
   },
   ...
}
```

```
name: Changesets
on:
  push:
    branches:
      - main
env:
  CI: true
  PNPM_CACHE_FOLDER: .pnpm-store
jobs:
  version:
    timeout-minutes: 15
    runs-on: ubuntu-latest
    steps:
      - name: checkout code repository
        uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - name: setup node.js
        uses: actions/setup-node@v3
        with:
          node-version: 14
      - name: install pnpm
        run: npm i pnpm@latest -g
      - name: Setup npmrc
        run: echo //registry.npmjs.org/:_authToken=${{ secrets.NPM_TOKEN }} >
.npmrc
      - name: setup pnpm config
        run: pnpm config set store-dir $PNPM_CACHE_FOLDER
      - name: install dependencies
        run: pnpm install
      - name: create and publish versions
        uses: changesets/action@v1
        with:
          version: pnpm ci:version
          commit: chore: update versions
          # chore: update versions
          publish: pnpm ci:publish
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

More info and documentation regarding this action can be found

here.)

Go to TOC

Workspace

pnpm has built-in support for monorepositories (AKA multi-package repositories, multi-project repositories, or monolithic repositories). You can create a workspace to unite multiple projects inside a single repository.

A workspace must have a `pnpm-workspace.yaml` file in its root. A workspace also may have an `.npmrc` in its root.

> If you are looking into monorepo management, you might also want to look into [Bit]. Bit uses pnpm under the hood but automates a lot of the things that are currently done manually in a traditional workspace managed by pnpm/npm/Yarn. There's an article about `bit install` that talks about it: [Painless Monorepo Dependency Management with Bit].

Bit Painless Monorepo Dependency Management with Bit

Workspace protocol (workspace:)

By default, pnpm will link packages from the workspace if the available packages match the declared ranges. For instance, `foo@1.0.0` is linked into `bar` if `bar` has `foo: ^1.0.0` in its dependencies and `foo@1.0.0` is in the workspace. However, if `bar` has `foo: 2.0.0` in dependencies and `foo@2.0.0` is not in the workspace, `foo@2.0.0` will be installed from the registry. This behavior introduces some uncertainty.

Luckily, pnpm supports the `workspace:` protocol. When this protocol is used, pnpm will refuse to resolve to anything other than a local workspace package. So, if you set `foo: workspace:2.0.0`, this time installation will fail because `foo@2.0.0` isn't present in the workspace.

This protocol is especially useful when the link-workspace-packages option is set to `false`. In that case, pnpm will only link packages from the workspace if the `workspace:` protocol is used.

Referencing workspace packages through aliases

Let's say you have a package in the workspace named `foo`. Usually, you would reference it as `foo: workspace:*`.

If you want to use a different alias, the following syntax will work too: `bar: workspace:foo@*`.

Before publish, aliases are converted to regular aliased dependencies. The above example will become: `bar: npm:foo@1.0.0`.

Referencing workspace packages through their relative path

In a workspace with 2 packages:

```
+ packages
    + foo
    + bar
```

`bar` may have `foo` in its dependencies declared as `foo: workspace:../foo`. Before publishing, these specs are converted to regular version specs supported by all package managers.

Publishing workspace packages

When a workspace package is packed into an archive (whether it's through `pnpm pack` or one of the publish commands like `pnpm publish`), we dynamically replace any `workspace:` dependency by:

- The corresponding version in the target workspace (if you use `workspace:*`, `workspace:~`, or `workspace:^`)
- The associated semver range (for any other range type)

So for example, if we have `foo`, `bar`, `qar`, `zoo` in the workspace and they all are at version `1.5.0`, the following:

```
{
    dependencies: {
        foo: workspace:*,
        bar: workspace:~,
        qar: workspace:^,
        zoo: workspace:^1.5.0
    }
}
```

Will be transformed into:

```
{
    dependencies: {
        foo: 1.5.0,
        bar: ~1.5.0,
        qar: ^1.5.0,
        zoo: ^1.5.0
    }
}
```

This feature allows you to depend on your local workspace packages while still being able to publish the resulting packages to the remote registry without needing intermediary publish steps - your consumers will be able to use your published workspaces as any other package, still benefitting from the guarantees semver offers.

Release workflow

Versioning packages inside a workspace is a complex task and pnpm currently does not provide a built-in solution for it. However, there are 2 well tested tools that handle versioning and support pnpm:

- changesets)
- Rush.)

For how to set up a repository using Rush, read [this page][rush-setup].

For using Changesets with pnpm, read [this guide][changesets-guide].

rush-setup [changesets-guide]: using-changesets.html)

Troubleshooting

pnpm cannot guarantee that scripts will be run in topological order if there are cycles between workspace dependencies. If pnpm detects cyclic dependencies during installation, it will produce a warning. If pnpm is able to find out which dependencies are causing the cycles, it will display them too.

If you see the message `There are cyclic workspace dependencies`, please inspect workspace dependencies declared in `dependencies`, `optionalDependencies` and `devDependencies`.

Usage examples

Here are a few of the most popular open source projects that use the workspace feature of pnpm:

Project	Stars	Migration date	Migration commit
Next.js	stars 98k	2022-05-29	`f7b81316aea4fc9962e5e54981a6d559004231aa`
Vite	stars 50k	2021-09-26	`3e1cce01d01493d33e50966d0d0fd39a86d229f9`
Vue 3.0	stars 34k	2021-10-09	`61c5fbd3e35152f5f32e95bf04d3ee083414cecb`
Prisma	stars 28k	2021-09-21	`c4c83e788aa16d61bae7a6d00adc8a58b3789a06`
Slidev	stars 24k	2021-04-12	`d6783323eb1ab1fc612577eb63579c8f7bc99c3a`
Element Plus	stars 18k	2021-09-23	`f9e192535ff74d1443f1d9e0c5394fad10428629`
Verdaccio	stars 14k	2021-09-21	`9dbf73e955fcb70b0a623c5ab89649b95146c744`
Astro	stars 23k	2022-03-08	`240d88aefe66c7d73b9c713c5da42ae789c011ce`
Cycle.js	stars 10k	2021-09-21	`f2187ab6688368edb904b649bd371a658f6a8637`
VueUse	stars 13k	2021-09-25	`826351ba1d9c514e34426c85f3d69fb9875c7dd9`
NextAuth.js	stars 13k	2022-05-03	`4f29d39521451e859dbdb83179756b372e3dd7aa`

Project	Stars	Migration date	Migration commit
SvelteKit	stars 12k	2021-09-26	`b164420ab26fa04fd0fbe0ac05431f36a89ef193`
Milkdown	stars 6.9k	2021-09-26	`4b2e1dd6125bc2198fd1b851c4f00eda70e9b913`
Vitest	stars 7.2k	2021-12-13	`d6ff0ccb819716713f5eab5c046861f4d8e4f988`
Logto	stars 5k	2021-07-29	`0b002e07850c8e6d09b35d22fab56d3e99d77043`
Nhost	stars 5.3k	2022-02-07	`10a1799a1fef2f558f737de3bb6cadda2b50e58f`
Rollup plugins	stars 2.9k	2021-09-21	`53fb18c0c2852598200c547a0b1d745d15b5b487`
ByteMD	stars repo not found	2021-02-18	`36ef25f1ea1cd0b08752df5f8c832302017bb7fb`
icestark	stars 1.8k	2021-12-16	`4862326a8de53d02f617e7b1986774fd7540fccd`
VuePress 2.0	stars 1.5k	2022-04-23	`b85b1c3b39e80a8de92a7469381061f75ef33623`
Turborepo	stars 18k	2022-03-02	`fd171519ec02a69c9afafc1bc5d9d1b481fba721`

Go to TOC

Colophon

This book is created by using the following sources:

- Pnpm - English
- GitHub source: pnpm/pnpm.github.io/docs
- Created: 2022-12-10
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf