Package 'dockerfiler'

May 14, 2026

Title Easy Dockerfile Creation from R

Version 1.0.0

Description Build a Dockerfile straight from your R session.
'dockerfiler' allows you to create step by step a Dockerfile, and
provide convenient tools to wrap R code inside this Dockerfile.

License MIT + file LICENSE

URL https://thinkr-open.github.io/dockerfiler/,
https://github.com/ThinkR-open/dockerfiler

BugReports https://github.com/ThinkR-open/dockerfiler/issues

Imports attempt (>= 0.3.1), cli (>= 2.3.0), desc (>= 1.2.0), fs (>=
1.5.0), glue (>= 1.4.2), jsonlite (>= 1.7.2), memoise, pak (>=
0.6.0), pkgbuild (>= 1.2.0), purrr, R6 (>= 2.5.0), remotes (>=
2.2.0), usethis (>= 2.0.1), utils

Suggests knitr (>= 1.31), renv, rmarkdown (>= 2.6), testthat (>=
3.0.0), withr

VignetteBuilder knitr

Config/fusen/version 0.6.0

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Author Colin Fay [cre, aut] (ORCID: <https://orcid.org/0000-0001-7343-1846>),
Vincent Guyader [aut] (ORCID: <https://orcid.org/0000-0003-0671-9270>),
Josiah Parry [aut] (ORCID: <https://orcid.org/0000-0001-9910-865X>),
Sébastien Rochette [aut] (ORCID:
<https://orcid.org/0000-0002-1565-9313>)

Maintainer Colin Fay <contact@colinfay.me>

Repository CRAN

Date/Publication 2026-05-14 06:30:02 UTC

Contents

`compact_sysreqs` *Compact Sysreqs*

Description

Compact Sysreqs

Usage

```
compact_sysreqs(
  pkg_installs,
  update_cmd = "apt-get update -y",
  install_cmd = "apt-get install -y",
  clean_cmd = "rm -rf /var/lib/apt/lists/*"
)
```

Arguments

`pkg_installs`	pkg_sysreqs as vector, `pak::pkg_sysreqs` output
`update_cmd`	command used to update packages, "apt-get update -y" by default
`install_cmd`	command used to install packages, "apt-get install -y" by default
`clean_cmd`	command used to clean package folder, "rm -rf /var/lib/apt/lists/*" by default

Value

vector of compacted command to run to install sysreqs

Examples

```
pkg_installs <- list("apt-get install -y htop", "apt-get install -y top")
compact_sysreqs(pkg_installs)
```

`Dockerfile` *A Dockerfile template*

Description

A Dockerfile template

A Dockerfile template

Public fields

`Dockerfile` The dockerfile content.

Methods

Public methods:

- `Dockerfile$new()`
- `Dockerfile$RUN()`
- `Dockerfile$ADD()`
- `Dockerfile$COPY()`
- `Dockerfile$WORKDIR()`
- `Dockerfile$EXPOSE()`
- `Dockerfile$VOLUME()`
- `Dockerfile$CMD()`
- `Dockerfile$LABEL()`
- `Dockerfile$ENV()`
- `Dockerfile$ENTRYPOINT()`
- `Dockerfile$USER()`
- `Dockerfile$ARG()`
- `Dockerfile$ONBUILD()`
- `Dockerfile$STOPSIGNAL()`
- `Dockerfile$HEALTHCHECK()`
- `Dockerfile$SHELL()`
- `Dockerfile$MAINTAINER()`
- `Dockerfile$custom()`
- `Dockerfile$COMMENT()`
- `Dockerfile$print()`
- `Dockerfile$write()`
- `Dockerfile$switch_cmd()`
- `Dockerfile$remove_cmd()`
- `Dockerfile$add_after()`
- `Dockerfile$clone()`

Method `new()`**:** Create a new Dockerfile object.

Usage:

```
Dockerfile$new(FROM = "rocker/r-base", AS = NULL)
```

Arguments:

`FROM` The base image. Default `"rocker/r-base"`. (Note: the high-level generators `dock_from_desc()` and `dock_from_renv()` use a different default, `rocker/r-ver` tagged with your R version.)

`AS` Optional build-stage name (`FROM ... AS <name>`). Default `NULL` (no `AS`).

Returns: A Dockerfile object.

Method `RUN()`**:** Add a RUN command.

Usage:

```
Dockerfile$RUN(cmd)
```

Arguments:

`cmd` The command to add.

Returns: the Dockerfile object, invisibly.

Method `ADD()`**:** Add a ADD command.

Usage:

```
Dockerfile$ADD(from, to, force = TRUE)
```

Arguments:

`from` The source file.

`to` The destination file.

`force` If TRUE, overwrite the destination file.

Returns: the Dockerfile object, invisibly.

Method `COPY()`**:** Add a COPY command.

Usage:

```
Dockerfile$COPY(from, to, stage = NULL, force = TRUE)
```

Arguments:

`from` The source file.

`to` The destination file.

`stage` Optional. Name of the build stage (e.g., `"builder"`) to copy files from. This corresponds to the `--from=` part in a Dockerfile COPY instruction (e.g., `COPY --from=builder /source /dest`). If `NULL`, the `--from=` argument is omitted.

`force` If TRUE, overwrite the destination file.

Returns: the Dockerfile object, invisibly.

Method `WORKDIR()`**:** Add a WORKDIR command.

Usage:

```
Dockerfile$WORKDIR(where)
```

Arguments:

`where` The working directory.

Returns: the Dockerfile object, invisibly.

Method `EXPOSE()`**:** Add a EXPOSE command.

Usage:

```
Dockerfile$EXPOSE(port)
```

Arguments:

`port` The port to expose.

Returns: the Dockerfile object, invisibly.

Method `VOLUME()`**:** Add a VOLUME command.

Usage:

```
Dockerfile$VOLUME(volume)
```

Arguments:

`volume` The volume to add.

Returns: the Dockerfile object, invisibly.

Method `CMD()`**:** Add a CMD command.

Usage:

```
Dockerfile$CMD(cmd)
```

Arguments:

`cmd` The command to add.

Returns: the Dockerfile object, invisibly.

Method `LABEL()`**:** Add a LABEL command.

Usage:

```
Dockerfile$LABEL(key, value)
```

Arguments:

`key, value` The key and value of the label.

Returns: the Dockerfile object, invisibly.

Method `ENV()`**:** Add an ENV command.

Usage:

```
Dockerfile$ENV(key, value)
```

Arguments:

`key, value` The key and value of the environment variable.

Returns: the Dockerfile object, invisibly.

Method `ENTRYPOINT()`**:** Add a ENTRYPOINT command.

Usage:

```
Dockerfile$ENTRYPOINT(cmd)
```

Arguments:

`cmd` The command to add.

Returns: the Dockerfile object, invisibly.

Method `USER()`**:** Add a USER command.

Usage:

```
Dockerfile$USER(user)
```

Arguments:

`user` The user to add.

Returns: the Dockerfile object, invisibly.

Method `ARG()`**:** Add a ARG command.

Usage:

```
Dockerfile$ARG(arg, ahead = FALSE, default = NULL)
```

Arguments:

`arg` The argument to add.

`ahead` If TRUE, add the argument at the beginning of the Dockerfile.

`default` Optional default value. When not `NULL`, the directive becomes `ARG <arg>=<default>`. Requires `arg` to be just the name (no embedded `=`); supplying both an inlined `=` in `arg` and a non-`NULL` `default` errors.

Returns: the Dockerfile object, invisibly.

Method `ONBUILD()`**:** Add a ONBUILD command.

Usage:

```
Dockerfile$ONBUILD(cmd)
```

Arguments:

`cmd` The command to add.

Returns: the Dockerfile object, invisibly.

Method `STOPSIGNAL()`**:** Add a STOPSIGNAL command.

Usage:

```
Dockerfile$STOPSIGNAL(signal)
```

Arguments:

`signal` The signal to add.

Returns: the Dockerfile object, invisibly.

Method `HEALTHCHECK()`**:** Add a HEALTHCHECK command.

Usage:

```
Dockerfile$HEALTHCHECK(check)
```

Arguments:

`check` The check to add.

Returns: the Dockerfile object, invisibly.

Method `SHELL()`**:** Add a SHELL command.

Usage:

```
Dockerfile$SHELL(shell)
```

Arguments:

`shell` The shell to add.

Returns: the Dockerfile object, invisibly.

Method `MAINTAINER()`**:** Add a MAINTAINER command.

Usage:

```
Dockerfile$MAINTAINER(name, email)
```

Arguments:

`name, email` The name and email of the maintainer.

Returns: the Dockerfile object, invisibly.

Method `custom()`**:** Add a custom command.

Usage:

```
Dockerfile$custom(base, cmd)
```

Arguments:

`base, cmd` The base and command to add.

Returns: the Dockerfile object, invisibly.

Method `COMMENT()`**:** Add a comment.

Usage:

```
Dockerfile$COMMENT(comment)
```

Arguments:

`comment` The comment to add.

Returns: the Dockerfile object, invisibly.

Method `print()`**:** Print the Dockerfile.

Usage:

```
Dockerfile$print()
```

Returns: used for side effect

Method `write()`**:** Write the Dockerfile to a file.

Usage:

```
Dockerfile$write(as = "Dockerfile", append = FALSE)
```

Arguments:

`as` The file to write to.

`append` boolean, if TRUE append to file.

Returns: used for side effect

Method `switch_cmd()`**:** Switch commands.

Usage:

```
Dockerfile$switch_cmd(a, b)
```

Arguments:

`a, b` The commands to switch.

Returns: the Dockerfile object, invisibly.

Method `remove_cmd()`**:** Remove a command.

Usage:

```
Dockerfile$remove_cmd(where)
```

Arguments:

`where` The commands to remove.

Returns: the Dockerfile object, invisibly.

Method `add_after()`**:** Add a command after another.

Usage:

```
Dockerfile$add_after(cmd, after)
```

Arguments:

`cmd` The command to add.
`after` Where to add the cmd

Returns: the Dockerfile object, invisibly.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Dockerfile$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
my_dock <- Dockerfile$new()
```

`docker_ignore_add` *Create a dockerignore file*

Description

Create a dockerignore file

Usage

```
docker_ignore_add(path)
```

Arguments

path	Where to write the file

Value

The path to the .dockerignore file, invisibly.

Examples

```
## Not run:
  docker_ignore_add()

## End(Not run)
```

---

dock_from_desc *Create a Dockerfile from a DESCRIPTION*

Description

Create a Dockerfile from a DESCRIPTION

Usage

```
dock_from_desc(
  path = "DESCRIPTION",
  FROM = paste0("rocker/r-ver:", R.Version()$major, ".", R.Version()$minor),
  AS = NULL,
  sysreqs = TRUE,
  repos = c(CRAN = "https://p3m.dev/cran/latest"),
  expand = FALSE,
  update_tar_gz = TRUE,
  build_from_source = TRUE,
  extra_sysreqs = NULL,
  github_pat = c("none", "build_arg", "secret"),
  strict_install = TRUE
)
```

Arguments

path	path to the DESCRIPTION file to use as an input.
FROM	The FROM of the Dockerfile. Default is paste0("rocker/r-ver:", R.Version()$major, ".", R.Version()$minor). Validated as a Docker image reference (alphanumerics, dot, slash, dash, underscore, optional :tag and / or @sha256:<hex>); other values raise an error to prevent shell-metacharacter injection into the generated FROM directive.
AS	The build-stage name of the Dockerfile (FROM ... AS <name>). Default is NULL (no AS). When non-NULL, validated as a simple build-stage name (^[a-zA-Z0-9][a-zA-Z0-9._-]*$).

`sysreqs` boolean. If TRUE, the Dockerfile will contain sysreq installation.

`repos` character. The URL(s) of the repositories to use for `options("repos")`. Default is `c(CRAN = "https://p3m.dev/cran/latest")` (Posit Public Package Manager). When `repos` is a single `CRAN`-keyed PPM URL (`packagemanager.posit.co`, `packagemanager.rstudio.com`, or `p3m.dev`), the codegen rewrites it to the `__linux__/$VERSION_CODENAME/` shape (codename resolved from `/etc/os-release` at image build time) and adds the strict `HTTPUserAgent` PPM requires, so the build pulls pre-compiled Linux binaries instead of compiling from source. Pass the legacy `c(CRAN = "https://cran.rstudio.com/")` to opt out. Each value must look like an http(s) URL (no quotes, spaces or newlines); each name (when set) must be a simple identifier (`^[A-Za-z][A-Za-z0-9._-]*$`). Other values raise an error to prevent injection into the generated `echo "options(...)"` shell command.

`expand` boolean. If `TRUE` each system requirement will have its own `RUN` line.

`update_tar_gz` boolean. If `TRUE` and `build_from_source` is also `TRUE`, an updated tar.gz is created.

`build_from_source` boolean. If `TRUE` no tar.gz is created and the Dockerfile directly mount the source folder.

`extra_sysreqs` character vector. Extra debian system requirements. Will be installed with apt-get install. Each entry must be a Debian package name (`^[a-z0-9][a-z0-9.+-]+$`); other values raise an error to prevent injection into the generated apt-get RUN.

`github_pat` character. How to provide a GitHub PAT to `remotes::install_github()` for private dependency repositories. One of `"none"` (default; the generated Dockerfile does not reference any PAT), `"build_arg"` (emit `ARG GITHUB_PAT` + `ENV` propagation; pass with `--build-arg GITHUB_PAT=$GITHUB_PAT`; the PAT will be visible in the image metadata), or `"secret"` (BuildKit secret mount on each `install_github()` / `install_local()` RUN; the PAT is never persisted in the image; requires BuildKit, so pass with `DOCKER_BUILDKIT=1 docker build --secret id=github_pat,e`

`strict_install` boolean. When `TRUE` (the default), every install RUN in the generated Dockerfile is prefixed with `options(warn = 2);` so that any R warning during install (missing CRAN package, partial download, archived package, 404 on a remote) becomes a hard error and aborts the docker build. Set to `FALSE` if your build environment routinely emits benign warnings (locale defaulting, NTP time-verification, ABI-version notices) that you do not want to fail the build. Must be a single scalar logical; `NA`, character, numeric, `NULL` and length-2+ vectors are rejected with an error.

Details

Two install strategies are available for the package itself:

- `build_from_source = TRUE` (the default): the generated Dockerfile mounts the source folder and installs from it directly. `update_tar_gz` is ignored.
- `build_from_source = FALSE`: a source tarball (`<pkg>_<version>.tar.gz`) is `COPY`'d into the image and installed with `remotes::install_local()`. When `update_tar_gz = TRUE`, a

fresh tarball is built with pkgbuild::build() first (and any stale <pkg>_*.tar.gz in the current directory is removed); when update_tar_gz = FALSE, an already-built tarball is expected alongside the DESCRIPTION.

The package name and its dependency-field names are read from the DESCRIPTION and validated against the CRAN package-name grammar before being interpolated into the generated directives.

Value

Dockerfile

Examples

```
## Not run:
# From the DESCRIPTION of the package in the working directory:
dock <- dock_from_desc("DESCRIPTION")
dock

# Pull source packages from the classic CRAN mirror instead of PPM:
dock_from_desc(
  "DESCRIPTION",
  repos = c(CRAN = "https://cran.rstudio.com/")
)

## End(Not run)
```

---

dock_from_renv *Create a Dockerfile from an* renv.lock *file*

Description

Create a Dockerfile from an renv.lock file

Usage

```
dock_from_renv(
  lockfile = "renv.lock",
  distro = NULL,
  FROM = "rocker/r-ver",
  AS = NULL,
  sysreqs = TRUE,
  repos = c(CRAN = "https://p3m.dev/cran/latest"),
  expand = FALSE,
  extra_sysreqs = NULL,
  use_pak = FALSE,
  user = "rstudio",
  dependencies = NA,
  sysreqs_platform = "ubuntu",
  renv_version,
```

```
  github_pat = c("none", "build_arg", "secret"),
  renv_paths_cache = NULL
)
```

Arguments

lockfile	Path to an `renv.lock` file to use as an input. The `basename(lockfile)` must be located at the docker build context root at `docker build` time, because the generated Dockerfile emits `COPY <basename(lockfile)> renv.lock`. Validated as a single string whose basename contains only alphanumerics, dots, underscores or hyphens (no spaces or shell metacharacters).
distro	• deprecated - only debian/ubuntu based images are supported
FROM	Docker image to start FROM. Default is `"rocker/r-ver"`, which is multi-arch (linux/amd64 + linux/arm64) and gets the lockfile's R version appended at codegen time (e.g. `rocker/r-ver:4.5.0`). Pass an already-tagged or already-digested reference (`rocker/r-ver:4.4.1`, `rocker/r-base@sha256:...`) to override the auto-tag; the user's tag is honoured verbatim, even if it differs from the lockfile's `R$Version`. R-devel and release-candidate users whose lockfile records `r-devel` or `4.5.0-RC` may want to pass an explicit tag like `FROM = "rocker/r-ver:devel"` to control which base image is pulled. Validated as a Docker image reference (`<host>[:<port>]/<image>[:<tag>][@sha256:<hex>]`); other values raise an error to prevent shell-metacharacter injection into the generated FROM directive.
AS	The AS of the Dockerfile. Default is `NULL`. When non-NULL, validated as a simple build-stage name (`^[a-zA-Z0-9][a-zA-Z0-9._-]*$`).
sysreqs	boolean. If `TRUE`, the Dockerfile will contain sysreq installation.
repos	character. The URL(s) of the repositories to use for `options("repos")`. Default is `c(CRAN = "https://p3m.dev/cran/latest")` (Posit Public Package Manager). When the URL is recognized as a PPM host (`packagemanager.posit.co`, `packagemanager.rstudio.com`, or `p3m.dev`), the codegen rewrites it to the `__linux__/$VERSION_CODENAME/` shape so the build pulls Linux binaries (5-10x faster than building from source). Each value must look like an http(s) URL (no quotes, spaces or newlines); each name (when set) must be a simple identifier (`^[A-Za-z][A-Za-z0-9._-]*$`). Other values raise an error to prevent injection into the generated `echo "options(...)"` shell command.
expand	boolean. If `TRUE` each system requirement will have its own `RUN` line.
extra_sysreqs	character vector. Extra debian system requirements. Will be installed with apt-get install. Each entry must be a Debian package name (`^[a-z0-9][a-z0-9.+-]+$`); other values raise an error to prevent injection into the generated apt-get RUN.
use_pak	boolean. If `TRUE` use pak to deal with dependencies during `renv::restore()`. FALSE by default. Must be a single `TRUE` or `FALSE` (no `NA`, no vector).
user	Name of the user the runtime container drops privilege to before the `renv::restore()` step (and therefore at runtime). Default is `"rstudio"` so the generated image runs as a non-root user out of the box, which is the recommended security posture.

The Dockerfile is emitted in two halves: every step that needs root (apt-get, R install commands, chown of the renv cache) runs first; then a `USER <user>` directive drops privilege; then the `renv::restore()` cache-mount RUN happens.

To make this work regardless of the FROM image, the package emits a defensive `RUN id -u <user> >/dev/null 2>&1 || useradd -m -d /home/<user> -s /bin/bash <user>` early. On rocker/* images the useradd is a no-op (the user already exists); on `r-base`, `ubuntu:*`, `debian:*` it creates the user with the standard home directory.

Pass `user = NULL` to opt out: no `USER` directive is emitted and the container runs as root (the previous behaviour). Pass any other string to use that user instead of `rstudio`. Custom homes (e.g. `/srv/myapp`) require also passing an explicit `renv_paths_cache`.

debian/ubuntu images only (useradd is the standard form); for alpine-based images you must pass `user = NULL` and handle user creation yourself with the alpine-native `adduser`.

The argument is validated at codegen time against `^[a-zA-Z_][a-zA-Z0-9_-]{0,31}$` (POSIX-style username, max 32 chars, no shell metacharacters). Other values raise an error to prevent metacharacter injection into the generated `RUN` commands.

`dependencies` What kinds of dependencies to install. Most commonly one of the following values:

- `NA`: only required (hard) dependencies,
- `TRUE`: required dependencies plus optional and development dependencies,
- `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.)

`sysreqs_platform`

System requirements platform.ubuntu by default. If `NULL`, then the current platform is used. Can be : "ubuntu-22.04" if needed to fit with the `FROM` Operating System. Only debian or ubuntu based images are supported

`renv_version` character or `NULL`. The renv version to install. The argument has three distinct modes, deliberately encoded with the missing-vs-`NULL` distinction:

- **not supplied (default)**: read the renv version from the `renv.lock` file. If the lockfile does not pin renv either, the latest available version is installed.
- `NULL` (explicit): always install the latest renv from the configured repos, even when the lockfile pins a specific version.
- a character string such as `"1.0.0"`: install that specific version regardless of what the lockfile says.

When supplied as a string, validated as a version-like token (`^[0-9]+(\.[0-9]+){0,3}([-.][a-zA-Z0-9`

`github_pat` character. How to provide a GitHub PAT to `renv::restore()` for private dependency repositories. One of `"none"` (default; the generated Dockerfile does not reference any PAT), `"build_arg"` (emit `ARG GITHUB_PAT` + `ENV` propagation; pass with `--build-arg GITHUB_PAT=$GITHUB_PAT`; the PAT will be visible in the image metadata), or `"secret"` (BuildKit secret mount on the `renv::restore()` RUN; the PAT is never persisted in the image; requires BuildKit, so pass with `DOCKER_BUILDKIT=1 docker build --secret id=github_pat,env=GITHUB_PAT ...`

renv_paths_cache
: character or NULL. Path used as the default of the RENV_PATHS_CACHE build-arg, propagated as an ENV variable, and used as the cache mount target for renv::restore(). Lets users override the renv cache location at image build time via --build-arg RENV_PATHS_CACHE=....

  When NULL (the default), the cache path is derived from user: /root/.cache/R/renv when user = NULL, and /home/<user>/.cache/R/renv when user is a non-root username. Pass an explicit string to override the convention (e.g. for custom-home users like user = "myapp" with home at /srv/myapp, pass renv_paths_cache = "/srv/myapp/.cache/R/renv").

  In all cases (user = NULL excepted), the generated Dockerfile emits a single RUN mkdir -p "${RENV_PATHS_CACHE}" && chown -R <user>:<user> "${RENV_PATHS_CACHE}" step right before the USER <user> directive so the cache mount target is writable from the un-privileged user. The cache path is double-quoted at shell expansion time so a build-arg override containing whitespace cannot break the command.

Details

System requirements for packages are provided through RStudio Package Manager via the pak package. The install commands provided from pak are added as RUN directives within the Dockerfile.

The R version is taken from the renv.lock file. Packages are installed using renv::restore() which ensures that the proper package version and source is used when installed.

Value

A R6 object of class Dockerfile.

Examples

```
## Not run:
dock <- dock_from_renv("renv.lock")
dock$write("Dockerfile")

## End(Not run)
```

---

get_sysreqs *Get system requirements*

Description

This function retrieves information about the system requirements using the pak::pkg_sysreqs().

Usage

```
get_sysreqs(packages, quiet = TRUE, batch_n = 30)
```

Arguments

packages	character vector. Packages names.
quiet	Boolean. If TRUE the function is quiet.
batch_n	numeric. Number of simultaneous packages to ask.

Value

A vector of system requirements.

Examples

```
## Not run:
get_sysreqs("glue")
get_sysreqs(c("curl", "xml2"))

## End(Not run)
```

---

parse_dockerfile *Parse a Dockerfile*

Description

Create a Dockerfile object from a Dockerfile.

Usage

```
parse_dockerfile(path)
```

Arguments

path	path to the Dockerfile

Value

A Dockerfile object

Examples

```
parse_dockerfile(system.file("Dockerfile", package = "dockerfiler"))
```

r *Turn an R expression into a shell* `R -e '...' call`

Description

Captures an R expression unevaluated and renders it as a single shell-quoted `R -e '...'` string, suitable for a Dockerfile `$RUN()` directive.

Usage

```
r(code)
```

Arguments

code	an R expression (captured unevaluated) to wrap.

Value

a length-1 character string of the form `R -e '...'`, shell-quoted with `base::shQuote()`.

Examples

```
r(print("yeay"))
r(install.packages("plumber", repos = "https://cloud.r-project.org"))
```

Index