Package 'renv'

August 4, 2026

Type Package

Title Project Environments

Version 1.2.4

Description A dependency management toolkit for R. Using 'renv', you can create and manage project-local R libraries, save the state of these libraries to a 'lockfile', and later restore your library as required. Together, these tools can help make your projects more isolated, portable, and reproducible.

License MIT + file LICENSE

URL https://rstudio.github.io/renv/, https://github.com/rstudio/renv

BugReports https://github.com/rstudio/renv/issues

Imports utils

Suggests BiocManager, cli, compiler, covr, cpp11, curl, devtools, generics, gitcreds, jsonlite, jsonvalidate, knitr, miniUI, modules, packrat, pak, R6, remotes, reticulate, rmarkdown, rstudioapi, shiny, testthat, uuid, waldo, yaml, webfakes

Encoding UTF-8

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first bioconductor,python,install,restore,snapshot,retrieve,remotes

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Kevin Ushey [aut, cre] (ORCID: <https://orcid.org/0000-0003-2880-7407>), Hadley Wickham [aut] (ORCID: <https://orcid.org/0000-0003-4757-117X>), Posit Software, PBC [cph, fnd]

Maintainer Kevin Ushey <kevin@rstudio.com>

Repository CRAN

Date/Publication 2026-08-03 23:10:02 UTC

Contents

renv-package *Project-local Environments for R*

Description

Project-local environments for R.

Details

You can use renv to construct isolated, project-local R libraries. Each project using renv will share package installations from a global cache of packages, helping to avoid wasting disk space on multiple installations of a package that might otherwise be shared across projects.

Author(s)

Maintainer: Kevin Ushey <kevin@rstudio.com> (ORCID)

Authors:

- Kevin Ushey <kevin@rstudio.com> (ORCID)
- Hadley Wickham <hadley@rstudio.com> (ORCID)

Other contributors:

- Posit Software, PBC [copyright holder, funder]

See Also

Useful links:

- https://rstudio.github.io/renv/
- https://github.com/rstudio/renv
- Report bugs at https://github.com/rstudio/renv/issues

activate *Activate or deactivate a project*

Description

`activate()` enables renv for a project in both the current session and in all future sessions. You should not generally need to call `activate()` yourself as it's called automatically by `init()`, which is the best way to start using renv in a new project.

`activate()` first calls `scaffold()` to set up the project infrastructure. Most importantly, this creates a project library and adds a an auto-loader to `.Rprofile` to ensure that the project library is automatically used for all future instances of the project. It then restarts the session to use that auto-loader.

`deactivate()` removes the infrastructure added by `activate()`, and restarts the session. By default it will remove the auto-loader from the `.Rprofile`; use `clean = TRUE` to also delete the lockfile and the project library.

Usage

```
activate(project = NULL, profile = NULL)

deactivate(project = NULL, clean = FALSE)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
profile	The profile to be activated. See vignette("profiles", package = "renv") for more information. When NULL (the default), the profile is not changed. Use profile = "default" to revert to the default renv profile.
clean	If TRUE, will also remove the renv/ directory and the lockfile.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Temporary deactivation

If you need to temporarily disable autoload activation you can set the RENV_CONFIG_AUTOLOADER_ENABLED envvar, e.g. Sys.setenv(RENV_CONFIG_AUTOLOADER_ENABLED = "false").

Examples

```
## Not run:

# activate the current project
renv::activate()

# activate a separate project
renv::activate(project = "~/projects/analysis")

# deactivate the currently-activated project
renv::deactivate()


## End(Not run)
```

---

autoload *Auto-load the active project*

Description

Automatically load the renv project associated with a particular directory. renv will search parent directories for the renv project root; if found, that project will be loaded via load().

Usage

```
autoload()
```

Details

To enable the renv auto-loader, you can place:

```
renv::autoload()
```

into your site-wide or user `.Rprofile` to ensure that renv projects are automatically loaded for any newly-launched R sessions, even if those R sessions are launched within the sub-directory of an renv project.

If you'd like to launch R within the sub-directory of an renv project without auto-loading renv, you can set the environment variable:

```
RENV_AUTOLOAD_ENABLED = FALSE
```

before starting R.

Note that `renv::autoload()` is only compatible with projects using `renv 0.15.3` or newer, as it relies on features within the `renv/activate.R` script that are only generated with newer versions of renv.

---

checkout *Checkout a repository*

Description

`renv::checkout()` can be used to retrieve the latest-available packages from a (set of) package repositories.

Usage

```
checkout(
  repos = NULL,
  ...,
  packages = NULL,
  date = NULL,
  clean = FALSE,
  actions = "restore",
  dependencies = NULL,
  restart = NULL,
  project = NULL
)
```

Arguments

repos	The R package repositories to use.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
packages	The packages to be installed. When NULL (the default), all packages currently used in the project will be installed, as determined by dependencies(). The recursive dependencies of these packages will be included as well.
date	The snapshot date to use. When set, the associated snapshot as available from the Posit's public Package Manager instance will be used. Ignored if repos is non-NULL.
clean	Boolean; remove packages not recorded in the lockfile from the target library? Use clean = TRUE if you'd like the library state to exactly reflect the lockfile contents after restore().
actions	The action(s) to perform with the requested repositories. This can either be "snapshot", in which renv will generate a lockfile based on the latest versions of the packages available from repos, or "restore" if you'd like to install those packages. You can use c("snapshot", "restore") if you'd like to generate a lockfile and install those packages in a single call. When actions contains "snapshot", the lockfile is generated from repository metadata (the PACKAGES file) without installing packages, so it may contain only a subset of the fields present in a lockfile generated by snapshot().
dependencies	A vector of DESCRIPTION field names that should be used for package dependency resolution. When NULL (the default), the value of renv::settings$package.dependency.fields is used. The aliases "strong", "most", and "all" are also supported. See tools::package_dependencies(for more details.
restart	Should the R session be restarted after the new packages have been checked out? When NULL (the default), the session is restarted if the "restore" action was taken.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

renv::checkout() is most useful with services like the Posit's Package Manager, as it can be used to switch between different repository snapshots within an renv project. In this way, you can upgrade (or downgrade) all of the packages used in a particular renv project to the package versions provided by a particular snapshot.

Note that calling renv::checkout() will also install the version of renv available as of the requested snapshot date, which might be older or lack features available in the currently-installed version of renv. In addition, the project's renv/activate.R script will be re-generated after checkout. If this is undesired, you can re-install a newer version of renv after checkout from your regular R package repository.

Caveats

If your library contains packages installed from other remote sources (e.g. GitHub), but a version of a package of the same name is provided by the repositories being checked out, then please be

aware that the package will be replaced with the version provided by the requested repositories. This could be a concern if your project uses R packages from GitHub whose name matches that of an existing CRAN package, but is otherwise unrelated to the package on CRAN.

Examples

```
## Not run:

# check out packages from PPM using the date '2023-01-02'
renv::checkout(date = "2023-01-02")

# alternatively, supply the full repository path
renv::checkout(repos = c(PPM = "https://packagemanager.rstudio.com/cran/2023-01-02"))

# only check out some subset of packages (and their recursive dependencies)
renv::checkout(packages = "dplyr", date = "2023-01-02")

# generate a lockfile based on a snapshot date
renv::checkout(date = "2023-01-02", actions = "snapshot")


## End(Not run)
```

---

clean *Clean a project*

Description

Clean up a project and its associated R libraries.

Usage

```
clean(project = NULL, ..., actions = NULL, prompt = interactive())
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
actions	The set of clean actions to take. See the documentation in **Actions** for a list of available actions, and the default actions taken when no actions are supplied.
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Actions

The following clean actions are available:

`package.locks` During package installation, R will create package locks in the library path, typically named `00LOCK-<package>`. On occasion, if package installation fails or R is terminated while installing a package, these locks can be left behind and will inhibit future attempts to reinstall that package. Use this action to remove such left-over package locks.

`library.tempdirs` During package installation, R may create temporary directories with names of the form `file\w{12}`, and on occasion those files can be left behind even after they are no longer in use. Use this action to remove such left-over directories.

`system.library` In general, it is recommended that only packages distributed with R are installed into the default library (the library path referred to by `.Library`). Use this action to remove any user-installed packages that have been installed to the system library.

Because this action is destructive, it is by default never run – it must be explicitly requested by the user.

`unused.packages` Remove packages that are installed in the project library, but no longer appear to be used in the project sources.

Because this action is destructive, it is by default only run in interactive sessions when prompting is enabled.

Examples

```
## Not run:

# clean the current project
renv::clean()


## End(Not run)
```

---

config *User-level settings*

Description

Configure different behaviors of renv.

Usage

```
config
```

Details

For a given configuration option:

1. If an R option of the form `renv.config.<name>` is available, then that option's value will be used;
2. If an environment variable of the form `RENV_CONFIG_<NAME>` is available, then that option's value will be used;
3. Otherwise, the default for that particular configuration value is used.

Any periods (`.`)s in the option name are transformed into underscores (`_`) in the environment variable name, and vice versa. For example, the configuration option `auto.snapshot` could be configured as:

- `options(renv.config.auto.snapshot = <...>)`
- `Sys.setenv(RENV_CONFIG_AUTO_SNAPSHOT = <...>)`

Note that if both the R option and the environment variable are defined, the R option will be used instead. Environment variables can be more useful when you want a particular configuration to be automatically inherited by child processes; if that behavior is not desired, then the R option may be preferred.

If you want to set and persist these options across multiple projects, it is recommended that you set them in a a startup `.Renviron` file; e.g. in your own `~/.Renviron`, or in the R installation's `etc/Rprofile.site` file. See Startup for more details.

Configuration options can also be set within the project `.Rprofile`, but be aware the options should be set before `source("renv/activate.R")` is called.

Configuration

The following renv configuration options are available:

renv.config.activate.prompt: Automatically prompt the user to activate the current project, if it does not appear to already be activated? This is mainly useful to help ensure that calls to `renv::snapshot()` and `renv::restore()` use the project library. See `?renv::activate` for more details. Defaults to `TRUE`.

renv.config.autoloader.enabled: Enable the renv auto-loader? When `FALSE`, renv will not automatically load a project containing an renv autoloader within its `.Rprofile`. In addition, renv will not write out the project auto-loader in calls to `renv::init()`. Defaults to `TRUE`.

renv.config.auto.snapshot: Automatically snapshot changes to the project library when the project dependencies change? Defaults to `FALSE`.

renv.config.bioconductor.init: The default value to be used for the `bioconductor` parameter in calls to `renv::init()`. This can be used if you'd like to automatically enable or disable Bioconductor by default for newly-initialized `renv` projects. See the documentation in `?renv::init` for more details. Defaults to `NULL`.

renv.config.bitbucket.host: The default Bitbucket host to be used during package retrieval. Defaults to `"api.bitbucket.org/2.0"`.

renv.config.copy.method: The method to use when attempting to copy directories. See **Copy Methods** for more information. Defaults to `"auto"`.

renv.config.crandb.enabled: Use the crandb service when looking up packages from CRAN which are not available in the configured repositories? When enabled, renv will query the crandb API to find the latest version of a package compatible with the current version of R. This can be useful when an older version of R is being used, and the latest version of a package on CRAN requires a newer version of R. Disabled by default. Defaults to `FALSE`.

renv.config.connect.timeout: The amount of time to spend (in seconds) when attempting to download a file. Only applicable when the `curl` downloader is used. Defaults to `20L`.

renv.config.connect.retry: The number of times to attempt re-downloading a file, when transient download errors occur. Only applicable when the `curl` downloader is used. Defaults to `3L`.

renv.config.cache.enabled: Enable the global renv package cache? When active, renv will install packages into a global cache, and link or copy packages from the cache into your R library as appropriate. This can greatly save on disk space and install time when R packages are shared across multiple projects in the same environment. Defaults to `TRUE`.

renv.config.cache.symlinks: Symlink packages from the global renv package cache into your project library? When `TRUE`, renv will use symlinks (or, on Windows, junction points) to reference packages installed in the cache. Set this to `FALSE` if you'd prefer to copy packages from the cache into your project library. Enabled by default, except on Windows where this feature is only enabled if the project library and global package cache are on the same volume. Defaults to `NULL`.

renv.config.dependency.errors: Many renv APIs require the enumeration of your project's R package dependencies. This option controls how errors that occur during this enumeration are handled. By default, errors are reported but are non-fatal. Set this to `"fatal"` to force errors to be fatal, and `"ignored"` to ignore errors altogether. See `dependencies()` for more details. Defaults to `"reported"`.

renv.config.dependencies.limit: By default, renv reports if it discovers more than this many files when looking for dependencies, as that may indicate you are running `dependencies()` in the wrong place. Set to `Inf` to disable this warning. Defaults to `1000L`.

renv.config.exported.functions: When `library(renv)` is called, should its exports be placed on the search path? Set this to `FALSE` to avoid issues that can arise with, for example, `renv::load()` masking `base::load()`. In general, we recommend referencing renv functions from its namespace explicitly; e.g. prefer `renv::snapshot()` over `snapshot()`. By default, all exported renv functions are attached and placed on the search path, for backwards compatibility with existing scripts using renv. Defaults to `"*"`.

renv.config.external.libraries: A character vector of external libraries, to be used in tandem with your projects. Be careful when using external libraries: it's possible that things can break within a project if the version(s) of packages used in your project library happen to be incompatible with packages in your external libraries; for example, if your project required `xyz 1.0` but `xyz 1.1` was present and loaded from an external library. Can also be an R function that provides the paths to external libraries. Library paths will be expanded via `.expand_R_libs_env_var()` as necessary. Defaults to `NULL`.

renv.config.filebacked.cache: Enable the renv file-backed cache? When enabled, renv will cache the contents of files that are read (e.g. DESCRIPTION files) in memory, thereby avoiding re-reading the file contents from the filesystem if the file has not changed. renv uses the file `mtime` to determine if the file has changed; consider disabling this if `mtime` is unreliable on your system. Defaults to `TRUE`.

renv.config.github.host: The default GitHub host to be used during package retrieval. Defaults to `"api.github.com"`.

renv.config.gitlab.host: The default GitLab host to be used during package retrieval. Defaults to `"gitlab.com"`.

renv.config.hydrate.libpaths: A character vector of library paths, to be used by `hydrate()` when attempting to hydrate projects. When empty, the default set of library paths (as documented in `?renv::hydrate`) are used instead. See `hydrate()` for more details. Defaults to `NULL`.

renv.config.install.build: Should downloaded package archives be built (via `R CMD build`) before installation? When TRUE, package vignettes will also be built as part of package installation. Because building packages before installation may require packages within 'Suggests' to be available, this option is not enabled by default. Defaults to `FALSE`.

renv.config.install.keep.source: Should `R CMD INSTALL` be invoked with `--with-keep.source` when installing packages from source? When `TRUE` (the default), renv preserves source references on installed functions, which can be useful for debugging. Set to `FALSE` to install packages with `--without-keep.source`; this can significantly reduce the serialized size of functions defined by installed packages. Defaults to `TRUE`.

renv.config.install.remotes: Should renv read a package's `Remotes:` field when determining how package dependencies should be installed? Defaults to `TRUE`.

renv.config.install.shortcuts: Allow for a set of 'shortcuts' when installing packages with renv? When enabled, if renv discovers that a package to be installed is already available within the user or site libraries, then it will install a local copy of that package. Defaults to `TRUE`.

renv.config.install.staged: DEPRECATED: Please use `renv.config.install.transactional` instead. Defaults to `TRUE`.

renv.config.install.transactional: Perform a transactional install of packages during install and restore? When enabled, renv will first install packages into a temporary library, and later copy or move those packages back into the project library only if all packages were successfully downloaded and installed. This can be useful if you'd like to avoid mutating your project library if installation of one or more packages fails. Defaults to `TRUE`.

renv.config.install.jobs: The number of packages to install concurrently during `install()` and `restore()`. Packages within the same dependency wave are installed in parallel using this many concurrent R CMD INSTALL processes. Set to `1L` for sequential installation. Defaults to `4L`.

renv.config.install.verbose: Be verbose when installing R packages from sources? When `TRUE`, renv will stream any output generated during package build + installation to the console. Defaults to `FALSE`.

renv.config.locking.enabled: Use interprocess locks when invoking methods which might mutate the project library? Enable this to allow multiple processes to use the same renv project, while minimizing risks relating to concurrent access to the project library. Disable this if you encounter locking issues. Locks are stored as files within the project at `renv/lock`; if you need to manually remove a stale lock you can do so via `unlink("renv/lock", recursive = TRUE)`. Defaults to `FALSE`.

renv.config.mran.enabled: DEPRECATED: MRAN is no longer maintained by Microsoft. Defaults to `FALSE`.

renv.config.namespaces.check: Check, on project load, whether any namespaces have already been loaded from a path outside the project's active library paths? Packages loaded before renv activates are not managed by renv and can cause `renv::status()` and `renv::snapshot()` to report inconsistencies. When enabled, renv emits a warning listing the offending packages and the paths they were loaded from. Defaults to `TRUE`.

renv.config.pak.enabled: Use the pak package to install packages? Defaults to `FALSE`.

renv.config.ppm.enabled: Boolean; enable Posit Package Manager integration in renv projects? When `TRUE`, renv will attempt to transform repository URLs used by PPM into binary URLs as appropriate for the current Linux platform. Set this to `FALSE` if you'd like to continue using source-only PPM URLs, or if you find that renv is improperly transforming your repository URLs. You can still set and use PPM repositories with this option disabled; it only controls whether renv tries to transform source repository URLs into binary URLs on your behalf. Defaults to `TRUE`.

renv.config.ppm.default: Boolean; should new projects use the Posit Public Package Manager instance by default? When `TRUE` (the default), projects initialized with `renv::init()` will use the P3M instance if the `repos` R option has not already been set by some other means (for example, in a startup `.Rprofile`). Defaults to `TRUE`.

renv.config.ppm.url: The default PPM URL to be used for new renv projects. Defaults to the CRAN mirror maintained by Posit at `https://packagemanager.posit.co/`. This option can be changed if you'd like renv to use an alternate package manager instance. Defaults to `"https://packagemanager.posit.co/cran/latest"`.

renv.config.repos.override: Override the R package repositories used during `restore()`? Primarily useful for deployment / continuous integration, where you might want to enforce the usage of some set of repositories over what is defined in `renv.lock` or otherwise set by the R session. This can be specified as a single repository URL (e.g., "https://cloud.r-project.org"), or as multiple named repositories separated by semicolons using the format NAME=URL (e.g., "CRAN=https://p3m.dev/cran/latest;R_UNIV=https://posit-dev.r-universe.dev"). Defaults to `NULL`.

renv.config.rspm.enabled: DEPRECATED: Please use `renv.config.ppm.enabled` instead. Defaults to `TRUE`.

renv.config.sandbox.enabled: Enable sandboxing for renv projects? When active, renv will attempt to sandbox the system library, preventing non-system packages installed in the system library from becoming available in renv projects. (That is, only packages with priority `"base"` or `"recommended"`, as reported by `installed.packages()`, are made available.) Sandboxing is done by linking or copying system packages into a separate library path, and then instructing R to use that library path as the system library path. In some environments, this action can take a large amount of time – in such a case, you may want to disable the renv sandbox. Defaults to `TRUE`.

renv.config.shims.enabled: Should renv shims be installed on package load? When enabled, renv will install its own shims over the functions `install.packages()`, `update.packages()` and `remove.packages()`, delegating these functions to `install()`, `update()` and `remove()` as appropriate. Defaults to `TRUE`.

renv.config.snapshot.inference: For packages which were installed from local sources, should renv try to infer the package's remote from its DESCRIPTION file? When `TRUE`, renv will check and prompt you to update the package's DESCRIPTION file if the remote source can be ascertained. Currently, this is only implemented for packages hosted on GitHub. Note that this check is only performed in interactive R sessions. Defaults to `TRUE`.

renv.config.snapshot.validate: Validate R package dependencies when calling snapshot? When `TRUE`, renv will attempt to diagnose potential issues in the project library before creating `renv.lock` – for example, if a package installed in the project library depends on a package which is not currently installed. Defaults to `TRUE`.

renv.config.startup.quiet: Be quiet during startup? When set, renv will not display the typical `Project <path> loaded. [renv <version>]` banner on startup. Defaults to `NULL`.

renv.config.synchronized.check: Check that the project library is synchronized with the lockfile on load? Defaults to `TRUE`.

renv.config.sysreqs.check: Check whether the requisite system packages are installed during package installation and restore? This feature uses the R System Requirements database maintained at https://github.com/rstudio/r-system-requirements. Defaults to `TRUE`.

renv.config.updates.check: Check for package updates when the session is initialized? This can be useful if you'd like to ensure that your project lockfile remains up-to-date with packages as they are released on CRAN. Defaults to `FALSE`.

renv.config.updates.parallel: Check for package updates in parallel? This can be useful when a large number of packages installed from non-CRAN remotes are installed, as these packages can then be checked for updates in parallel. Defaults to `2L`.

renv.config.user.environ: Load the user R environ (typically located at `~/.Renviron`) when renv is loaded? Defaults to `TRUE`.

renv.config.user.library: Include the system library on the library paths for your projects? Note that this risks breaking project encapsulation and is not recommended for projects which you intend to share or collaborate on with other users. See also the caveats for the `renv.config.external.libraries` option. Defaults to `FALSE`.

renv.config.user.profile: Load the user R profile (typically located at `~/.Rprofile`) when renv is loaded? This is disabled by default, as running arbitrary code from the the user `~/.Rprofile` could risk breaking project encapsulation. If your goal is to set environment variables that are visible within all renv projects, then placing those in `~/.Renviron` is often a better choice. Defaults to `FALSE`.

Copy methods

If you find that renv is unable to copy some directories in your environment, you may want to try setting the `copy.method` option. By default, renv will try to choose a system tool that is likely to succeed in copying files on your system – `robocopy` on Windows, and `cp` on Unix. renv will also instruct these tools to preserve timestamps and attributes when copying files. However, you can select a different method as appropriate.

The following methods are supported:

`auto`	Use `robocopy` on Windows, and `cp` on Unix-alikes.
`R`	Use R's built-in `file.copy()` function.
`cp`	Use `cp` to copy files.
`robocopy`	Use `robocopy` to copy files. (Only available on Windows.)
`rsync`	Use `rsync` to copy files.

You can also provide a custom copy method if required; e.g.

```
options(renv.config.copy.method = function(src, dst) {
  # copy a file from 'src' to 'dst'
})
```

Note that renv will always first attempt to copy a directory first to a temporary path within the target folder, and then rename that temporary path to the final target destination. This helps avoid issues where a failed attempt to copy a directory could leave a half-copied directory behind in the final location.

Project-local settings

For settings that should persist alongside a particular project, the various settings available in settings can be used.

Examples

```
# disable automatic snapshots
options(renv.config.auto.snapshot = FALSE)

# disable with environment variable
Sys.setenv(RENV_CONFIG_AUTO_SNAPSHOT = FALSE)
```

consent *Consent to usage of renv*

Description

Provide consent to renv, allowing it to write and update certain files on your filesystem.

Usage

```
consent(provided = FALSE)
```

Arguments

provided	The default provided response. If you need to provide consent from a non-interactive R session, you can invoke `renv::consent(provided = TRUE)` explicitly.

Details

As part of its normal operation, renv will write and update some files in your project directory, as well as an application-specific cache directory. These paths are documented within paths.

In accordance with the CRAN Repository Policy, renv must first obtain consent from you, the user, before these actions can be taken. Please call `renv::consent()` first to provide this consent.

You can also set the R option:

```
options(renv.consent = TRUE)
```

to implicitly provide consent for e.g. non-interactive R sessions.

Value

`TRUE` if consent is provided, or an R error otherwise.

dependencies *Find R package dependencies in a project*

Description

`dependencies()` will scan files within your project, looking for R files and the packages used within those R files. This is done primarily by parsing the code and looking for calls of the form `library(package)`, `require(package)`, `requireNamespace("package")`, and `package::method()`. renv also supports package loading with box (`box::use(...)`) and pacman (`pacman::p_load(...)`), as well as dependency checks of the form `rlang::check_installed("package")` and `rlang::is_installed("package")`.

For R package projects, `renv` will also detect dependencies expressed in the `DESCRIPTION` file. For projects using Python, R dependencies within the R code chunks of your project's `.ipynb` files will also be used.

Note that the `rmarkdown` package is required in order to scan dependencies in R Markdown files.

Usage

```
dependencies(
  path = getwd(),
  root = NULL,
  ...,
  quiet = NULL,
  progress = TRUE,
  errors = c("reported", "fatal", "ignored"),
  dev = FALSE
)
```

Arguments

path	The path to a .R, .Rmd, .qmd, DESCRIPTION, a directory containing such files, or an R function. The default uses all files found within the current working directory and its children.
root	The root directory to be used for dependency discovery. Defaults to the active project directory. You may need to set this explicitly to ensure that your project's .renvignores (if any) are properly handled.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
quiet	Boolean; be quiet while checking for dependencies? Setting quiet = TRUE is equivalent to setting progress = FALSE and errors = "ignored", and overrides those options when not NULL.
progress	Boolean; report progress output while enumerating dependencies?
errors	How should errors that occur during dependency enumeration be handled? • "reported" (the default): errors are reported to the user, but otherwise ignored. • "fatal": errors are fatal and stop execution. • "ignored": errors are ignored and not reported to the user.
dev	Boolean; include development dependencies? These packages are typically required when developing the project, but not when running it (i.e. you want them installed when humans are working on the project but not when computers are deploying it). Development dependencies include packages listed in the Suggests field of a DESCRIPTION found in the project root, and roxygen2 or devtools if their use is implied by other project metadata. They also include packages used in ~/.Rprofile if config$user.profile() is TRUE.

Value

An R data.frame of discovered dependencies, mapping inferred package names to the files in which they were discovered. Note that the Package field might name a package remote, rather than just a plain package name.

Missing dependencies

`dependencies()` uses static analysis to determine which packages are used by your project. This means that it inspects, but doesn't run, the R code in your project. Static analysis generally works well, but is not 100% reliable in detecting the packages required by your project. For example, `renv` is unable to detect this kind of usage:

```
for (package in c("dplyr", "ggplot2")) {
  library(package, character.only = TRUE)
}
```

It also can't generally tell if one of the packages you use, uses one of its suggested packages. For example, the `tidyr::separate_wider_delim()` function requires the `stringr` package, but `stringr` is only suggested, not required, by `tidyr`.

For a small number of common cases, renv infers these indirect dependencies from the combination of packages in use. For example, a project that uses `dplyr` together with a database backend (such as `DBI` and `RSQLite`) is assumed to also require `dbplyr`, which `dplyr` loads at runtime when working with databases. These rules are defined by the `renv.dependencies.implied` R option; set it to an empty list (`options(renv.dependencies.implied = list())`) to disable this inference entirely.

If you find that renv's dependency discovery misses one or more packages that you actually use in your project, one escape hatch is to include a file called `_dependencies.R` that includes straightforward library calls:

```
library(dplyr)
library(ggplot2)
library(stringr)
```

Ignoring files

By default, renv will read your project's `.gitignore`s (if present) to determine whether certain files or folders should be included when traversing directories. If preferred, you can also create a `.renvignore` file (with entries of the same format as a standard `.gitignore` file) to tell renv which files to ignore within a directory. If both `.renvignore` and `.gitignore` exist within a folder, the `.renvignore` will be used in lieu of the `.gitignore`.

See https://git-scm.com/docs/gitignore for documentation on the `.gitignore` format. Some simple examples here:

```
# ignore all R Markdown files
*.Rmd

# ignore all data folders
data/

# ignore only data folders from the root of the project
/data/
```

Using ignore files is important if your project contains a large number of files; for example, if you have a `data/` directory containing many text files.

Profile-specific Ignore Rules:

Profile-specific sections are also supported in `.renvignore` files. These sections are marked with a comment header of the form `#| <code>`, where `<code>` is R code that indicates if this section of the `.renvignore` should apply. The `profile` variable is set to the same value as the current profile, or `"default"` if the default profile (no profile) is selected. See `vignette("profiles", package = "renv")` for more information on profiles.

```
# ignore all directories by default
*/

#| profile == "default"
!default

#| profile == "extra"
!extra
```

Note that the first section in a `.renvignore` file implicitly applies to all profiles.

Errors

renv's attempts to enumerate package dependencies in your project can fail – most commonly, because of failures when attempting to parse your R code. You can use the `errors` argument to suppress these problems, but a more robust solution is tell renv not to look at the problematic code. As well as using `.renvignore`, as described above, you can also suppress errors discovered within individual `.Rmd` chunks by including `renv.ignore=TRUE` in the chunk header. For example:

````
```{r chunk-label, renv.ignore=TRUE}
# code in this chunk will be ignored by renv
```
````

Similarly, if you'd like renv to parse a chunk that is otherwise ignored (e.g. because it has `eval=FALSE` as a chunk header), you can set:

````
```{r chunk-label, eval=FALSE, renv.ignore=FALSE}
# code in this chunk will _not_ be ignored
```
````

Development dependencies

renv has some support for distinguishing between development and run-time dependencies. For example, your Shiny app might rely on ggplot2 (a run-time dependency) but while you use usethis during development, your app doesn't need it to run (i.e. it's only a development dependency).

You can record development dependencies by listing them in the `Suggests` field of your project's `DESCRIPTION` file. Development dependencies will be installed by `install()` (when called without arguments) but will not be tracked in the project snapshot. If you need greater control, you can also try project profiles as discussed in `vignette("profiles")`.

Examples

```
## Not run:

# find R package dependencies in the current directory
renv::dependencies()


## End(Not run)
```

`diagnostics` *Print a diagnostics report*

Description

Print a diagnostics report, summarizing the state of a project using renv. This report can occasionally be useful when diagnosing issues with renv.

Usage

```
diagnostics(project = NULL)
```

Arguments

`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Value

This function is normally called for its side effects.

`embed` *Capture and re-use dependencies within a* `.R`*,* `.Rmd` *or* `.qmd`

Description

Together, `embed()` and `use()` provide a lightweight way to specify and restore package versions within a file. `use()` is a lightweight lockfile specification that `embed()` can automatically generate and insert into a script or document.

Calling `embed()` inspects the dependencies of the specified document then generates and inserts a call to `use()` that looks something like this:

```
renv::use(
  "digest@0.6.30",
  "rlang@0.3.4"
)
```

When you next run your R script or render your `.Rmd` or `.qmd`, `use()` will:

1. Create a temporary library path,
2. Install the requested packages and their recursive dependencies into that library,
3. Activate the library, so it's used for the rest of the script.

Manual usage:

You can also create calls to `use()` yourself, either specifying the packages needed by hand, or by supplying the path to a lockfile, `renv::use(lockfile = "/path/to/renv.lock")`.

This can be useful in projects where you'd like to associate different lockfiles with different documents, as in a blog where you want each post to capture the dependencies at the time of writing. Once you've finished writing each, the post, you can use `renv::snapshot(lockfile = "/path/to/renv.lock")` to "save" the state that was active while authoring that bost, and then use `renv::use(lockfile = "/path/to/renv.lock")` in that document to ensure the blog post always uses those dependencies onfuture renders.

`renv::use()` is inspired in part by the groundhog package, which also allows one to specify a script's R package requirements within that same R script.

Usage

```
embed(path = NULL, ..., lockfile = NULL, project = NULL)

use(
  ...,
  lockfile = NULL,
  library = NULL,
  repos = getOption("repos"),
  isolate = TRUE,
  sandbox = TRUE,
  attach = FALSE,
  verbose = TRUE
)
```

Arguments

`path` The path to an R or R Markdown script. The default will use the current document, if running within RStudio.

Package Resolution:

When `embed()` generates a call to `use()`, it needs to determine the package versions to record. The `lockfile` parameter controls where these versions come from:

1. If `lockfile` is `NULL` (the default), and the project has an existing `renv.lock` file, that project lockfile is used. Only packages present in the lockfile will be included – any dependencies not recorded in the lockfile will be omitted.
2. If `lockfile` is a path to a lockfile, or a lockfile object, that lockfile is used directly.

3. If `lockfile` is `FALSE`, or if no project lockfile exists, package versions are inferred from the currently-installed packages in the active library paths.
4. If `lockfile` is `NA`, package versions are inferred from the latest versions available in the active package repositories.

`...`	The R packages to be used with this script. Ignored if `lockfile` is non-`NULL`.
`lockfile`	The lockfile to use. When supplied, renv will use the packages as declared in the lockfile.
`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.
`library`	The library path into which the requested packages should be installed. When `NULL` (the default), a library path within the R temporary directory will be generated and used. Note that this same library path will be re-used on future calls to `renv::use()`, allowing `renv::use()` to be used multiple times within a single script.
`repos`	The R package repositories to use. When `NULL`, packages will be resolved from the renv cache, without querying any external repositories. This can be useful if you'd like to use packages that have already been cached by renv, even when no active package repositories have been configured.
`isolate`	Boolean; should the active library paths be included in the set of library paths activated for this script? Set this to `TRUE` if you only want the packages provided to `renv::use()` to be visible on the library paths.
`sandbox`	Should the system library be sandboxed? See the sandbox documentation in config for more details. You can also provide an explicit sandbox path if you want to configure where `renv::use()` generates its sandbox. By default, the sandbox is generated within the R temporary directory.
`attach`	Boolean; should the set of requested packages be automatically attached? If `TRUE`, packages will be loaded and attached via a call to `library()` after install. Ignored if `lockfile` is non-`NULL`.
`verbose`	Boolean; be verbose while installing packages?

Value

This function is normally called for its side effects.

`history` *View and revert to a historical lockfile*

Description

`history()` uses your version control system to show prior versions of the lockfile and `revert()` allows you to restore one of them.

These functions are currently only implemented for projects that use git.

Usage

```
history(project = NULL)

revert(commit = "HEAD", ..., project = NULL)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
commit	The commit associated with a prior version of the lockfile.
...	Optional arguments; currently unused.

Value

history() returns a data.frame summarizing the commits in which renv.lock has been changed. revert() is usually called for its side-effect but also invisibly returns the commit used.

Examples

```
## Not run:

# get history of previous versions of renv.lock in VCS
db <- renv::history()

# choose an older commit
commit <- db$commit[5]

# revert to that version of the lockfile
renv::revert(commit = commit)


## End(Not run)
```

---

imbue *Imbue an renv Installation*

Description

Imbue an renv installation into a project, thereby making the requested version of renv available within.

Usage

```
imbue(project = NULL, version = NULL, quiet = FALSE)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
version	The version of renv to install. If NULL, the version of renv currently installed will be used. The requested version of renv will be retrieved from the renv public GitHub repository, at https://github.com/rstudio/renv.
quiet	Boolean; avoid printing output during install of renv?

Details

Normally, this function does not need to be called directly by the user; it will be invoked as required by init() and activate().

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

---

init *Use renv in a project*

Description

Call renv::init() to start using renv in the current project. This will:

1. Set up project infrastructure (as described in scaffold()) including the project library and the .Rprofile that ensures renv will be used in all future sessions,
2. Discover the packages that are currently being used in your project (via dependencies()), and install them into the project library (as described in hydrate()),
3. Create a lockfile that records the state of the project library so it can be restored by others (as described in snapshot()),
4. Restart R (if running inside RStudio).

If you call renv::init() with a project that is already using renv, it will attempt to do the right thing: it will restore the project library if it's missing, or otherwise ask you what to do.

Usage

```
init(
  project = NULL,
  ...,
  profile = NULL,
  settings = NULL,
  bare = FALSE,
  force = FALSE,
  repos = NULL,
  bioconductor = NULL,
```

```
  load = TRUE,
  restart = interactive()
)
```

Arguments

project	The project directory. When NULL (the default), the current working directory will be used. The R working directory will be changed to match the requested project directory.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
profile	The profile to be activated. See vignette("profiles", package = "renv") for more information. When NULL (the default), the profile is not changed. Use profile = "default" to revert to the default renv profile.
settings	A list of settings to be used with the newly-initialized project.
bare	Boolean; initialize the project with an empty project library, without attempting to discover and install R package dependencies?
force	Boolean; force initialization? By default, renv will refuse to initialize the home directory as a project, to defend against accidental misusages of init().
repos	The R repositories to be used in this project. See **Repositories** for more details.
bioconductor	The version of Bioconductor to be used with this project. Setting this may be appropriate if renv is unable to determine that your project depends on a package normally available from Bioconductor. Set this to TRUE to use the default version of Bioconductor recommended by the BiocManager package. When NULL (the default), the value is inferred from the bioconductor.init configuration option – see config for more details.
load	Boolean; should the project be loaded after it is initialized?
restart	Boolean; attempt to restart the R session after initializing the project? A session restart will be attempted if the "restart" R option is set by the frontend hosting R.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Repositories

If the default R repositories have not already been set, renv will use the Posit Public Package Manager CRAN mirror for package installation. The primary benefit to using this mirror is that it can provide pre-built binaries for R packages on a variety of commonly-used Linux distributions. This behavior can be configured or disabled if desired – see the options in config() for more details.

Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

`install` *Install packages*

Description

Install one or more R packages, from a variety of remote sources. `install()` uses the same machinery as `restore()` (i.e. it uses cached packages where possible) but it does not respect the lockfile, instead installing the latest versions available from CRAN.

See `vignette("package-install")` for more details.

Usage

```
install(
  packages = NULL,
  ...,
  include = NULL,
  exclude = NULL,
  library = NULL,
```

```
  type = NULL,
  rebuild = FALSE,
  repos = NULL,
  prompt = interactive(),
  dependencies = NULL,
  verbose = NULL,
  transactional = NULL,
  lock = FALSE,
  project = NULL
)
```

Arguments

packages	Either NULL (the default) to install all packages required by the project, or a character vector of packages to install. renv supports a subset of the remotes syntax used for package installation, e.g: • pkg: install latest version of pkg from CRAN. • pkg@version: install specified version of pkg from CRAN. • username/repo: install package from GitHub. • username/repo@ref: install from a specific git ref (branch, tag, or SHA). • username/repo:subdir: install from a subdirectory of a GitHub repo. • bioc::pkg: install pkg from Bioconductor. See https://remotes.r-lib.org/articles/dependencies.html and the examples below for more details. Note that renv deviates from the remotes spec in one important way: subdirectories are separated from the repository specification with a : rather than /. For example, to install from the subdir subdirectory of GitHub package username/repo, you would use: `renv::install("username/repo:subdir")`
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
include	Packages which should be installed. include can occasionally be useful when you'd like to call renv::install() with no arguments, but restrict package installation to only some subset of dependencies in the project.
exclude	Packages which should not be installed. exclude is useful when using renv::install() to install all dependencies in a project, except for a specific set of packages.
library	The R library to be used. When NULL, the active project library will be used instead.
type	The type of package to install ("source" or "binary"). Defaults to the value of getOption("pkgType").
rebuild	Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt.

repos — The repositories to use when restoring packages installed from CRAN or a CRAN-like repository. By default, the repositories recorded in the lockfile will be used, ensuring that (e.g.) CRAN packages are re-installed from the same CRAN mirror.

Use `repos = getOption("repos")` to override with the repositories set in the current session, or see the `repos.override` option in config for an alternate way override.

prompt — Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`.

dependencies — A vector of DESCRIPTION field names that should be used for package dependency resolution. When `NULL` (the default), the value of `renv::settings$package.dependency.fields` is used. The aliases "strong", "most", and "all" are also supported. See `tools::package_dependencies(` for more details.

verbose — Boolean; report output from `R CMD build` and `R CMD INSTALL` during installation? When `NULL` (the default), the value of `config$install.verbose()` will be used. When `FALSE`, installation output will be emitted only if a package fails to install.

transactional — Whether or not to use a 'transactional' package installation. See **Transactional Restore** in `restore()` for more details. When `NULL` (the default), the value of the `install.transactional` `config` option will be used.

lock — Boolean; update the `renv.lock` lockfile after the successful installation of the requested packages?

project — The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Value

A named list of package records which were installed by renv.

Remotes

`install()` (called without arguments) will respect the `Remotes` field of the `DESCRIPTION` file (if present). This allows you to specify places to install a package other than the latest version from CRAN. See https://remotes.r-lib.org/articles/dependencies.html for details.

Remotes can optionally be prefixed with a package name and `=`, to explicitly associate a package name with a remote specification. For example:

```
Remotes: mypkg=user/repo
```

This is most useful when the package name cannot be inferred from the remote (e.g. when the repository name differs from the package name).

Bioconductor

Packages from Bioconductor can be installed by using the `bioc::` prefix. For example,

```
renv::install("bioc::Biobase")
```

will install the latest-available version of Biobase from Bioconductor.

renv depends on BiocManager (or, for older versions of R, BiocInstaller) for the installation of packages from Bioconductor. If these packages are not available, renv will attempt to automatically install them before fulfilling the installation request.

Examples

```
## Not run:

# install the latest version of 'digest'
renv::install("digest")

# install an old version of 'digest' (using archives)
renv::install("digest@0.6.18")

# install 'digest' from GitHub (latest dev. version)
renv::install("eddelbuettel/digest")

# install a package from GitHub, using specific commit
renv::install("eddelbuettel/digest@df55b00bff33e945246eff2586717452e635032f")

# install a package from Bioconductor
# (note: requires the BiocManager package)
renv::install("bioc::Biobase")

# install a package from a subdirectory of a GitHub repo
renv::install("username/repo:subdir")

# install a package, specifying path explicitly
renv::install("~/path/to/package")

# install packages as declared in the project DESCRIPTION file
renv::install()


## End(Not run)
```

`isolate` *Isolate a project*

Description

Copy packages from the renv cache directly into the project library, so that the project can continue to function independently of the renv cache.

Usage

```
isolate(project = NULL)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

After calling isolate(), renv will still be able to use the cache on future install()s and restore()s. If you'd prefer that renv copy packages from the cache, rather than use symlinks, you can set the renv configuration option:

```
options(renv.config.cache.symlinks = FALSE)
```

to force renv to copy packages from the cache, as opposed to symlinking them. If you'd like to disable the cache altogether for a project, you can use:

```
settings$use.cache(FALSE)
```

to explicitly disable the cache for the project.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Examples

```
## Not run:

# isolate a project
renv::isolate()


## End(Not run)
```

---

load *Load a project*

Description

renv::load() sets the library paths to use a project-local library, sets up the system library sandbox, if needed, and creates shims for install.packages(), update.packages(), and remove.packages().

You should not generally need to call renv::load() yourself, as it's called automatically by the project auto-loader created by init()/ activate(). However, if needed, you can use renv::load("<project>") to explicitly load an renv project located at a particular path.

Usage

```
load(project = NULL, quiet = FALSE, profile = NULL, ...)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
quiet	Boolean; be quiet during load?
profile	The profile to be activated. See vignette("profiles", package = "renv") for more information. When NULL (the default), the profile is not changed. Use profile = "default" to revert to the default renv profile.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Shims

To help you take advantage of the package cache, renv places a couple of shims on the search path:

- install.packages() instead calls renv::install().
- remove.packages() instead calls renv::remove().
- update.packages() instead calls renv::update().

This allows you to keep using your existing muscle memory for installing, updating, and remove packages, while taking advantage of renv features like the package cache.

If you'd like to bypass these shims within an R session, you can explicitly call the version of these functions from the utils package, e.g. with utils::install.packages(<...>).

If you'd prefer not to use the renv shims at all, they can be disabled by setting the R option options(renv.config.shims.enabled = FALSE) or by setting the environment variable RENV_CONFIG_SHIMS_ENABLED = FALSE. See ?config for more details.

Examples

```
## Not run:

# load a project -- note that this is normally done automatically
# by the project's auto-loader, but calling this explicitly to
# load a particular project may be useful in some circumstances
renv::load()


## End(Not run)
```

`lockfile` *Create a lockfile*

Description

Create an `renv` lockfile from a variety of sources.

Usage

```
lockfile(from = NULL, to = NULL, project = NULL)
```

Arguments

`from`	The source from which the lockfile should be created. This can be: • `NULL` (the default), in which case the lockfile is created from the state of the active project's library; • The path to a Posit Connect `manifest.json` file, or a manifest that has already been read in as an R list, in which case the manifest is converted into a lockfile. The set of supported sources may be expanded in future releases of `renv`.
`to`	The path to a lockfile to be written. When `NULL` (the default), the lockfile is returned as an R object and not written to disk; otherwise, the lockfile is written to `to` and returned invisibly.
`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

`lockfile()` provides a generic entry point for producing an `renv` lockfile. The kind of lockfile produced depends on the value supplied as `from`; for example, given the path to a Posit Connect `manifest.json` file, `lockfile()` will convert that manifest into a lockfile.

By default the lockfile is returned as an R object. Supply `to` to also write it to disk, after which `restore()` can be used to restore the associated project library.

Value

An `renv` lockfile, as an R object of class `renv_lockfile`. When `to` is supplied, the lockfile is returned invisibly.

See Also

`lockfiles`, for a description of the structure of an `renv` lockfile.

Examples

```
## Not run:

# create a lockfile from a Posit Connect 'manifest.json' file
lock <- lockfile(from = "manifest.json")

# convert a 'manifest.json' file and write 'renv.lock' in a single call
lockfile(from = "manifest.json", to = "renv.lock")


## End(Not run)
```

`lockfiles` *Lockfiles*

Description

A **lockfile** records the state of a project at some point in time.

Usage

```
lockfile_create(
  type = settings$snapshot.type(project = project),
  libpaths = .libPaths(),
  packages = NULL,
  exclude = NULL,
  prompt = interactive(),
  force = FALSE,
  ...,
  project = NULL
)

lockfile_read(file = NULL, ..., project = NULL)

lockfile_write(lockfile, file = NULL, ..., project = NULL)

lockfile_modify(
  lockfile = NULL,
  ...,
  remotes = NULL,
  repos = NULL,
  project = NULL
)
```

Arguments

`type`	The type of snapshot to perform:

- `"implicit"`, (the default), uses all packages captured by `dependencies()`.
- `"explicit"` uses packages recorded in `DESCRIPTION`.
- `"all"` uses all packages in the project library.
- `"custom"` uses a custom filter.

See **Snapshot types** below for more details.

`libpaths` The library paths to be used when generating the lockfile.

`packages` A vector of packages to be included in the lockfile. When `NULL` (the default), all packages relevant for the type of snapshot being performed will be included. When set, the `type` argument is ignored. Recursive dependencies of the specified packages will be added to the lockfile as well.

`exclude` A vector of packages to be explicitly excluded from the lockfile. Note that transitive package dependencies will always be included, to avoid potentially creating an incomplete / non-functional lockfile.

`prompt` Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`.

`force` Boolean; force generation of a lockfile even when pre-flight validation checks have failed?

`...` Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.

`project` The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

`file` A file path, or R connection.

`lockfile` An `renv` lockfile; typically created by either `lockfile_create()` or `lockfile_read()`.

`remotes` An R vector of remote specifications.

`repos` A named vector, mapping R repository names to their URLs.

Details

A lockfile captures the state of a project's library at some point in time. In particular, the package names, their versions, and their sources (when known) are recorded in the lockfile.

Projects can be restored from a lockfile using the `restore()` function. This implies reinstalling packages into the project's private library, as encoded within the lockfile.

While lockfiles are normally generated and used with `snapshot()` / `restore()`, they can also be edited by hand if so desired. Lockfiles are written as `.json`, to allow for easy consumption by other tools.

An example lockfile follows:

```
{
  "R": {
    "Version": "3.6.1",
    "Repositories": [
      {
        "Name": "CRAN",
```

```
      "URL": "https://cloud.r-project.org"
    }
  ]
 },
 "Packages": {
  "markdown": {
    "Package": "markdown",
    "Version": "1.0",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "4584a57f565dd7987d59dda3a02cfb41"
  },
  "mime": {
    "Package": "mime",
    "Version": "0.7",
    "Source": "Repository",
    "Repository": "CRAN",
    "Hash": "908d95ccbfd1dd274073ef07a7c93934"
  }
 }
}
```

The sections used within a lockfile are described next.

renv:
Information about the version of renv used to manage this project.

Version	The version of the renv package used with this project.

R:
Properties related to the version of R associated with this project.

Version	The version of R used.
Repositories	The R repositories used in this project.

Packages:
R package records, capturing the packages used or required by a project at the time when the lockfile was generated.

Package	The package name.
Version	The package version.
Source	The location from which this package was retrieved.
Repository	The name of the repository (if any) from which this package was retrieved.
Hash	(Optional) A unique hash for this package, used for package caching.

Additional remote fields, further describing how the package can be retrieved from its corresponding source, will also be included as appropriate (e.g. for packages installed from GitHub).

Python:

Metadata related to the version of Python used with this project (if any).

Version	The version of Python being used.
Type	The type of Python environment being used ("virtualenv", "conda", "system")
Name	The (optional) name of the environment being used.

Note that the `Name` field may be empty. In that case, a project-local Python environment will be used instead (when not directly using a system copy of Python).

Caveats

These functions are primarily intended for expert users – in most cases, `snapshot()` and `restore()` are the primariy tools you will need when creating and using lockfiles.

See Also

Other reproducibility: `restore()`, `snapshot()`

`migrate` *Migrate a project from packrat to renv*

Description

Migrate a project's infrastructure from packrat to renv.

Usage

```
migrate(
  project = NULL,
  packrat = c("lockfile", "sources", "library", "options", "cache")
)
```

Arguments

`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.
`packrat`	Components of the Packrat project to migrate. See the default argument list for components of the Packrat project that can be migrated. Select a subset of those components for migration as appropriate.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Migration

When migrating Packrat projects to renv, the set of components migrated can be customized using the `packrat` argument. The set of components that can be migrated are as follows:

Name	Description
`lockfile`	Migrate the Packrat lockfile (`packrat/packrat.lock`) to the renv lockfile (`renv.lock`).
`sources`	Migrate package sources from the `packrat/src` folder to the renv sources folder. Currently, only CRAN package
`library`	Migrate installed packages from the Packrat library to the renv project library.
`options`	Migrate compatible Packrat options to the renv project.
`cache`	Migrate packages from the Packrat cache to the renv cache.

Examples

```
## Not run:

# migrate Packrat project infrastructure to renv
renv::migrate()


## End(Not run)
```

`modify` *Modify a Lockfile*

Description

Modify a project's lockfile, either interactively or non-interactively.

Usage

```
modify(project = NULL, changes = NULL)
```

Arguments

`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.
`changes`	A list of changes to be merged into the lockfile. When `NULL` (the default), the lockfile is instead opened for interactive editing.

Details

After edit, if the lockfile edited is associated with the active project, any state-related changes (e.g. to R repositories) will be updated in the current session.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

Examples

```
## Not run:

# modify an existing lockfile
if (interactive())
  renv::modify()


## End(Not run)
```

`paths` *Path for storing global state*

Description

By default, renv stores global state in the following OS-specific folders:

Platform	Location
Linux	`~/.cache/R/renv`
macOS	`~/Library/Caches/org.R-project.R/R/renv`
Windows	`%LOCALAPPDATA%/R/cache/R/renv`

If desired, this path can be customized by setting the `RENV_PATHS_ROOT` environment variable. This can be useful if you'd like, for example, multiple users to be able to share a single global cache.

Usage

```
paths
```

Customising individual paths

The various state sub-directories can also be individually adjusted, if so desired (e.g. you'd prefer to keep the cache of package installations on a separate volume). The various environment variables that can be set are enumerated below:

Environment Variable	Description
`RENV_PATHS_ROOT`	The root path used for global state storage.
`RENV_PATHS_LIBRARY`	The path to the project library.
`RENV_PATHS_LIBRARY_ROOT`	The parent path for project libraries.
`RENV_PATHS_LIBRARY_STAGING`	The parent path used for staged package installs.
`RENV_PATHS_SANDBOX`	The path to the sandboxed R system library.
`RENV_PATHS_LOCKFILE`	The path to the lockfile.
`RENV_PATHS_CELLAR`	The path to the cellar, containing local package binaries and sources.
`RENV_PATHS_SOURCE`	The path containing downloaded package sources.
`RENV_PATHS_BINARY`	The path containing downloaded package binaries.
`RENV_PATHS_CACHE`	The path containing cached package installations.

RENV_PATHS_PREFIX	An optional prefix to prepend to the constructed library / cache paths.
RENV_PATHS_RENV	The path to the project's renv folder. For advanced users only.
RENV_PATHS_RTOOLS	(Windows only) The path to Rtools.
RENV_PATHS_EXTSOFT	(Windows only) The path containing external software needed for compilation of Windows

(If you want these settings to persist in your project, it is recommended that you add these to an appropriate R startup file. For example, these could be set in: a project-local .Renviron, the user-level .Renviron, or a site-wide file at file.path(R.home("etc"), "Renviron.site"). See Startup for more details).

Note that renv will append platform-specific and version-specific entries to the set paths as appropriate. For example, if you have set:

```
Sys.setenv(RENV_PATHS_CACHE = "/mnt/shared/renv/cache")
```

then the directory used for the cache will still depend on the renv cache version (e.g. v2), the R version (e.g. 3.5) and the platform (e.g. x86_64-pc-linux-gnu). For example:

```
/mnt/shared/renv/cache/v2/R-3.5/x86_64-pc-linux-gnu
```

This ensures that you can set a single RENV_PATHS_CACHE environment variable globally without worry that it may cause collisions or errors if multiple versions of R needed to interact with the same cache.

If reproducibility of a project is desired on a particular machine, it is highly recommended that the renv cache of installed packages + binary packages is backed up and persisted, so that packages can be easily restored in the future – installation of packages from source can often be arduous.

Sharing state across operating systems

If you need to share the same cache with multiple different Linux operating systems, you may want to set the RENV_PATHS_PREFIX environment variable to help disambiguate the paths used on Linux. For example, setting RENV_PATHS_PREFIX = "ubuntu-bionic" would instruct renv to construct a cache path like:

```
/mnt/shared/renv/cache/v2/ubuntu-bionic/R-3.5/x86_64-pc-linux-gnu
```

If this is required, it's strongly recommended that this environment variable is set in your R installation's Renviron.site file, typically located at file.path(R.home("etc"), "Renviron.site"), so that it can be active for any R sessions launched on that machine.

Starting from renv 0.13.0, you can also instruct renv to auto-generate an OS-specific component to include as part of library and cache paths, by setting the environment variable:

```
RENV_PATHS_PREFIX_AUTO = TRUE
```

The prefix will be constructed based on fields within the system's /etc/os-release file. Note that this is the default behavior with renv 1.0.6 when using R 4.4.0 or later.

Package cellar

If your project depends on one or more R packages that are not available in any remote location, you can still provide a locally-available tarball for renv to use during restore. By default, these packages should be made available in the folder as specified by the RENV_PATHS_CELLAR environment variable. The package sources should be placed in a file at one of these locations:

- ${RENV_PATHS_CELLAR}/<package>_<version>.<ext>
- ${RENV_PATHS_CELLAR}/<package>/<package>_<version>.<ext>
- <project>/renv/cellar/<package>_<version>.<ext>
- <project>/renv/cellar/<package>/<package>_<version>.<ext>

where .<ext> is .tar.gz for source packages, or .tgz for binaries on macOS and .zip for binaries on Windows. During restore(), renv will search the cellar for a compatible package, and prefer installation with that copy of the package if appropriate.

Older versions

Older version of renv used a different default cache location. Those cache locations are:

Platform	Location
Linux	~/.local/share/renv
macOS	~/Library/Application Support/renv
Windows	%LOCALAPPDATA%/renv

If an renv root directory has already been created in one of the old locations, that will still be used. This change was made to comply with the CRAN policy requirements of R packages.

Examples

```
# get the path to the project library
path <- renv::paths$library()
```

---

plan *Plan package installations*

Description

renv::plan() resolves the packages required by a project and generates a lockfile, without installing any packages. This allows you to see what renv::restore() would do before actually performing the installation.

Usage

```
plan(
  packages = NULL,
  ...,
  dependencies = NULL,
  lockfile = paths$lockfile(project = project),
  project = NULL
)
```

Arguments

packages	The packages to be included in the plan. When NULL (the default), all packages currently used in the project will be included, as determined by dependencies().
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
dependencies	A vector of DESCRIPTION field names that should be used for package dependency resolution. When NULL (the default), the value of renv::settings$package.dependency.fields is used. The aliases "strong", "most", and "all" are also supported. See tools::package_dependencies(for more details.
lockfile	The path where the generated lockfile should be written. Use NULL to skip writing the lockfile.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

Because packages are not installed, the generated lockfile is based on package metadata from the configured repositories (i.e. the PACKAGES file). This means the lockfile may contain only a subset of the fields that would be present in a lockfile generated by snapshot(), which reads the full DESCRIPTION from installed packages. The generated lockfile is still suitable for use with restore().

Value

The generated lockfile, invisibly.

Examples

```
## Not run:

# plan the packages required by the current project
renv::plan()

# plan a specific set of packages
renv::plan(packages = c("dplyr", "ggplot2"))


## End(Not run)
```

project *Retrieve the active project*

Description

Retrieve the path to the active project (if any).

Usage

```
project(default = NULL)
```

Arguments

default	The value to return when no project is currently active. Defaults to NULL.

Value

The active project directory, as a length-one character vector.

Examples

```
## Not run:

# get the currently-active renv project
renv::project()


## End(Not run)
```

purge *Purge packages from the cache*

Description

Purge packages from the cache. This can be useful if a package which had previously been installed in the cache has become corrupted or unusable, and needs to be reinstalled.

Usage

```
purge(package, ..., version = NULL, hash = NULL, prompt = interactive())
```

Arguments

package	A single package to be removed from the cache.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
version	The package version to be removed. When NULL, all versions of the requested package will be removed.
hash	The specific hashes to be removed. When NULL, all hashes associated with a particular package's version will be removed.
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt.

Details

purge() is an inherently destructive option. It removes packages from the cache, and so any project which had symlinked that package into its own project library would find that package now unavailable. These projects would hence need to reinstall any purged packages. Take heed of this in case you're looking to purge the cache of a package which is difficult to install, or if the original sources for that package are no longer available!

Value

The set of packages removed from the renv global cache, as a character vector of file paths.

Examples

```
## Not run:

# remove all versions of 'digest' from the cache
renv::purge("digest")

# remove only a particular version of 'digest' from the cache
renv::purge("digest", version = "0.6.19")


## End(Not run)
```

---

rebuild *Rebuild the packages in your project library*

Description

Rebuild and reinstall packages in your library. This can be useful as a diagnostic tool – for example, if you find that one or more of your packages fail to load, and you want to ensure that you are starting from a clean slate.

Usage

```
rebuild(
  packages = NULL,
  recursive = TRUE,
  ...,
  type = NULL,
  prompt = interactive(),
  library = NULL,
  project = NULL
)
```

Arguments

packages	The package(s) to be rebuilt. When NULL, all packages in the library will be reinstalled.
recursive	Boolean; should dependencies of packages be rebuilt recursively? Defaults to TRUE.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
type	The type of package to install ("source" or "binary"). Defaults to the value of getOption("pkgType").
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt.
library	The R library to be used. When NULL, the active project library will be used instead.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Value

A named list of package records which were installed by renv.

Examples

```
## Not run:

# rebuild the 'dplyr' package + all of its dependencies
renv::rebuild("dplyr", recursive = TRUE)

# rebuild only 'dplyr'
renv::rebuild("dplyr", recursive = FALSE)


## End(Not run)
```

record *Update package records in a lockfile*

Description

Use record() to record a new entry within an existing renv lockfile.

Usage

```
record(records, lockfile = NULL, project = NULL, enrich = TRUE)
```

Arguments

records	A list of named records, mapping package names to a definition of their source. See **Records** for more details.
lockfile	Path to a lockfile. When NULL (the default), the renv.lock located in the root of the current project will be used.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
enrich	Should resolved records be enriched with the DESCRIPTION fields that snapshot() would write (Depends, Imports, Suggests, LinkingTo, License, etc.)? Defaults to TRUE. When TRUE, record() consults the active package repositories (and crandb, when enabled, for CRAN packages) to fill in the additional fields, erroring if the remote source is unreachable. The Hash field is not computed for enriched records. Additional DESCRIPTION-only fields (Title, Description, Author, Maintainer) are only included when the source can supply them – typically when crandb is reachable for CRAN packages, or when the source is a Git host (GitHub, GitLab, Bitbucket). Set to FALSE to keep the minimal record (Package, Version, Source, Repository) produced by remote resolution.

Details

This function can be useful when you need to change one or more of the package records within an renv lockfile – for example, because a recorded package cannot be restored in a particular environment, and you know of a suitable alternative.

Records

Records can be provided either using the **remotes** short-hand syntax, or by using an R list of entries to record within the lockfile. See ?lockfiles for more information on the structure of a package record.

A package's record can be removed from the lockfile by setting its entry to NULL; for example, use renv::record(list(dplyr = NULL)) to remove the record for the dplyr package from the lockfile. Note that this only modifies the lockfile; use remove() if you'd also like to remove the package from the project library.

Examples

```
## Not run:

# use digest 0.6.22 from package repositories -- different ways
# of specifying the remote. use whichever is most natural
renv::record("digest@0.6.22")
renv::record(list(digest = "0.6.22"))
renv::record(list(digest = "digest@0.6.22"))

# alternatively, provide a full record as a list
digest_record <- list(
  Package = "digest",
  Version = "0.6.22",
  Source  = "Repository",
  Repository = "CRAN"
)

renv::record(list(digest = digest_record))

# remove the record for digest from the lockfile
renv::record(list(digest = NULL))


## End(Not run)
```

---

refresh *Refresh the local cache of available packages*

Description

Query the active R package repositories for available packages, and update the in-memory cache of those packages.

Usage

```
refresh()
```

Details

Note that R also maintains its own on-disk cache of available packages, which is used by `available.packages()`. Calling `refresh()` will force an update of both types of caches. renv prefers using an in-memory cache as on occasion the temporary directory can be slow to access (e.g. when it is a mounted network filesystem).

Value

A list of package databases, invisibly – one for each repository currently active in the R session. Note that this function is normally called for its side effects.

Examples

```
## Not run:

# check available packages
db <- available.packages()

# wait some time (suppose packages are uploaded / changed in this time)
Sys.sleep(5)

# refresh the local available packages database
# (the old locally cached db will be removed)
db <- renv::refresh()


## End(Not run)
```

`rehash` *Re-hash packages in the renv cache*

Description

Re-hash packages in the renv cache, ensuring that any previously-cached packages are copied to a new cache location appropriate for this version of renv. This can be useful if the cache scheme has changed in a new version of renv, but you'd like to preserve your previously-cached packages.

Usage

```
rehash(prompt = interactive(), ...)
```

Arguments

`prompt`	Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`.
`...`	Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.

Details

When re-hashing, packages are relocated to the cache location appropriate for the active version of renv: they are copied when migrating from a previous cache version, and moved when re-hashing within the active cache. Because a package's previous cache location is not retained, project libraries that still link to that old location will be left with broken links after a re-hash. Use `repair()` (or, equivalently, `restore()`) within those projects to reinstall and re-link the affected packages.

remote *Resolve a Remote*

Description

Given a remote specification, resolve it into an renv package record that can be used for download and installation (e.g. with install).

Usage

```
remote(spec)
```

Arguments

spec	A remote specification. This should be a string, conforming to the Remotes specification as defined in https://remotes.r-lib.org/articles/dependencies.html.

remove *Remove packages*

Description

Remove (uninstall) R packages.

Usage

```
remove(packages, ..., library = NULL, prompt = interactive(), project = NULL)
```

Arguments

packages	A character vector of R packages to remove.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
library	The library from which packages should be removed. When NULL, the active library (that is, the first entry reported in .libPaths()) is used instead.
prompt	Boolean; prompt the user before removing packages? The prompt is only shown when removing packages from a library other than the project library.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

Note that remove() only removes packages from the requested library; it does not modify the project lockfile. Use record() if you'd like to remove a package's record from the lockfile.

Value

A vector of package records, describing the packages (if any) which were successfully removed.

Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

`repair` *Repair a project*

Description

Use `repair()` to recover from some common issues that can occur with a project. Currently, two operations are performed:

Usage

```
repair(library = NULL, lockfile = NULL, project = NULL)
```

Arguments

library	The R library to be used. When NULL, the active project library will be used instead.
lockfile	The path to a lockfile (if any). When available, renv will use the lockfile when attempting to infer the remote associated with the inaccessible version of each missing package. When NULL (the default), the project lockfile will be used.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

1. Packages with broken symlinks into the cache will be re-installed.
2. Packages that were installed from sources, but appear to be from an remote source (e.g. GitHub), will have their DESCRIPTION files updated to record that remote source explicitly.

---

restore *Restore project library from a lockfile*

Description

Restore a project's dependencies from a lockfile, as previously generated by snapshot().

Usage

```
restore(
  project = NULL,
  ...,
  library = NULL,
  lockfile = NULL,
  packages = NULL,
  exclude = NULL,
  rebuild = FALSE,
  repos = NULL,
  clean = FALSE,
  strict = FALSE,
  transactional = NULL,
  retry = NULL,
  prompt = interactive()
)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.

library	The library paths to be used during restore.
lockfile	Path to a lockfile. When NULL (the default), the renv.lock located in the root of the current project will be used.
packages	A subset of packages recorded in the lockfile to restore. When NULL (the default), all packages available in the lockfile will be restored. Any required recursive dependencies of the requested packages will be restored as well.
exclude	A subset of packages to be excluded during restore. This can be useful for when you'd like to restore all but a subset of packages from a lockfile. Note that if you attempt to exclude a package which is required as the recursive dependency of another package, your request will be ignored.
rebuild	Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt.
repos	The repositories to use when restoring packages installed from CRAN or a CRAN-like repository. By default, the repositories recorded in the lockfile will be used, ensuring that (e.g.) CRAN packages are re-installed from the same CRAN mirror. Use repos = getOption("repos") to override with the repositories set in the current session, or see the repos.override option in config for an alternate way override.
clean	Boolean; remove packages not recorded in the lockfile from the target library? Use clean = TRUE if you'd like the library state to exactly reflect the lockfile contents after restore().
strict	Boolean; when TRUE, packages whose lockfile records contain a repository URL in the Repository field will only be retrieved from that exact repository. If the recorded repository is unavailable, the restore will fail rather than falling back to other configured repositories. This can be useful when you need to ensure that packages are installed from a specific source (e.g. an internal package repository).
transactional	Whether or not to use a 'transactional' restore. See **Transactional Restore** for more details. When NULL (the default), the value of the install.transactional config option will be used.
retry	Whether to retry packages that fail to install by attempting to install the latest available versions instead. This is helpful when restoring after an R upgrade, where the older versions recorded in the lockfile may no longer install successfully. When NULL (the default), renv will prompt in interactive sessions and otherwise leave the failures unresolved. Use retry = TRUE to retry without prompting (e.g. in CI), or retry = FALSE to disable the retry entirely.
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt.

Details

renv::restore() compares packages recorded in the lockfile to the packages installed in the project library. Where there are differences it resolves them by installing the lockfile-recorded

package into the project library. If `clean = TRUE`, `restore()` will additionally delete any packages in the project library that don't appear in the lockfile.

Value

A named list of package records which were installed by renv.

Transactional Restore

By default, `renv::restore()` will perform a 'transactional' restore, wherein the project library is mutated only if all packages within the lockfile are successfully restored. The intention here is to prevent the private library from entering an inconsistent state, if some subset of packages were to install successfully but some other subset of packages did not. `renv::restore(transactional = FALSE)` can be useful if you're attempting to restore packages from a lockfile, but would like to update or change certain packages piece-meal if they fail to install.

The term 'transactional' here borrows from the parlance of a 'database transaction', where the failure of any intermediate step implies that the whole transaction will be rolled back, so that the state of the database before the transaction was initiated can be preserved. See https://en.wikipedia.org/wiki/Database_transaction for more details.

See Also

Other reproducibility: `lockfiles`, `snapshot()`

Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
```

```
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

`retrieve` *Retrieve packages*

Description

Retrieve (download) one or more packages from external sources. Using `renv::retrieve()` can be useful in CI / CD workflows, where you might want to download all packages listed in a lockfile before later invoking `restore()`. Packages will be downloaded to an internal path within `renv`'s local state directories – see paths for more details.

Usage

```
retrieve(packages = NULL, ..., lockfile = NULL, destdir = NULL, project = NULL)
```

Arguments

`packages`	Either `NULL` (the default) to install all packages required by the project, or a character vector of packages to install. renv supports a subset of the remotes syntax used for package installation, e.g: • `pkg`: install latest version of `pkg` from CRAN. • `pkg@version`: install specified version of `pkg` from CRAN. • `username/repo`: install package from GitHub. • `username/repo@ref`: install from a specific git ref (branch, tag, or SHA). • `username/repo:subdir`: install from a subdirectory of a GitHub repo. • `bioc::pkg`: install `pkg` from Bioconductor. See `https://remotes.r-lib.org/articles/dependencies.html` and the examples below for more details. Note that renv deviates from the remotes spec in one important way: subdirectories are separated from the repository specification with a `:` rather than `/`. For example, to install from the `subdir` subdirectory of GitHub package `username/repo`, you would use: `renv::install("username/repo:subdir")`
`...`	Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.
`lockfile`	The path to an `renv` lockfile. When set, `renv` will retrieve the packages as defined within that lockfile. If `packages` is also non-`NULL`, then only those packages will be retrieved.
`destdir`	The directory where packages should be downloaded. When `NULL` (the default), the default internal storage locations (normally used by e.g. `install()` or `restore()`) will be used.
`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

If `destdir` is `NULL` and the requested package is already available within the `renv` cache, `renv` will return the path to that package directory in the cache.

Value

A named vector, mapping package names to the paths where those packages were downloaded.

Examples

```
## Not run:

# retrieve package + versions as defined in the lockfile
# normally used as a pre-flight step to renv::restore()
renv::retrieve()

# download one or more packages locally
renv::retrieve("rlang", destdir = ".")


## End(Not run)
```

---

run *Run a script*

Description

Run an R script, in the context of a project using renv. The script will be run within an R subprocess.

Usage

```
run(script, ..., job = NULL, name = NULL, args = NULL, project = NULL)
```

Arguments

`script`	The path to an R script.
`...`	Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.
`job`	Run the requested script as an RStudio job? Requires a recent version of both RStudio and the rstudioapi packages. When `NULL`, the script will be run as a job if possible, and as a regular R process launched by `system2()` if not.
`name`	The name to associate with the job, for scripts run as a job.
`args`	description A character vector of command line arguments to be passed to the launched job. These parameters can be accessed via `commandArgs(trailingOnly = FALSE)`.
`project`	The path to the renv project. This project will be loaded before the requested script is executed. When `NULL` (the default), renv will automatically determine the project root for the associated script if possible.

Value

The project directory, invisibly. Note that this function is normally called for its side effects.

---

sandbox *The default library sandbox*

Description

An R installation can have up to three types of library paths available to the user:

- The *user library*, where R packages downloaded and installed by the current user are installed. This library path is only visible to that specific user.
- The *site library*, where R packages maintained by administrators of a system are installed. This library path, if it exists, is visible to all users on the system.
- The *default library*, where R packages distributed with R itself are installed. This library path is visible to all users on the system.

Normally, only so-called "base" and "recommended" packages should be installed in the default library. (You can get a list of these packages with `installed.packages(priority = c("base", "recommended"))`). However, it is possible for users and administrators to install packages into the default library, if the filesystem permissions permit them to do so. (This, for example, is the default behavior on macOS.)

Because the site and default libraries are visible to all users, having those accessible in renv projects can potentially break isolation – that is, if a package were updated in the default library, that update would be visible to all R projects on the system.

To help defend against this, renv uses something called the "sandbox" to isolate renv projects from non-"base" packages that are installed into the default library. When an renv project is loaded, renv will:

- Create a new, empty library path (called the "sandbox"),
- Link only the "base" and "recommended" packages from the default library into the sandbox,
- Mark the sandbox as read-only, so that users are unable to install packages into this library,
- Instruct the R session to use the "sandbox" as the default library.

This process is mostly transparent to the user. However, because the sandbox is read-only, if you later need to remove the sandbox, you'll need to reset file permissions manually; for example, with `renv::sandbox$unlock()`.

If you'd prefer to keep the sandbox unlocked, you can also set:

```
RENV_SANDBOX_LOCKING_ENABLED = FALSE
```

in an appropriate startup `.Renviron` or `Renviron.site` file.

The sandbox can also be disabled entirely with:

```
RENV_CONFIG_SANDBOX_ENABLED = FALSE
```

The sandbox library path can also be configured using the `RENV_PATHS_SANDBOX` environment variable: see paths for more details.

Usage

```
sandbox
```

`scaffold` *Generate project infrastructure*

Description

Create the renv project infrastructure. This will:

- Create a project library, `renv/library`.
- Install renv into the project library.
- Update the project `.Rprofile` to call `source("renv/activate.R")` so that renv is automatically loaded for new R sessions launched in this project.
- Create `renv/.gitignore`, which tells git to ignore the project library.
- Create `.Rbuildignore`, if the project is also a package. This tells `R CMD build` to ignore the renv infrastructure,
- Write a (bare) lockfile, `renv.lock`.

Usage

```
scaffold(
  project = NULL,
  version = NULL,
  repos = getOption("repos"),
  settings = NULL
)
```

Arguments

`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.
`version`	The version of renv to associate with this project. By default, the version of renv currently installed is used.
`repos`	The R repositories to associate with this project.
`settings`	A list of renv settings, to be applied to the project after creation. These should map setting names to the desired values. See settings for more details.

Examples

```
## Not run:
# create scaffolding with 'devtools' ignored
renv::scaffold(settings = list(ignored.packages = "devtools"))

## End(Not run)
```

`settings` *Project settings*

Description

Define project-local settings that can be used to adjust the behavior of renv with your particular project.

- Get the current value of a setting with (e.g.) `settings$snapshot.type()`
- Set current value of a setting with (e.g.) `settings$snapshot.type("explicit")`.

Settings are automatically persisted across project sessions by writing to `renv/settings.json`. You can also edit this file by hand, but you'll need to restart the session for those changes to take effect.

`bioconductor.enabled`**:**

Should renv use Bioconductor with this project? When enabled (the default), renv will infer that packages tagged with a `biocViews` field (and stamped with Bioconductor provenance) come from Bioconductor, activate the Bioconductor repositories when required, and record the Bioconductor release in the lockfile. Set this to `FALSE` for projects that should be treated as repository-only – for example, when all packages (including any that happen to carry a `biocViews` field) are served from a single CRAN-like repository such as a Posit Package Manager "R repository". When disabled, renv will never infer a Bioconductor source, activate Bioconductor repositories, or write a Bioconductor entry into the lockfile.

`bioconductor.version`**:**

The Bioconductor version to be used with this project. Use this if you'd like to lock the version of Bioconductor used on a per-project basis. When unset, renv will try to infer the appropriate Bioconductor release using the BiocVersion package if installed; if not, renv uses `BiocManager::version()` to infer the appropriate Bioconductor version.

`external.libraries`**:**

A vector of library paths, to be used in addition to the project's own private library. This can be useful if you have a package available for use in some system library, but for some reason renv is not able to install that package (e.g. sources or binaries for that package are not publicly available, or you have been unable to orchestrate the pre-requisites for installing some packages from source on your machine).

`lockfile.sanitize`**:**

Should renv sanitize repository URLs when writing the lockfile? When `TRUE`, embedded credentials are stripped from URLs (e.g. `https://user:token@host/path` becomes `https://host/path`) to prevent accidental credential leakage. Set this to `FALSE` if your repository URLs contain credentials required for restore to function correctly.

`ignored.packages`**:**

A vector of packages, which should be ignored when attempting to snapshot the project's private library. Note that if a package has already been added to the lockfile, that entry in the lockfile will not be ignored.

`package.dependency.fields`:

When installing a package with `install()`, what `DESCRIPTION` fields should be used to determine that package's dependencies? The default uses `c("Imports", "Depends", "LinkingTo")`, but if you also want to install `Suggests` dependencies for a package, you can set this to `c("Imports", "Depends", "LinkingTo", "Suggests")`.

`ppm.enabled`:

Enable Posit Package Manager integration in this project? When `TRUE`, renv will attempt to transform repository URLs used by PPM into binary URLs as appropriate for the current Linux platform. Set this to `FALSE` if you'd like to continue using source-only PPM URLs, or if you find that renv is improperly transforming your repository URLs. You can still set and use PPM repositories with this option disabled; it only controls whether renv tries to transform source repository URLs into binary URLs on your behalf.

`ppm.ignored.urls`:

When Posit Package Manager integration is enabled, `renv` will attempt to transform source repository URLs into binary repository URLs. This setting can be used if you'd like to avoid this transformation with some subset of repository URLs.

`r.version`:

The version of R to encode within the lockfile. This can be set as a project-specific option if you'd like to allow multiple users to use the same renv project with different versions of R. renv will still warn the user if the major + minor version of R used in a project does not match what is encoded in the lockfile.

`snapshot.type`:

The type of snapshot to perform by default. See snapshot for more details.

`snapshot.dev`:

Whether to include development dependencies by default when calling `renv::snapshot()` or `renv::status()`. When `TRUE`, development dependencies (e.g., packages listed in `Suggests` or development tools like `devtools`) will be included. Defaults to `FALSE`.

`use.cache`:

Enable the renv package cache with this project. When active, renv will install packages into a global cache, and link packages from the cache into your renv projects as appropriate. This can greatly save on disk space and install time when for R packages which are used across multiple projects in the same environment.

`vcs.manage.ignores`:

Should renv attempt to manage the version control system's ignore files (e.g. `.gitignore`) within this project? Set this to `FALSE` if you'd prefer to take control. Note that if this setting is enabled, you will need to manually ensure internal data in the project's `renv/` folder is explicitly ignored.

`vcs.ignore.cellar`:

Set whether packages within a project-local package cellar are excluded from version control. See `vignette("package-sources", package = "renv")` for more information.

`vcs.ignore.library`:

Set whether the renv project library is excluded from version control.

`vcs.ignore.local`:

Set whether renv project-specific local sources are excluded from version control.

Usage

```
settings
```

Value

A named list of renv settings.

Defaults

You can change the default values of these settings for newly-created renv projects by setting R options for `renv.settings` or `renv.settings.<name>`. For example:

```
options(renv.settings = list(snapshot.type = "all"))
options(renv.settings.snapshot.type = "all")
```

If both of the `renv.settings` and `renv.settings.<name>` options are set for a particular key, the option associated with `renv.settings.<name>` is used instead. We recommend setting these in an appropriate startup profile, e.g. `~/.Rprofile` or similar.

Examples

```
## Not run:

# view currently-ignored packaged
renv::settings$ignored.packages()

# ignore a set of packages
renv::settings$ignored.packages("devtools", persist = FALSE)


## End(Not run)
```

---

snapshot *Record current state of the project library in the lockfile*

Description

Call `renv::snapshot()` to update a lockfile with the current state of dependencies in the project library. The lockfile can be used to later restore these dependencies as required.

It's also possible to call `renv::snapshot()` with a non-renv project, in which case it will record the current state of dependencies in the current library paths. This makes it possible to restore the current packages, providing lightweight portability and reproducibility without isolation.

If you want to automatically snapshot after each change, you can set `config$config$auto.snapshot(TRUE)` – see `?config` for more details.

Usage

```
snapshot(
  project = NULL,
  ...,
  library = NULL,
  lockfile = paths$lockfile(project = project),
  type = settings$snapshot.type(project = project),
  dev = NULL,
  repos = getOption("repos"),
  packages = NULL,
  exclude = NULL,
  prompt = interactive(),
  update = FALSE,
  force = FALSE,
  reprex = FALSE,
  description = NULL
)
```

Arguments

`project`	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.
`...`	Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.
`library`	The R libraries to snapshot. When `NULL`, the active R libraries (as reported by `.libPaths()`) are used.
`lockfile`	The location where the generated lockfile should be written. By default, the lockfile is written to a file called `renv.lock` in the project directory. When `NULL`, the lockfile (as an R object) is returned directly instead.
`type`	The type of snapshot to perform: • `"implicit"`, (the default), uses all packages captured by `dependencies()`. • `"explicit"` uses packages recorded in `DESCRIPTION`. • `"all"` uses all packages in the project library. • `"custom"` uses a custom filter. See **Snapshot types** below for more details.

dev	Boolean; include development dependencies? These packages are typically required when developing the project, but not when running it (i.e. you want them installed when humans are working on the project but not when computers are deploying it). Development dependencies include packages listed in the Suggests field of a DESCRIPTION found in the project root, and roxygen2 or devtools if their use is implied by other project metadata. They also include packages used in ~/.Rprofile if config$user.profile() is TRUE.
repos	The R repositories to be recorded in the lockfile. Defaults to the currently active package repositories, as retrieved by getOption("repos").
packages	A vector of packages to be included in the lockfile. When NULL (the default), all packages relevant for the type of snapshot being performed will be included. When set, the type argument is ignored. Recursive dependencies of the specified packages will be added to the lockfile as well.
exclude	A vector of packages to be explicitly excluded from the lockfile. Note that transitive package dependencies will always be included, to avoid potentially creating an incomplete / non-functional lockfile.
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, confirm is accepted as an alias for prompt.
update	Boolean; if the lockfile already exists, then attempt to update that lockfile without removing any prior package records.
force	Boolean; force generation of a lockfile even when pre-flight validation checks have failed?
reprex	Boolean; generate output appropriate for embedding the lockfile as part of a reprex?
description	A package DESCRIPTION, provided either as a named list of DESCRIPTION fields, or a path to a package directory or DESCRIPTION file. When set, all other arguments are ignored, and a single lockfile record for that package is returned.

Value

The generated lockfile, as an R object (invisibly). Note that this function is normally called for its side effects.

When description is provided, a single lockfile record (a named list) is returned instead.

Snapshot types

Depending on how you prefer to manage your R package dependencies, you may want to enable an alternate snapshot type.. The types available are as follows:

"implicit" (The default) Capture only packages which appear to be used in your project, as determined by renv::dependencies(). This ensures that only the packages actually required by your project will enter the lockfile; the downside if it might be slow if your project contains a large number of files. If speed becomes an issue, you might consider using .renvignore files to limit which files renv uses for dependency discovery, or switching to explicit mode, as described next.

`"explicit"` Only capture packages which are explicitly listed in the project `DESCRIPTION` file. This workflow is recommended for users who wish to manage their project's R package dependencies directly, and can be used for both package and non-package R projects. Packages used in this manner should be recorded in either the `Depends` or `Imports` field of the `DESCRIPTION` file.

`"all"` Capture all packages within the active R libraries in the lockfile. This is the quickest and simplest method, but may lead to undesired packages (e.g. development dependencies) entering the lockfile.

`"custom"` Like `"implicit"`, but use a custom user-defined filter instead. The filter should be specified by the R option `renv.snapshot.filter`, and should either be a character vector naming a function (e.g. `"package::method"`), or be a function itself. The function should only accept one argument (the project directory), and should return a vector of package names to include in the lockfile.

You can change the snapshot type for the current project with `settings()`. For example, the following code will switch to using `"explicit"` snapshots:

```
renv::settings$snapshot.type("explicit")
```

When the `packages` argument is set, `type` is ignored, and instead only the requested set of packages, and their recursive dependencies, will be written to the lockfile.

Package projects

When snapshotting a package project (that is, a project containing a `DESCRIPTION` file), renv normally records only the package's dependencies in the lockfile, and excludes the package itself. If you'd like the package itself to be recorded as well – for example, because the project is deployed as a Shiny application, using an `app.R` that runs an application exported from the package – you can set:

```
Config/renv/snapshot/include-self: TRUE
```

in the package's `DESCRIPTION` file. With this set, the package is also treated as a dependency of the project, and so is expected to be installed in the project library. Note that for the package to be restorable, it should be installed from a remote source – for example, with `renv::install("<user>/<repo>")` for a package available on GitHub.

The R option `renv.snapshot.ignore.self` can also be used to control this behavior: set it to `FALSE` to include the package in the lockfile, or `TRUE` to exclude it. When set, this option takes precedence over the `DESCRIPTION` field.

See Also

More on handling package `dependencies()`

Other reproducibility: `lockfiles`, `restore()`

Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

`status` *Report inconsistencies between lockfile, library, and dependencies*

Description

`renv::status()` reports issues caused by inconsistencies across the project lockfile, library, and `dependencies()`. In general, you should strive to ensure that `status()` reports no issues, as this maximizes your chances of successfully `restore()`ing the project in the future or on another machine.

`renv::load()` will report if any issues are detected when starting an renv project; we recommend resolving these issues before doing any further work on your project.

See the headings below for specific advice on resolving any issues revealed by `status()`.

Usage

```
status(
  project = NULL,
```

```
  ...,
  library = NULL,
  lockfile = NULL,
  sources = TRUE,
  cache = FALSE,
  dev = NULL
)
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
library	The library paths. By default, the library paths associated with the requested project are used.
lockfile	Path to a lockfile. When NULL (the default), the renv.lock located in the root of the current project will be used.
sources	Boolean; check that each of the recorded packages have a known installation source? If a package has an unknown source, renv may be unable to restore it.
cache	Boolean; perform diagnostics on the global package cache? When TRUE, renv will validate that the packages installed into the cache are installed at the expected + proper locations, and validate the hashes used for those storage locations.
dev	Boolean; include development dependencies? These packages are typically required when developing the project, but not when running it (i.e. you want them installed when humans are working on the project but not when computers are deploying it). Development dependencies include packages listed in the Suggests field of a DESCRIPTION found in the project root, and roxygen2 or devtools if their use is implied by other project metadata. They also include packages used in ~/.Rprofile if config$user.profile() is TRUE.

Value

This function is normally called for its side effects, but it invisibly returns a list containing the following components:

- library: packages in your library.
- lockfile: packages in the lockfile.
- synchronized: are the library and lockfile in sync?

Missing packages

status() first checks that all packages used by the project are installed. This must be done first because if any packages are missing we can't tell for sure that a package isn't used; it might be a

dependency that we don't know about. Once you have resolve any installation issues, you'll need to run `status()` again to reveal the next set of potential problems.

There are four possibilities for an uninstalled package:

- If it's used and recorded, call `renv::restore()` to install the version specified in the lockfile.
- If it's used and not recorded, call `renv::install()` to install it from CRAN or elsewhere.
- If it's not used and recorded, call `renv::snapshot()` to remove it from the lockfile.
- If it's not used and not recorded, there's nothing to do. This the most common state because you only use a small fraction of all available packages in any one project.

If you have multiple packages in an inconsistent state, we recommend `renv::restore()`, then `renv::install()`, then `renv::snapshot()`, but that also suggests you should be running status more frequently.

Lockfile vs `dependencies()`

Next we need to ensure that packages are recorded in the lockfile if and only if they are used by the project. Fixing issues of this nature only requires calling `snapshot()` because there are four possibilities for a package:

- If it's used and recorded, it's ok.
- If it's used and not recorded, call `renv::snapshot()` to add it to the lockfile.
- If it's not used but is recorded, call `renv::snapshot()` to remove it from the lockfile.
- If it's not used and not recorded, it's also ok, as it may be a development dependency.

Out-of-sync sources

The final issue to resolve is any inconsistencies between the version of the package recorded in the lockfile and the version installed in your library. To fix these issues you'll need to either call `renv::restore()` or `renv::snapshot()`:

- Call `renv::snapshot()` if your project code is working. This implies that the library is correct and you need to update your lockfile.
- Call `renv::restore()` if your project code isn't working. This probably implies that you have the wrong package versions installed and you need to restore from known good state in the lockfile.

If you're not sure which case applies, it's generally safer to call `renv::snapshot()`. If you want to rollback to an earlier known good status, see `history()` and `revert()`.

Different R Version

renv will also notify you if the version of R used when the lockfile was generated, and the version of R currently in use, do not match. In this scenario, you'll need to consider:

- Is the version of R recorded in the lockfile correct? If so, you'll want to ensure that version of R is installed and used when working in this project.
- Otherwise, you can call `renv::snapshot()` to update the version of R recorded in the lockfile, to match the version of R currently in use.

If you'd like to set the version of R recorded in a lockfile independently of the version of R currently in use, you can set the `r.version` project setting – see settings for more details.

Examples

```
## Not run:

# disable automatic snapshots
auto.snapshot <- getOption("renv.config.auto.snapshot")
options(renv.config.auto.snapshot = FALSE)

# initialize a new project (with an empty R library)
renv::init(bare = TRUE)

# install digest 0.6.19
renv::install("digest@0.6.19")

# save library state to lockfile
renv::snapshot()

# remove digest from library
renv::remove("digest")

# check library status
renv::status()

# restore lockfile, thereby reinstalling digest 0.6.19
renv::restore()

# restore automatic snapshots
options(renv.config.auto.snapshot = auto.snapshot)


## End(Not run)
```

`sysreqs` *R System Requirements*

Description

Compute the system requirements (system libraries; operating system packages) required by a set of R packages.

Usage

```
sysreqs(
  packages = NULL,
  ...,
  local = FALSE,
  check = NULL,
```

```
  report = TRUE,
  distro = NULL,
  collapse = FALSE,
  project = NULL
)
```

Arguments

packages	A vector of R package names. When NULL (the default), the project's package dependencies as reported via dependencies() are used.
...	Unused arguments, reserved for future expansion. If any arguments are matched to ..., renv will signal an error.
local	Boolean; should renv rely on locally-installed copies of packages when resolving system requirements? When FALSE, renv will use https://crandb.r-pkg.org to resolve the system requirements for these packages.
check	Boolean; should renv also check whether the requires system packages appear to be installed on the current system? Ignored when distro is supplied.
report	Boolean; should renv also report the commands which could be used to install all of the requisite package dependencies?
distro	The name of the Linux distribution for which system requirements should be checked – typical values are "ubuntu", "debian", and "redhat". These should match the distribution names used by the R system requirements database. A version suffix can be included; for example, "ubuntu:24.04".
collapse	Boolean; when reporting which packages need to be installed, should the report be collapsed into a single installation command? When FALSE (the default), a separate installation line is printed for each required system package.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

This function relies on the database of package system requirements maintained by Posit at https://github.com/rstudio/r-system-requirements, as well as the "meta-CRAN" service at https://crandb.r-pkg.org. This service primarily exists to map the (free-form) SystemRequirements field used by R packages to the system packages made available by a particular operating system.

As an example, the curl R package depends on the libcurl system library, and declares this with a SystemRequirements field of the form:

- libcurl (>= 7.62): libcurl-devel (rpm) or libcurl4-openssl-dev (deb)

This dependency can be satisfied with the following command line invocations on different systems:

- Debian: `sudo apt install libcurl4-openssl-dev`
- Redhat: `sudo dnf install libcurl-devel`

and so `sysreqs("curl")` would help provide the name of the package whose installation would satisfy the libcurl dependency.

Examples

```
## Not run:

# report the required system packages for this system
sysreqs()

# report the required system packages for a specific OS
sysreqs(platform = "ubuntu")


## End(Not run)
```

`update` *Update packages*

Description

Update packages which are currently out-of-date. Currently supports CRAN, Bioconductor, other CRAN-like repositories, GitHub, GitLab, Git, and BitBucket.

Updates will only be checked from the same source – for example, if a package was installed from GitHub, but a newer version is available on CRAN, that updated version will not be seen.

Usage

```
update(
  packages = NULL,
  ...,
  exclude = NULL,
  library = NULL,
  type = NULL,
  rebuild = FALSE,
  check = FALSE,
  prompt = interactive(),
  lock = FALSE,
  all = FALSE,
  project = NULL
)
```

Arguments

`packages`	A character vector of R packages to update. When `NULL` (the default), all packages (apart from any listed in the `ignored.packages` project setting) will be updated.
`...`	Unused arguments, reserved for future expansion. If any arguments are matched to `...`, renv will signal an error.

exclude	A set of packages to explicitly exclude from updating. Use `renv::update(exclude = <...>)` to update all packages except for a specific set of excluded packages.
library	The R library to be used. When `NULL`, the active project library will be used instead.
type	The type of package to install ("source" or "binary"). Defaults to the value of `getOption("pkgType")`.
rebuild	Force packages to be rebuilt, thereby bypassing any installed versions of the package available in the cache? This can either be a boolean (indicating that all installed packages should be rebuilt), or a vector of package names indicating which packages should be rebuilt.
check	Boolean; check for package updates without actually installing available updates? This is useful when you'd like to determine what updates are available, without actually installing those updates.
prompt	Boolean; prompt the user before taking any action? For backwards compatibility, `confirm` is accepted as an alias for `prompt`.
lock	Boolean; update the `renv.lock` lockfile after the successful installation of the requested packages?
all	Boolean; should `renv` check all library paths for out-of-date packages? When `FALSE` (the default), only the project library will be checked for out-of-date packages.
project	The project directory. If `NULL`, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Value

A named list of package records which were installed by renv.

Examples

```
## Not run:

# update the 'dplyr' package
renv::update("dplyr")


## End(Not run)
```

`upgrade` *Upgrade renv*

Description

Upgrade the version of renv associated with a project, including using a development version from GitHub. Automatically snapshots the updated renv, updates the activate script, and restarts R.

If you want to update all packages (including renv) to their latest CRAN versions, use `update()`.

Usage

```
upgrade(project = NULL, version = NULL, reload = NULL, prompt = interactive())
```

Arguments

project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.
version	The version of renv to be installed. When NULL (the default), the latest version of renv will be installed as available from CRAN (or whatever active package repositories are active) Alternatively, you can install the latest development version with "main", or a specific commit with a SHA, e.g. "5049cef8a".
reload	Boolean; reload renv after install? When NULL (the default), renv will be reloaded only if updating renv for the active project. Since it's not possible to guarantee a clean reload in the current session, this will attempt to restart your R session.
prompt	Boolean; prompt upgrade before proceeding?

Value

A boolean value, indicating whether the requested version of renv was successfully installed. Note that this function is normally called for its side effects.

Note

upgrade() is expected to work for renv versions >= 1.0.1. To upgrade from prior versions of renv, users should

```
renv::deactivate(); install.packages("renv"); renv::activate(); renv::record("renv")
```

Examples

```
## Not run:

# upgrade to the latest version of renv
renv::upgrade()

# upgrade to the latest version of renv on GitHub (development version)
renv::upgrade(version = "main")


## End(Not run)
```

use_python *Use python*

Description

Associate a version of Python with your project.

Usage

```
use_python(
  python = NULL,
  ...,
  type = c("auto", "virtualenv", "conda", "system"),
  name = NULL,
  project = NULL
)
```

Arguments

python	The path to the version of Python to be used with this project. See **Finding Python** for more details. Alternatively, set python = FALSE to deactivate Python integration for the project – this removes the Python section from renv.lock.
...	Optional arguments; currently unused.
type	The type of Python environment to use. When "auto" (the default), virtual environments will be used.
name	The name or path that should be used for the associated Python environment. If NULL and python points to a Python executable living within a pre-existing virtual environment, that environment will be used. Otherwise, a project-local environment will be created instead, using a name generated from the associated version of Python.
project	The project directory. If NULL, then the active project will be used. If no project is currently active, then the current working directory is used instead.

Details

When Python integration is active, renv will:

- Save metadata about the requested version of Python in renv.lock – in particular, the Python version, and the Python type ("virtualenv", "conda", "system"),
- Capture the set of installed Python packages during renv::snapshot(),
- Re-install the set of recorded Python packages during renv::restore().

In addition, when the project is loaded, the following actions will be taken:

- The RENV_PYTHON environment variable will be set, indicating the version of Python currently active for this sessions,

- The RETICULATE_PYTHON environment variable will be set, so that the reticulate package can automatically use the requested copy of Python as appropriate,
- The requested version of Python will be placed on the PATH, so that attempts to invoke Python will resolve to the expected version of Python.

You can override the version of Python used in a particular project by setting the RENV_PYTHON environment variable; e.g. as part of the project's .Renviron file. This can be useful if you find that renv is unable to automatically discover a compatible version of Python to be used in the project.

Value

TRUE, indicating that the requested version of Python has been successfully activated. Note that this function is normally called for its side effects.

Finding Python

In interactive sessions, when python = NULL, renv will prompt for an appropriate version of Python. renv will search a pre-defined set of locations when attempting to find Python installations on the system:

- getOption("renv.python.root"),
- /opt/python,
- /opt/local/python,
- ~/opt/python,
- /usr/local/opt (for macOS Homebrew-installed copies of Python),
- /opt/homebrew/opt (for M1 macOS Homebrew-installed copies of Python),
- ~/.pyenv/versions,
- Python instances available on the PATH.

In non-interactive sessions, renv will first check the RETICULATE_PYTHON environment variable; if that is unset, renv will look for Python on the PATH. It is recommended that the version of Python to be used is explicitly supplied for non-interactive usages of use_python().

Warning

We strongly recommend using Python virtual environments, for a few reasons:

1. If something goes wrong with a local virtual environment, you can safely delete that virtual environment, and then re-initialize it later, without worry that doing so might impact other software on your system.
2. If you choose to use a "system" installation of Python, then any packages you install or upgrade will be visible to any other application that wants to use that same Python installation. Using a virtual environment ensures that any changes made are isolated to that environment only.
3. Choosing to use Anaconda will likely invite extra frustration in the future, as you may be required to upgrade and manage your Anaconda installation as new versions of Anaconda are released. In addition, Anaconda installations tend to work poorly with software not specifically installed as part of that same Anaconda installation.

In other words, we recommend selecting "system" or "conda" only if you are an expert Python user who is already accustomed to managing Python / Anaconda installations on your own.

Examples

```
## Not run:

# use python with a project
renv::use_python()

# use python with a project; create the environment
# within the project directory in the '.venv' folder
renv::use_python(name = ".venv")

# use python with a pre-existing virtual environment located elsewhere
renv::use_python(name = "~/.virtualenvs/env")

# use virtualenv python with a project
renv::use_python(type = "virtualenv")

# use conda python with a project
renv::use_python(type = "conda")

# deactivate Python integration
renv::use_python(python = FALSE)


## End(Not run)
```

Index