Package 'pak'

January 15, 2023

Version 0.4.0

Title Another Approach to Package Installation

Description The goal of 'pak' is to make package installation faster and more reliable. In particular, it performs all HTTP operations in parallel, so metadata resolution and package downloads are fast. Metadata and package files are cached on the local disk as well. 'pak' has a dependency solver, so it finds version conflicts before performing the installation. This version of 'pak' supports CRAN, 'Bioconductor' and 'GitHub' packages as well.

License GPL-3

Encoding UTF-8

LazyData true

ByteCompile true

RoxygenNote 7.2.1.9000

Depends R (>= 3.2)

Imports tools, utils

Suggests callr (>= 3.7.0), cli (>= 3.2.0), covr, curl (>= 4.3.2), desc (>= 1.4.1), digest, distro, filelock (>= 1.0.2), gitcreds, glue (>= 1.6.2), mockery, pingr, jsonlite (>= 1.8.0), pkgcache (>= 2.0.4), pkgdepends (>= 0.4.0), pkgsearch (>= 3.1.0), prettyunits, processx (>= 3.5.2), ps (>= 1.6.0), rprojroot (>= 2.0.2), rstudioapi, testthat, withr

Note This field has Remotes syntax, but repeat remotes in `Remotes`!

Config/needs/dependencies callr, desc, cli, curl, distro, filelock, glue, jsonlite, pkgcache, pkgdepends, pkgsearch, prettyunits, processx, ps, rprojroot

Config/Needs/website r-lib/asciicast, r-lib/roxygen2, tidyverse/tidytemplate

Config/testthat/edition 3

Config/build/extra-sources configure*

URL https://pak.r-lib.org/

BugReports https://github.com/r-lib/pak/issues

BuildResaveData no

NeedsCompilation no

Author Gábor Csárdi [aut, cre],
Jim Hester [aut],
RStudio [cph, fnd]

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Repository CRAN

Date/Publication 2023-01-15 21:50:02 UTC

R topics documented:

`cache_summary` *Package cache utilities*

Description

Various utilities to inspect and clean the package cache. See the pkgcache package if you need for control over the package cache.

Usage

```
cache_summary()

cache_list(...)

cache_delete(...)

cache_clean()
```

Arguments

`...`	For `cache_list()` and `cache_delete()`, `...` may contain filters, where the argument name is the column name. E.g. `package`, `version`, etc. Call `cache_list()` without arguments to see the available column names. If you call `cache_delete()` without arguments, it will delete all cached files.

Details

`cache_summary()` returns a summary of the package cache.

`cache_list()` lists all (by default), or a subset of packages in the package cache.

`cache_delete()` deletes files from the cache.

`cache_clean()` deletes all files from the cache.

Value

`cache_summary()` returns a list with elements:

- `cachepath`: absolute path to the package cache
- `files`: number of files (packages) in the cache

- `size`: total size of package cache in bytes

`cache_list()` returns a data frame with the data about the cache.

`cache_delete()` returns nothing.

`cache_clean()` returns nothing.

Examples

```
cache_summary()

cache_list()

cache_list(package = "recipes")

cache_list(platform = "source")

cache_delete(package = "knitr")
cache_delete(platform = "macos")

cache_clean()
```

`FAQ` *Frequently Asked Questions*

Description

Please take a look at this list before asking questions.

Package installation

How do I reinstall a package?:

pak does not reinstall a package, if the same version is already installed. Sometimes you still want a reinstall, e.g. to fix a broken installation. In this case you can delete the package and then install it, or use the `?reinstall` parameter:

```
pak::pkg_install("tibble")
```

```
pak::pkg_install("tibble?reinstall")
```

How do I install a dependency from a binary package:

Sometimes it is suffifient to install the binary package of an older version of a dependency, instead of the newer source package that potentially needs compilers, system tools or libraries.

`pkg_install()` and `lockfile_create()` default to `upgrade = FALSE`, which always chooses binaries over source packages, so if you use `pkg_install()` you don't need to do anything extra.

The `local_install_*` functions default to `upgrade = TRUE`, as does `pak()` with `pkf = NULL`, so for these you need to explicitly use `upgrade = FALSE`.

How do I install a package from source?:

To force the installation of a source package (instead of a binary package), use the `?source` parameter:

```
pak::pkg_install("tibble?source")
```

How do I install the latest version of a dependency?:

If you want to always install a dependency from source, because you want the latest version or some other reason, you can use the `source` parameter with the <package>= form: `<package>=?source`. For example to install tibble, with its cli dependency installed from source you could write:

```
pak::pkg_install(c("tibble", "cli=?source"))
```

How do I ignore an optional dependency?:

```
pak::pkg_install(
  c("tibble", "DiagrammeR=?ignore", "formattable=?ignore"),
  dependencies = TRUE
)
```

The syntax is

```
<packagename>=?ignore
```

Note that you can only ignore *optional* dependencies, i.e. packages in `Suggests` and `Enhances`.

Others

How can I use pak with renv?:

You cannot currently, but keep on eye on this issue: https://github.com/r-lib/pak/issues/343

`Get started with pak` *Simplified manual. Start here!*

Description

You don't need to read long manual pages for a simple task. This manual page collects the most common pak use cases.

Package installation

Install a package from CRAN or Bioconductor:

```
pak::pkg_install("tibble")
```

pak automatically sets a CRAN repository and the Bioconductor repositories that correspons to the current R version.

Install a package from GitHub:

```
pak::pkg_install("tidyverse/tibble")
```

Use the `user/repo` form. You can specify a branch or tag: `user/repo@branch` or `user/repo@tag`.

Install a package from a URL:

```
pak::pkg_install(
  "url::https://cran.r-project.org/src/contrib/Archive/tibble/tibble_3.1.7.tar.gz"
)
```

The URL may point to an R package file, made with `R CMD build`, or a `.tar.gz` or `.zip` archive of a package tree.

Package updates

Update a package:

```
pak::pkg_install("tibble")
```

`pak::pkg_install()` automatically updates the package.

Update all dependencies of a package:

```
pak::pkg_install("tibble", upgrade = TRUE)
```

`update` = `TRUE` updates the package itself and all of its dependencies, if necessary.

Reinstall a package:
Add `?reinstall` to the package name or package reference in general:

```
pak::pkg_install("tibble?reinstall")
```

Dependency lookup

Dependencies of a CRAN or Bioconductor package:

```
pak::pkg_deps("tibble")
```

The results are returned in a data frame.

Dependency tree of a CRAN / Bioconductor package:

```
pak::pkg_deps_tree("tibble")
```

The results are also silently returned in a data frame.

Dependency tree of a package on GitHub:

```
pak::pkg_deps_tree("tidyverse/tibble")
```

Use the `user/repo` form. As usual, you can also select a branch, tag, or sha, with the `user/repo@branch`, `user/repo@tag` or `user/repo@sha` forms.

Dependency tree of the package in the current directory:

```
pak::local_deps_tree("tibble")
```

Assuming package is in directory `tibble`.

Explain a recursive dependency:
How does tibble depend on rlang?

```
pak::pkg_deps_explain("tibble", "rlang")
```

Use can also use the `user/repo` form for packages from GitHub, `url::...` for packages at URLs, etc.

Package development

Install dependencies of local package:

```
pak::local_install_deps()
```

Install local package:

```
pak::local_install()
```

Install all dependencies of local package:

```
pak::local_install_dev_deps()
```

Installs development and optional dependencies as well.

Repositories

List current repositories:

```
pak::repo_get()
```

If you haven't set a CRAN or Bioconductor repository, pak does that automatically.

Add custom repository:

```
pak::repo_add(rhub = 'https://r-hub.r-universe.dev')
pak::repo_get()
```

Remove custom repositories:

```
options(repos = getOption("repos")["CRAN"])
pak::repo_get()
```

If you set the `repos` option to a CRAN repo only, or unset it completely, then pak keeps only CRAN and (by default) Bioconductor.

Time travel using RSPM:

```
pak::repo_add(CRAN = "RSPM@2022-06-30")
pak::repo_get()
```

Sets a repository that is equivalent to CRAN's state closest to the specified date. Name this repository `CRAN`, otherwise pak will also add a default CRAN repository.

Time travel using MRAN:

```
pak::repo_add(CRAN = "MRAN@2022-06-30")
pak::repo_get()
```

Sets a repository that is equivalent to CRAN's state at the specified date. Name this repository `CRAN`, otherwise pak will also add a default CRAN repository.

Caches

By default pak caches both metadata and downloaded packages.

Inspect metadata cache:

```
pak::meta_list()
```

Update metadata cache:

By default `pkg_install()` and similar functions automatically update the metadata for the currently set repositories if it is older than 24 hours. You can also force an update manually:

```
pak::meta_update()
```

Clean metadata cache:

```
pak::meta_clean(force = TRUE)
pak::meta_summary()
```

Inspect package cache:

Downloaded packages are also cached.

```
pak::cache_list()
```

View a package cache summary:

```
pak::cache_summary()
```

Clean package cache:

```
pak::cache_clean()
```

Libraries

List packages in a library:

```
pak::lib_status(Sys.getenv("R_LIBS_USER"))
```

Pass the directory of the library as the argument.

`Great pak features` *A list of the most important pak features*

Description

A list of the most important pak features.

pak is fast

Parallel HTTP:

pak performs HTTP queries concurrently. This is true when

- it downloads package metadata from package repositories,
- it resolves packages from CRAN, GitHub, URLs, etc,
- it downloads the actual package files,
- etc.

Parallel installation:

pak installs packages concurrently, as much as their dependency graph allows this.

Caching:

pak caches metadata and package files, so you don't need to re-download the same files over and over.

pak is safe

Plan installation up front:

pak creates an installation plan before downloading any packages. If the plan is unsuccessful, then it fails without downloading any packages.

Auto-install missing dependencies:

When requesting the installation of a package, pak makes sure that all of its dependencies are also installed.

Keeping binary packages up-to-date:

pak automatically discards binary packages from the cache, if a new build of the same version is available on CRAN.

Correct CRAN metadata errors:

pak can correct some of CRAN's metadata issues, e.g.:

- New version of the package was released since we obtained the metadata.
- macOS binary package is only available at https://mac.r-project.org/ because of a synchronization issue.

Graceful handling of locked package DLLs on Windows:

pak handles the situation of locked package DLLs, as well as possible. It detects which process locked them, and offers the choice of terminating these processes. It also unloads packages from the current R session as needed.

pak keeps its own dependencies isolated:
pak keeps its own dependencies in a private package library and never loads any packages. (Only in background processes).

pak is convenient

pak comes as a self-contained binary package:
On the most common platforms. No dependencies, no system dependencies, no compiler needed. (See also the installation manual.)

Install packages from multiple sources:

- CRAN, Bioconductor
- GitHub
- URLs
- Local files or directories.

Ignore certain optional dependencies:
pak can ignore certain optional dependencies if requested.

CRAN package file sizes:
pak knows the sizes of CRAN package files, so it can estimate how much data you need to download, before the installation.

Bioconductor version detection:
pak automatically selects the Bioconductor version that is appropriate for your R version. No need to set any repositories.

Time travel with MRAN or RSPM:
pak can use MRAN (Microsoft R Application Network, https://mran.microsoft.com/) or RSPM (RStudio Public Package Manager, https://packagemanager.rstudio.com/client/#/) to install from snapshots or CRAN.

pak can install dependencies of local packages:
Very handy for package development!

`handle_package_not_found` *Install missing packages on the fly*

Description

Use this function to set up a global error handler, that is called if R fails to load a package. This handler will offer you the choice of installing the missing package (and all its dependencies), and in some cases it can also remedy the error and restart the code.

Usage

```
handle_package_not_found(err)
```

Arguments

`err`	The error object, of class `packageNotFoundError`.

Details

You are not supposed to call this function directly. Instead, set it up as a global error handler, possibly in your `.Rprofile` file:

```
if (interactive() && getRversion() >= "4.0.0") {
  globalCallingHandlers(
    packageNotFoundError = function(err) {
      try(pak::handle_package_not_found(err))
    }
  )
}
```

Global error handlers are only supported in R 4.0.0 and later.

Currently `handle_package_not_found()` does not do anything in non-interactive mode (including in knitr, testthat and RStudio notebooks), this might change in the future.

In some cases it is possible to remedy the original computation that tried to load the missing package, and pak will offer you to do so after a successful installation. Currently, in R 4.0.4, it is not possible to continue a failed `library()` call.

Value

Nothing.

`Installing pak` *All about installing pak.*

Description

Read this if the default installation methods do not work for you or if you want the release candidate or development version.

Pre-built binaries:

Our pre-built binaries have the advantage that they are completely self-containted and dependency free. No additional R packages, system libraries or tools (e.g. compilers) are needed for them. Install a pre-built binary build of pak from our repository on GitHub:

```
install.packages("pak", repos = sprintf(
  "https://r-lib.github.io/p/pak/stable/%s/%s/%s",
  .Platform$pkgType,
  R.Version()$os,
  R.Version()$arch
))
```

This is supported for the following systems:

OS	CPU	R version
Linux	x86_64	R 3.4.0 - R-devel
Linux	aarch64	R 3.4.0 - R-devel
macOS High Sierra+	x86_64	R 3.4.0 - R-devel
macOS Big Sur+	aarch64	R 4.1.0 - R-devel
Windows	x86_64	R 3.4.0 - R-devel

Notes:

- For macOS we only support the official CRAN R build. Other builds, e.g. Homebrew R, are not supported.
- We only support R builds that have an R shared library. CRAN's Windows and macOS installers are such, so the the R builds in the common Linux distributions. But this might be an issue if you build R yourself without the `--enable-R-shlib` option.

Install from CRAN:

Install the released version of the package from CRAN as usual:

```
install.packages("pak")
```

This potentially needs a C compiler on platforms CRAN does not have binaries packages for.

Nightly builds:

We have nightly binary builds, for the same systems as the table above:

```
install.packages("pak", repos = sprintf(
  "https://r-lib.github.io/p/pak/devel/%s/%s/%s",
  .Platform$pkgType,
  R.Version()$os,
  R.Version()$arch
))
```

`stable`, `rc` *and* `devel` *streams:*

We have three types of binaries available:

- `stable` corresponds to the latest CRAN release of CRAN.
- `rc` is a release candidate build, and it is available about 1-2 weeks before a release. Otherwise it is the same as the `stable` build.
- `devel` has builds from the development tree. Before release it might be the same as the `rc` build.

The streams are available under different repository URLs:

```
stream <- "rc"
install.packages("pak", repos = sprintf(
  "https://r-lib.github.io/p/pak/%s/%s/%s/%s",
  stream,
  .Platform$pkgType,
  R.Version()$os,
  R.Version()$arch
))
```

`lib_status` *Status of packages in a library*

Description

Status of packages in a library

Usage

```
lib_status(lib = .libPaths()[1])

pkg_list(lib = .libPaths()[1])
```

Arguments

`lib`	Path to library.

Value

Data frame the contains data about the packages installed in the library. `pak:::include_docs("pkgdepends", "docs/lib-status-return.rds")`

Examples

```
lib_status(.Library)
```

See Also

Other package functions: `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

`local_deps` *Dependencies of a package tree*

Description

Dependencies of a package tree

Usage

```
local_deps(root = ".", upgrade = TRUE, dependencies = NA)

local_deps_tree(root = ".", upgrade = TRUE, dependencies = NA)

local_dev_deps(root = ".", upgrade = TRUE, dependencies = TRUE)

local_dev_deps_tree(root = ".", upgrade = TRUE, dependencies = TRUE)
```

Arguments

`root`	Path to the package tree.
`upgrade`	Whether to use the most recent available package versions.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Value

All of these functions return the dependencies in a data frame. `local_deps_tree()` and `local_dev_deps_tree()` also print the dependency tree.

See Also

Other local package trees: `local_deps_explain()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`

`local_deps_explain` *Explain dependencies of a package tree*

Description

These functions are similar to `pkg_deps_explain()`, but work on a local package tree. `local_dev_deps_explain()` also includes development dependencies.

Usage

```
local_deps_explain(deps, root = ".", upgrade = TRUE, dependencies = NA)

local_dev_deps_explain(deps, root = ".", upgrade = TRUE, dependencies = TRUE)
```

Arguments

`deps`	Package names of the dependencies to explain.
`root`	Path to the package tree.
`upgrade`	Whether to use the most recent available package versions.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

See Also

Other local package trees: `local_deps()`, `local_install_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`

`local_install` *Install a package tree*

Description

Installs a package tree (or source package file), together with its dependencies.

Usage

```
local_install(
  root = ".",
  lib = .libPaths()[1],
  upgrade = TRUE,
  ask = interactive(),
  dependencies = NA
)
```

Arguments

`root`	Path to the package tree.
`lib`	Package library to install the packages to. Note that *all* dependent packages will be installed here, even if they are already installed in another library. The only exceptions are base and recommended packages installed in `.Library`. These are not duplicated in `lib`, unless a newer version of a recommemded package is needed.
`upgrade`	When `FALSE`, the default, pak does the minimum amount of work to give you the latest version(s) of `pkg`. It will only upgrade dependent packages if `pkg`, or one of their dependencies explicitly require a higher version than what you currently have. It will also prefer a binary package over to source package, even it the binary package is older. When `upgrade = TRUE`, pak will ensure that you have the latest version(s) of `pkg` and all their dependencies.
`ask`	Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Details

`local_install()` is equivalent to `pkg_install("local::.")`.

Value

Data frame, with information about the installed package(s).

See Also

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install_dev_deps(`
`local_package_trees`, `pak()`

`local_install_deps` *Install the dependencies of a package tree*

Description

Installs the hard dependencies of a package tree (or source package file), without installing the package tree itself.

Usage

```
local_install_deps(
  root = ".",
  lib = .libPaths()[1],
  upgrade = TRUE,
  ask = interactive(),
  dependencies = NA
)
```

Arguments

`root`	Path to the package tree.
`lib`	Package library to install the packages to. Note that *all* dependent packages will be installed here, even if they are already installed in another library. The only exceptions are base and recommended packages installed in `.Library`. These are not duplicated in `lib`, unless a newer version of a recommemded package is needed.
`upgrade`	When `FALSE`, the default, pak does the minimum amount of work to give you the latest version(s) of `pkg`. It will only upgrade dependent packages if `pkg`, or one of their dependencies explicitly require a higher version than what you currently have. It will also prefer a binary package over to source package, even it the binary package is older. When `upgrade = TRUE`, pak will ensure that you have the latest version(s) of `pkg` and all their dependencies.
`ask`	Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies,

- `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Details

Note that development (and optional) dependencies, under `Suggests` in `DESCRIPTION`, are not installed. If you want to install them as well, use `local_install_dev_deps()`.

Value

Data frame, with information about the installed package(s).

See Also

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_dev_deps()`, `local_install()`, `local_package_trees`, `pak()`

`local_install_dev_deps` *Install all (development) dependencies of a package tree*

Description

Installs all dependencies of a package tree (or source package file), without installing the package tree itself. It installs the development dependencies as well, specified in the `Suggests` field of `DESCRIPTION`.

Usage

```
local_install_dev_deps(
  root = ".",
  lib = .libPaths()[1],
  upgrade = TRUE,
  ask = interactive(),
  dependencies = TRUE
)
```

Arguments

`root`	Path to the package tree.
`lib`	Package library to install the packages to. Note that *all* dependent packages will be installed here, even if they are already installed in another library. The only exceptions are base and recommended packages installed in `.Library`. These are not duplicated in `lib`, unless a newer version of a recommemded package is needed.

`upgrade`	When `FALSE`, the default, pak does the minimum amount of work to give you the latest version(s) of pkg. It will only upgrade dependent packages if pkg, or one of their dependencies explicitly require a higher version than what you currently have. It will also prefer a binary package over to source package, even it the binary package is older. When `upgrade = TRUE`, pak will ensure that you have the latest version(s) of pkg and all their dependencies.
`ask`	Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

See Also

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install()`, `local_package_trees`, `pak()`

`local_package_trees` *About local package trees*

Description

pak can install packages from local package trees. This is convenient for package development. See the following functions:

- `local_install()` installs a package from a package tree and all of its dependencies.
- `local_install_deps()` installs all hard dependencies of a package.
- `local_install_dev_deps()` installs all hard and soft dependencies of a package. This function is intended for package development.

Details

Note that the last two functions do not install the package in the specified package tree itself, only its dependencies. This is convenient if the package itself is loaded via some other means, e.g. `devtools::load_all()`, for development.

See Also

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install_dev_deps(`
`local_install()`, `pak()`

local_system_requirements — *Query system requirements*

Description

Returns a character vector of commands to run that will install system requirements for the queried operating system.

`local_system_requirements()` queries system requirements for a dev package (and its dependencies) given its `root` path.

`pkg_system_requirements()` queries system requirements for existing packages (and their dependencies).

Usage

```
local_system_requirements(
  os = NULL,
  os_release = NULL,
  root = ".",
  execute = FALSE,
  sudo = execute,
  echo = FALSE
)

pkg_system_requirements(
  package,
  os = NULL,
  os_release = NULL,
  execute = FALSE,
  sudo = execute,
  echo = FALSE
)
```

Arguments

`os, os_release`	The operating system and operating system release version, e.g. "ubuntu", "debian", "centos", "redhat". See `https://github.com/rstudio/r-system-requirements#operating-systems` for all full list of supported operating systems. If `NULL`, the default, these will be looked up using `distro::distro()`.
`root`	Path to the package tree.
`execute, sudo`	If `execute` is `TRUE`, pak will execute the system commands (if any). If `sudo` is `TRUE`, pak will prepend the commands with sudo.
`echo`	If `echo` is `TRUE` and `execute` is `TRUE`, echo the command output.
`package`	Package names to lookup system requirements for.

Value

A character vector of commands needed to install the system requirements for the package.

Examples

```
local_system_requirements("ubuntu", "20.04")


pkg_system_requirements("pak", "ubuntu", "20.04")
pkg_system_requirements("pak", "redhat", "7")
pkg_system_requirements("config", "ubuntu", "20.04") # no sys reqs
pkg_system_requirements("curl", "ubuntu", "20.04")
pkg_system_requirements("git2r", "ubuntu", "20.04")
pkg_system_requirements(c("config", "git2r", "curl"), "ubuntu", "20.04")
# queried packages must exist
pkg_system_requirements("iDontExist", "ubuntu", "20.04")
pkg_system_requirements(c("curl", "iDontExist"), "ubuntu", "20.04")
```

`lockfile_create` *Create a lock file*

Description

The lock file can be used later, possibly in a new R session, to carry out the installation of the dependencies, with `lockfile_install()`.

Usage

```
lockfile_create(
  pkg = "deps::.",
  lockfile = "pkg.lock",
  lib = NULL,
  upgrade = FALSE,
  dependencies = NA
)
```

Arguments

`pkg` Package names or package references. E.g.

- `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general,
- `tidyverse/ggplot2`: package from GitHub,
- `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch,
- `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL,
- `.`: package in the current working directory.

	See "Package sources" for more details.
`lockfile`	Path to the lock file.
`lib`	Package library to install the packages to. Note that *all* dependent packages will be installed here, even if they are already installed in another library. The only exceptions are base and recommended packages installed in `.Library`. These are not duplicated in `lib`, unless a newer version of a recommemded package is needed.
`upgrade`	When `FALSE`, the default, pak does the minimum amount of work to give you the latest version(s) of `pkg`. It will only upgrade dependent packages if `pkg`, or one of their dependencies explicitly require a higher version than what you currently have. It will also prefer a binary package over to source package, even it the binary package is older. When `upgrade = TRUE`, pak will ensure that you have the latest version(s) of `pkg` and all their dependencies.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Details

Note, since the URLs of CRAN and most CRAN-like repositories change over time, in practice you cannot use the lock file *much* later. For example, binary packages of older package version might be deleted from the repository, breaking the URLs in the lock file.

Currently the intended use case of lock files in on CI systems, to facilitate caching. The (hash of the) lock file provides a good key for caching systems.

See Also

Other lock files: `lockfile_install()`

`lockfile_install` *Install packages based on a lock file*

Description

Install a lock file that was created with `lockfile_create()`.

Usage

```
lockfile_install(lockfile = "pkg.lock", lib = .libPaths()[1], update = TRUE)
```

Arguments

`lockfile`	Path to the lock file.
`lib`	Library to carry out the installation on.
`update`	Whether to online install the packages that either not installed in `lib`, or a different version is installed for them.

See Also

Other lock files: `lockfile_create()`

`meta_summary` *Metadata cache utilities*

Description

Various utilities to inspect, update and clean the metadata cache. See the pkgcache package if you need for control over the metadata cache.

Usage

```
meta_summary()

meta_list(pkg = NULL)

meta_update()

meta_clean(force = FALSE)
```

Arguments

`pkg`	Package names, if specified then only entries for pkg are returned.
`force`	If `FALSE`, then pak will ask for confirmation.

Details

`meta_summary()` returns a summary of the metadata cache.

`meta_list()` lists all (or some) packages in the metadata database.

`meta_update()` updates the metadata database. You don't normally need to call this function manually, because all pak functions (e.g. `pkg_install()`, `pkg_download()`, etc.) call it automatically, to make sure that they use the latest available metadata.

`meta_clean()` deletes the whole metadata DB.

Value

meta_summary() returns a list with entries:

- cachepath: absolute path of the metadata cache.
- current_db: the file that contains the current metadata database. It is currently an RDS file, but this might change in the future.
- raw_files: the files that are the downloaded PACKAGES* files.
- db_files: all metadata database files.
- size: total size of the metadata cache.

meta_list() returns a data frame of all available packages in the configured repositories.

meta_update() returns nothing.

meta_clean() returns nothing

Examples

Metadata cache summary:

```
meta_summary()
#> $cachepath
#> [1] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata"
#>
#> $current_db
#> [1] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/pkgs-34444e3072.rds"
#>
#> $raw_files
#>  [1] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCann-59693086a0/b
#>  [2] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCann-59693086a0/s
#>  [3] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCexp-90d4a3978b/b
#>  [4] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCexp-90d4a3978b/s
#>  [5] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCsoft-2a43920999/
#>  [6] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCsoft-2a43920999/
#>  [7] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCworkflows-26330b
#>  [8] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/BioCworkflows-26330b
#>  [9] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/CRAN-075c426938/bin/
#> [10] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/CRAN-075c426938/src/
#>
#> $db_files
#> [1] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/pkgs-34444e3072.rds"
#> [2] "/Users/gaborcsardi/Library/Caches/org.R-project.R/R/pkgcache/_metadata/pkgs-ccacf1b389.rds"
#>
#> $size
#> [1] 174848200
```

The current metadata DB:

```
meta_list()
```

Selected packages only:

```
meta_list(pkg = c("shiny", "htmlwidgets"))
```

Update the metadata DB

```
meta_update()
```

Delete the metadata DB

```
meta_clean()
```

`Package dependency types` *Various types of R package dependencies*

Description

Various types of R package dependencies

Details

```
pak:::include_docs("pkgdepends", "docs/deps.rds")
```

`Package sources` *Install packages from CRAN, Bioconductor, GitHub, URLs, etc.*

Description

Install packages from CRAN, Bioconductor, GitHub, URLs, etc. Learn how to tell pak which packages to install, and where those packages can be found.

If you want a quick overview of package sources, see "Get started with pak".

Details

```
pak:::include_docs("pkgdepends", "docs/pkg-refs.rds", top = FALSE)
```

pak — *Install specified required packages*

Description

Install the specified packages, or the ones required by the package or project in the current working directory.

Usage

```
pak(pkg = NULL, ...)
```

Arguments

`pkg`	Package names or remote package specifications to install. See pak package sources for details. If `NULL`, will install all development dependencies for the current package.
`...`	Extra arguments are passed to `pkg_install()` or `local_install_dev_deps()`.

Details

This is a convenience function:

- If you want to install some packages, it is easier to type than `pkg_install()`.
- If you want to install all the packages that are needed for the development of a package or project, then it is easier to type than `local_install_dev_deps()`.
- You don't need to remember two functions to install packages, just one.

See Also

Other package functions: `lib_status()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

Other local package trees: `local_deps_explain()`, `local_deps()`, `local_install_deps()`, `local_install_dev_deps(`
`local_install()`, `local_package_trees`

pak configuration — *Environment variables and options that modify the defualt behavior*

Description

pak behavior can be finetuned with environment variables and options (as in `base::options()`).

R options affecting pak's behavior

`Ncpus`:
Set to the desired number of worker processes for package installation. If not set, then pak will use the number of logical processors in the machine.

`repos`:
The CRAN-like repositories to use. See `base::options()` for details.

pak configuration

Configuration entries (unless noted otherwise on this manual page) have a corresponding environment variable, and a corresponding option.

The environment variable is always uppercase and uses underscores as the word separator. It always has the `PKG_` prefix.

The option is typically lowercase, use it uses underscores as the word separator, but it always has the `pkg.` prefix (notice the dot!).

Some examples:

Config entry name	Env var name	Option name
platforms	`PKG_PLATFORMS`	`pkg.platforms`
cran_mirror	`PKG_CRAN_MIRROR`	`pkg.cran_mirror`

pak configuration entries:
`pak:::doc_config()`

Notes:
From version 0.4.0 pak copies the `PKG_*` environment variables and the `pkg.*` options to the pak subprocess, where they are actually used, so you don't need to restart R or reaload pak after a configuration change.

`pak_cleanup` *Clean up pak caches*

Description

Clean up pak caches

Usage

```
pak_cleanup(
  package_cache = TRUE,
  metadata_cache = TRUE,
  pak_lib = TRUE,
  force = FALSE
)
```

Arguments

`package_cache`	Whether to clean up the cache of package files.
`metadata_cache`	Whether to clean up the cache of package meta data.
`pak_lib`	This argument is now deprecated and does nothing.
`force`	Do not ask for confirmation. Note that to use this function in non-interactive mode, you have to specify `force = FALSE`.

See Also

Other pak housekeeping: `pak_sitrep()`

`pak_install_extra` *Install all optional dependencies of pak*

Description

These packages are not required for any pak functionality. They are recommended for some functions that return values that are best used with these packages. E.g. many functions return data frames, which print nicer when the pillar package is available.

Usage

```
pak_install_extra(upgrade = FALSE)
```

Arguments

`upgrade`	Whether to install or upgrade to the latest versions of the optional packages.

Details

Currently only one package is optional: **pillar**.

`pak_setup` *Set up private pak library (deprecated)*

Description

This function is deprecated and does nothing. Recent versions of pak do not need a `pak_setup()` call.

Usage

```
pak_setup(mode = c("auto", "download", "copy"), quiet = FALSE)
```

Arguments

`mode`	Where to get the packages from. "download" will try to download them from CRAN. "copy" will try to copy them from your current "regular" package library. "auto" will try to copy first, and if that fails, then it tries to download.
`quiet`	Whether to omit messages.

Value

The path to the private library, invisibly.

`pak_sitrep` *pak SITuation REPort*

Description

It prints

- pak version,
- the current library path,
- location of the private library,
- whether the pak private library exists,
- whether the pak private library is functional.

Usage

```
pak_sitrep()
```

Examples

```
pak_sitrep()
```

See Also

Other pak housekeeping: `pak_cleanup()`

`pak_update` *Update pak itself*

Description

Use this function to update the released or development version of pak.

Usage

```
pak_update(force = FALSE, stream = c("auto", "stable", "rc", "devel"))
```

Arguments

`force`	Whether to force an update, even if no newer version is available.
`stream`	Whether to update to the • `"stable"`, • `"rc"` (release candidate) or • `"devel"` (development) version. • `"auto"` updates to the same stream as the current one. Often there is no release candidate version, then `"rc"` also installs the stable version.

Value

Nothing.

`pkg_deps` *Look up the dependencies of a package*

Description

Look up the dependencies of a package

Usage

```
pkg_deps(pkg, upgrade = TRUE, dependencies = NA)
```

Arguments

`pkg`	Package names or package references. E.g. • `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general, • `tidyverse/ggplot2`: package from GitHub, • `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch, • `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL, • `.`: package in the current working directory. See "Package sources" for more details.
`upgrade`	Whether to use the most recent available package versions.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Value

A data frame with the dependency data, it includes `pkg` as well. It has the following columns. `pak:::include_docs("pkgdepends", "docs/resolution-result.rds")`

Examples

```
pkg_deps("dplyr")
```

For a package on GitHub:

```
pkg_deps("r-lib/callr")
```

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

`pkg_deps_explain` *Explain how a package depends on other packages*

Description

Extract dependency chains from pkg to `deps`.

Usage

```
pkg_deps_explain(pkg, deps, upgrade = TRUE, dependencies = NA)
```

Arguments

`pkg`	Package names or package references. E.g. • `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general, • `tidyverse/ggplot2`: package from GitHub, • `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch, • `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL, • `.`: package in the current working directory. See "Package sources" for more details.
`deps`	Package names of the dependencies to explain.
`upgrade`	Whether to use the most recent available package versions.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Details

This function is similar to `pkg_deps_tree()`, but its output is easier to read if you are only interested is certain packages (`deps`).

Value

A named list with a print method. First entries are the function arguments: `pkg`, `deps`, `dependencies`, the last one is `paths` and it contains the results in a named list, the names are the package names in `deps`.

Examples

How does dplyr depend on rlang?

```
pkg_deps_explain("dplyr", "rlang")
```

How does the GH version of usethis depend on cli and ps?

```
pkg_deps_explain("r-lib/usethis", c("cli", "ps"))
```

`pkg_deps_tree` *Draw the dependency tree of a package*

Description

Draw the dependency tree of a package

Usage

```
pkg_deps_tree(pkg, upgrade = TRUE, dependencies = NA)
```

Arguments

`pkg`	Package names or package references. E.g. • `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general, • `tidyverse/ggplot2`: package from GitHub, • `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch, • `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL, • `.`: package in the current working directory. See "Package sources" for more details.
`upgrade`	Whether to use the most recent available package versions.
`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Value

The same data frame as `pkg_deps()`, invisibly.

Examples

```
pkg_deps_tree("dplyr")

pkg_deps_tree("r-lib/usethis")
```

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

`pkg_download` *Download a package and its dependencies*

Description

TODO: explain result

Usage

```
pkg_download(
  pkg,
  dest_dir = ".",
  dependencies = FALSE,
  platforms = NULL,
  r_versions = NULL
)
```

Arguments

`pkg` Package names or package references. E.g.

- `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general,
- `tidyverse/ggplot2`: package from GitHub,
- `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch,
- `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL,
- `.`: package in the current working directory.

See "Package sources" for more details.

`dest_dir` Destination directory for the packages. If it does not exist, then it will be created.

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

- `NA`: only required (hard) dependencies,
- `TRUE`: required dependencies plus optional and development dependencies,

- `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

`platforms`	Types of binary or source packages to download. The default is the value of `pkgdepends::default_platforms()`.
`r_versions`	R version(s) to download packages for. (This does not matter for source packages, but it does for binaries.) It defaults to the current R version.

Value

Data frame with information about the downloaded packages, invisibly. Columns: `pak:::include_docs("pkgdepends", "docs/download-result.rds")`

Examples

```
dl <- pkg_download("forcats")

dl

dl$fulltarget

pkg_download("r-lib/pak", platforms = "source")
```

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_install()`, `pkg_remove()`, `pkg_status()`

`pkg_history` *Query the history of a CRAN package*

Description

Query the history of a CRAN package

Usage

```
pkg_history(pkg)
```

Arguments

`pkg`	Package name.

Value

A data frame, with one row per package version. The columns are the entries of the `DESCRIPTION` files in the released package versions.

Examples

```
pkg_history("ggplot2")
```

---

pkg_install *Install packages*

Description

Install one or more packages and their dependencies into a single package library.

Usage

```
pkg_install(
  pkg,
  lib = .libPaths()[[1L]],
  upgrade = FALSE,
  ask = interactive(),
  dependencies = NA
)
```

Arguments

`pkg`	Package names or package references. E.g. • `ggplot2`: package from CRAN, Bioconductor or a CRAN-like repository in general, • `tidyverse/ggplot2`: package from GitHub, • `tidyverse/ggplot2@v3.4.0`: package from GitHub tag or branch, • `https://examples.com/.../ggplot2_3.3.6.tar.gz`: package from URL, • `.`: package in the current working directory. See "Package sources" for more details.
`lib`	Package library to install the packages to. Note that *all* dependent packages will be installed here, even if they are already installed in another library. The only exceptions are base and recommended packages installed in `.Library`. These are not duplicated in `lib`, unless a newer version of a recommemded package is needed.
`upgrade`	When `FALSE`, the default, pak does the minimum amount of work to give you the latest version(s) of `pkg`. It will only upgrade dependent packages if `pkg`, or one of their dependencies explicitly require a higher version than what you currently have. It will also prefer a binary package over to source package, even it the binary package is older. When `upgrade = TRUE`, pak will ensure that you have the latest version(s) of `pkg` and all their dependencies.
`ask`	Whether to ask for confirmation when installing a different version of a package that is already installed. Installations that only add new packages never require confirmation.

`dependencies`	What kinds of dependencies to install. Most commonly one of the following values: • `NA`: only required (hard) dependencies, • `TRUE`: required dependencies plus optional and development dependencies, • `FALSE`: do not install any dependencies. (You might end up with a non-working package, and/or the installation might fail.) See Package dependency types for other possible values and more information about package dependencies.

Value

(Invisibly) A data frame with information about the installed package(s).

Examples

```
pkg_install("dplyr")
```

Upgrade dplyr and all its dependencies:

```
pkg_install("dplyr", upgrade = TRUE)
```

Install the development version of dplyr:

```
pkg_install("tidyverse/dplyr")
```

Switch back to the CRAN version. This will be fast because pak will have cached the prior install.

```
pkg_install("dplyr")
```

See Also

Get started with pak, Package sources, FAQ, The dependency solver.

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_remove()`, `pkg_status()`

`pkg_name_check` *Check if an R package name is available*

Description

Additionally, look up the candidate name in a number of dictionaries, to make sure that it does not have a negative meaning.

Usage

```
pkg_name_check(name, dictionaries = NULL)
```

Arguments

`name`	Package name candidate.
`dictionaries`	Character vector, the dictionaries to query. Available dictionaries: * `wikipedia` * `wiktionary`, * `acromine` (`http://www.nactem.ac.uk/software/acromine/`), * `sentiment` (`https://github.com/fnielsen/afinn`), * `urban` (Urban Dictionary). If `NULL` (by default), the Urban Dictionary is omitted, as it is often offensive.

Details

Valid package name check:

Check the validity of `name` as a package name. See 'Writing R Extensions' for the allowed package names. Also checked against a list of names that are known to cause problems.

CRAN checks:

Check `name` against the names of all past and current packages on CRAN, including base and recommended packages.

Bioconductor checks:

Check `name` against all past and current Bioconductor packages.

Profanity check:

Check `name` with `https://www.purgomalum.com/service/containsprofanity` to make sure it is not a profanity.

Dictionaries:

See the `dictionaries` argument.

Value

`pkg_name_check` object with a custom print method.

Examples

```
pkg_name_check("sicily")
```

`pkg_remove` *Remove installed packages*

Description

Remove installed packages

Usage

```
pkg_remove(pkg, lib = .libPaths()[[1L]])
```

Arguments

`pkg`	A character vector of packages to remove.
`lib`	library to remove packages from.

Value

Nothing.

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_status()`

`pkg_search` *Search CRAN packages*

Description

Search the indexed database of current CRAN packages. It uses the pkgsearch package. See that package for more details and also `pkgsearch::pkg_search()` for pagination, more advanced searching, etc.

Usage

```
pkg_search(query, ...)
```

Arguments

`query`	Search query string.
`...`	Additional arguments passed to `pkgsearch::pkg_search()`

Value

A data frame, that is also a `pak_search_result` object with a custom print method. To see the underlying table, you can use `[]` to drop the extra classes. See examples below.

Examples

Simple search

```
pkg_search("survival")
```

See the underlying data frame

```
psro <- pkg_search("ropensci")
psro[]
```

pkg_status *Display installed locations of a package*

Description

Display installed locations of a package

Usage

```
pkg_status(pkg, lib = .libPaths())
```

Arguments

`pkg`	Name of one or more installed packages to display status for.
`lib`	One or more library paths to lookup packages status in. By default all libraries are used.

Value

Data frame with data about installations of `pkg`. `pak:::include_docs("pkgdepends", "docs/lib-status-return.rds")`

Examples

```
pkg_status("MASS")
```

See Also

Other package functions: `lib_status()`, `pak()`, `pkg_deps_tree()`, `pkg_deps()`, `pkg_download()`, `pkg_install()`, `pkg_remove()`

repo_add *Add a new CRAN-like repository*

Description

Add a new repository to the list of repositories that pak uses to look for packages.

Usage

```
repo_add(..., .list = NULL)

repo_resolve(spec)
```

Arguments

`...`	Repository specifications, possibly named character vectors. See details below.
`.list`	List or character vector of repository specifications. This argument is easier to use programmatically than `...`. See details below.
`spec`	Repository specification, a possibly named character scalar.

Details

`repo_add()` adds new repositories. It resolves the specified repositories using `repo_resolve()` and then modifies the `repos` global option.

`repo_add()` only has an effect in the current R session. If you want to keep your configuration between R sessions, then set the `repos` option to the desired value in your user or project `.Rprofile` file.

Value

`repo_resolve()` returns a named character scalar, the URL of the repository.

Repository specifications

The format of a repository specification is a named or unnamed character scalar. If the name is missing, pak adds a name automatically. The repository named `CRAN` is the main CRAN repository, but otherwise names are informational.

Currently supported repository specifications:

- URL pointing to the root of the CRAN-like repository. Example:

  ```
  https://cloud.r-project.org
  ```
- `RSPM@<date>`, RSPM (RStudio Package Manager) snapshot, at the specified date.
- `RSPM@<package>-<version>` RSPM snapshot, for the day after the release of `<version>` of `<package>`.
- `RSPM@R-<version>` RSPM snapshot, for the day after R `<version>` was released.
- `MRAN@<date>`, MRAN (Microsoft R Application Network) snapshot, at the specified date.
- `MRAN@<package>-<version>` MRAN snapshot, for the day after the release of `<version>` of `<package>`.
- `MRAN@R-<version>` MRAN snapshot, for the day after R `<version>` was released.

Notes:

- See more about RSPM at `https://packagemanager.rstudio.com/client/#/`.
- See more about MRAN snapshots at `https://mran.microsoft.com/timemachine`.
- All dates (or times) can be specified in the ISO 8601 format.
- If RSPM does not have a snapshot available for a date, the next available date is used.
- Dates that are before the first, or after the last RSPM snapshot will trigger an error.
- Dates before the first, or after the last MRAN snapshot will trigger an error.
- Unknown R or package versions will trigger an error.

Exaples

```
repo_add(RSPMdplyr100 = "RSPM@dplyr-1.0.0")
repo_get()

repo_resolve("MRAN@2020-01-21")

repo_resolve("RSPM@2020-01-21")

repo_resolve("MRAN@dplyr-1.0.0")

repo_resolve("RSPM@dplyr-1.0.0")

repo_resolve("MRAN@R-4.0.0")

repo_resolve("RSPM@R-4.0.0")
```

See Also

Other repository functions: `repo_get()`, `repo_status()`

`repo_get` *Query the currently configured CRAN-like repositories*

Description

pak uses the `repos` option, see `options()`. It also automatically adds a CRAN mirror if none is set up, and the correct version of the Bioconductor repositories. See the `cran_mirror` and `bioc` arguments.

Usage

```
repo_get(r_version = getRversion(), bioc = TRUE, cran_mirror = NULL)
```

Arguments

`r_version`	R version to use to determine the correct Bioconductor version, if `bioc = TRUE`.
`bioc`	Whether to automatically add the Bioconductor repositories to the result.
`cran_mirror`	CRAN mirror to use. Leave it at `NULL` to use the mirror in `getOption("repos")` or an automatically selected one.

Details

`repo_get()` returns the table of the currently configured repositories.

Examples

```
repo_get()
```

See Also

Other repository functions: `repo_add()`, `repo_status()`

`repo_status` *Show the status of CRAN-like repositories*

Description

It checks the status of the configured or supplied repositories.

Usage

```
repo_status(
  platforms = NULL,
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = NULL
)

repo_ping(
  platforms = NULL,
  r_version = getRversion(),
  bioc = TRUE,
  cran_mirror = NULL
)
```

Arguments

`platforms`	Platforms to use, default is the current platform, plus source packages.
`r_version`	R version(s) to use, the default is the current R version, via `getRversion()`.
`bioc`	Whether to add the Bioconductor repositories. If you already configured them via `options(repos)`, then you can set this to `FALSE`.
`cran_mirror`	The CRAN mirror to use.

Details

`repo_ping()` is similar to `repo_status()` but also prints a short summary of the data, and it returns its result invisibly.

Value

A data frame that has a row for every repository, on every queried platform and R version. It has these columns:

- `name`: the name of the repository. This comes from the names of the configured repositories in `options("repos")`, or added by pkgcache. It is typically `CRAN` for CRAN, and the current Bioconductor repositories are `BioCsoft`, `BioCann`, `BioCexp`, `BioCworkflows`.

- `url`: base URL of the repository.
- `bioc_version`: Bioconductor version, or `NA` for non-Bioconductor repositories.
- `platform`: platform, possible values are `source`, `macos` and `windows` currently.
- `path`: the path to the packages within the base URL, for a given platform and R version.
- `r_version`: R version, one of the specified R versions.
- `ok`: Logical flag, whether the repository contains a metadata file for the given platform and R version.
- `ping`: HTTP response time of the repository in seconds. If the `ok` column is `FALSE`, then this columns in `NA`.
- `error`: the error object if the HTTP query failed for this repository, platform and R version.

Examples

```
repo_status()

repo_status(
  platforms = c("windows", "macos"),
  r_version = c("4.0", "4.1")
)

repo_ping()
```

See Also

Other repository functions: `repo_add()`, `repo_get()`

`The dependency solver` *Find the ideal set of packages and versions to install*

Description

pak contains a package dependency solver, that makes sure that the package source and version requirements of all packages are satisfied, before starting an installation. For CRAN and BioC packages this is usually automatic, because these repositories are generally in a consistent state. If packages depend on other other package sources, however, this is not the case.

Details

Here is an example of a conflict detected:

```
> pak::pkg_install(c("r-lib/pkgcache@conflict", "r-lib/cli@message"))
Error: Cannot install packages:
  * Cannot install `r-lib/pkgcache@conflict`.
    - Cannot install dependency r-lib/cli@main
  * Cannot install `r-lib/cli@main`.
- Conflicts r-lib/cli@message
```

`r-lib/pkgcache@conflict` depends on the main branch of `r-lib/cli`, whereas, we explicitly requested the `message` branch. Since it cannot install both versions into a single library, pak quits.

When pak considers a package for installation, and the package is given with its name only, (e.g. as a dependency of another package), then the package may have *any* package source. This is necessary, because one R package library may contain only at most one version of a package with a given name.

pak's behavior is best explained via an example. Assume that you are installing a local package (see below), e.g. `local::.`, and the local package depends on `pkgA` and `user/pkgB`, the latter being a package from GitHub (see below), and that `pkgA` also depends on `pkgB`. Now pak must install `pkgB` *and* `user/pkgB`. In this case pak interprets `pkgB` as a package from any package source, instead of a standard package, so installing `user/pkgB` satisfies both requirements.

Note that that `cran::pkgB` and `user/pkgB` requirements result a conflict that pak cannot resolve. This is because the first one *must* be a CRAN package, and the second one *must* be a GitHub package, and two different packages with the same cannot be installed into an R package library.

Index