Package 'pak'

July 23, 2025

Title Another Approach to Package Installation

Version 0.9.0

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

URL https://pak.r-lib.org/, https://github.com/r-lib/pak

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

Depends R (>= 3.5)

Imports tools, utils

Suggests callr (>= 3.7.0), cli (>= 3.2.0), covr, curl (>= 4.3.2), desc (>= 1.4.1), filelock (>= 1.0.2), gitcreds, glue (>= 1.6.2), jsonlite (>= 1.8.0), keyring (>= 1.4.0), pingr, pkgbuild (>= 1.4.2), pkgcache (>= 2.2.4), pkgdepends (>= 0.9.0), pkgload, pkgsearch (>= 3.1.0), processx (>= 3.8.1), ps (>= 1.6.0), rstudioapi, testthat (>= 3.2.0), webfakes, withr, yaml

Biarch true

ByteCompile true

Config/build/extra-sources configure*

Config/needs/dependencies callr, cli, curl, desc, filelock, jsonlite, keyring, lpSolve, pkgbuild, pkgcache, pkgdepends, pkgsearch, processx, ps, yaml

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

Config/Needs/deploy cli@3.6.2, curl, desc, gitcreds, glue@1.6.2, gaborcsardi/jsonlite, processx, R6@2.5.1

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-05-13

Encoding UTF-8

RoxygenNote 7.3.2.9000

NeedsCompilation yes

Author Gábor Csárdi [aut, cre],
Jim Hester [aut],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
Winston Chang [ctb] (R6, callr, processx),
Ascent Digital Services [cph, fnd] (callr, processx),
Hadley Wickham [ctb, cph] (cli, curl, pkgbuild, yaml),
Jeroen Ooms [ctb] (curl, jsonlite),
Maëlle Salmon [ctb] (desc, pkgsearch),
Duncan Temple Lang [ctb] (jsonlite),
Lloyd Hilaiel [cph] (jsonlite),
Alec Wong [ctb] (keyring),
Michel Berkelaar and lpSolve authors [ctb] (lpSolve),
R Consortium [fnd] (pkgsearch),
Jay Loden [ctb] (ps),
Dave Daeschler [ctb] (ps),
Giampaolo Rodola [ctb] (ps),
Shawn Garbett [ctb] (yaml),
Jeremy Stephens [ctb] (yaml),
Kirill Simonov [ctb] (yaml),
Yihui Xie [ctb] (yaml),
Zhuoer Dong [ctb] (yaml),
Jeffrey Horner [ctb] (yaml),
Will Beasley [ctb] (yaml),
Brendan O'Connor [ctb] (yaml),
Gregory Warnes [ctb] (yaml),
Michael Quinn [ctb] (yaml),
Zhian Kamvar [ctb] (yaml),
Charlie Gao [ctb] (yaml),
Kuba Podgórski [ctb] (zip),
Rich Geldreich [ctb] (zip)

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

Repository CRAN

Date/Publication 2025-05-27 12:10:02 UTC

Contents

`Authenticated repositories`

Authenticated repositories

Description

pak supports HTTP basic authentication when interacting with CRAN-like repositories.

Configuring authenticated repositories:

To use authentication you need to include a user name in the repository URL. You can set the repository URL in the `repos` option with `base::options()` as usual, or you can use `repo_add()`.

For testing purposes pak includes a web app that creates an authenticated proxy to CRAN. This is how to run the proxy in a subprocess:

```
repo <- webfakes::new_app_process(pak:::auth_proxy_app())
repo$url()
#> [1] "http://127.0.0.1:59571/"
```

(This needs the webfakes and callr packages.)

Next, we configure the proxy as the main CRAN repository. The default username of the proxy is `"username"` and the default password is `"token"`. We want to replace the default CRAN repository with the proxy, so we name it `CRAN`:

```
repo_add(CRAN = repo$url(), username = "username")
repo_get()
#> x Did not find credentials for repo <http://username@127.0.0.1:59571/>, keyring
#> lookup failed (macos backend).
#> # A data frame: 6 x 7
#>   name        url         type  r_version bioc_version username has_password
#> * <chr>       <chr>       <chr> <chr>     <chr>        <chr>    <lgl>
#> 1 CRAN        http://usern~ cran  *         <NA>         username FALSE
#> 2 BioCsoft    https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 3 BioCann     https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 4 BioCexp     https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 5 BioCworkflows https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 6 BioCbooks   https://bioc~ bioc  4.4.2     3.20         <NA>     NA
```

Note that the output includes a `username` and a `has_password` column. These are only present if at least one configured repository needs authentication. `has_password` is `FALSE` here, because pak did not find the credentials for this repository.

`repo_get()` also displays a message if it cannot find the credentials for an authenticated repository.

Next we are going to store the credentials in a place where pak can find them.

Credential lookup:

pak can look up credentials from two sources:

1. The current user's `netrc` file.
2. The system credential store via the keyring package. pak comes with its own copy of the keyring package, you don't need to install it separately.

`netrc` *files:*
If the `NETRC` environment variable is set, pak uses its value to determine the location of the `netrc` file.
Otherwise pak looks for the `netrc` file in current user's home directory, at `~/.netrc`. On Windows it also looks for `~/_netrc` if the file starting with a dot does not exist.
If you create a `netrc` file, make sure that is only readable by you. E.g. on Unix run

```
chmod 600 ~/.netrc
```

`netrc` files are simple text files that can store passwords for multiple hosts. They may contain three types of tokens:

`machine <hostname>`*:*
A host name, without the protocol. Subsequent `login` and `password` tokens belong to this host, until another `machine` token is found, or the end of file.

`login <username>`*:*
User name. It must be preceded by a `machine` token.

`password <password>`*:*
Password. It must be preceded by a `machine` and a `login` token.
Whitespace is ignored in `netrc` files. You may include multiple tokens on the same line, or have one token per line. Here is an example:

```
machine myhost.mydomain.com login myuser password secret
machine myhost2.mydomain.com
login myuser
password secret
login anotheruser
password stillsecret
```

If you need to include whitespace in a password, put the password in double quotes.

The credential store:
pak uses the keyring package to query the system credential store (or an alternative keyring credential store) to find credentials for authenticated repositories. pak comes with a copy of the keyring package, so you don't need to install it separately.
To store a repository password in the system credential store use the `repo_auth_key_set()` function. If you want to use a non-default keyring backend, set the `keyring_backend` option. In this manual we will use the backend that stores secrets in environment variables. This is an ephemeral store that is destroyed when the R process terminates.
To continue our example from above:

```
options(keyring_backend = "env")
repo_auth_key_set(repo$url(), username = "username", password = "token")
```

Use `repo_auth_key_get()` to check that the key is properly set:

```
repo_auth_key_get(repo$url(), username = "username")
#> [1] "token"
```

`repo_get()` now does not show a warning message, and also sets the `has_password` column to `TRUE`, because pak could find the credentials for our CRAN proxy:

```
repo_get()
#> x Did not find credentials for repo <http://username@127.0.0.1:59571/>, keyring
#> lookup failed (macos backend).
#> # A data frame: 6 x 7
#>   name        url         type  r_version bioc_version username has_password
#> * <chr>       <chr>       <chr> <chr>     <chr>        <chr>    <lgl>
#> 1 CRAN        http://usern~ cran  *         <NA>         username FALSE
#> 2 BioCsoft    https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 3 BioCann     https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 4 BioCexp     https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 5 BioCworkflows https://bioc~ bioc  4.4.2     3.20         <NA>     NA
#> 6 BioCbooks   https://bioc~ bioc  4.4.2     3.20         <NA>     NA
```

Repo vs. host credentials:
pak handles credentials for repositories and hosts. A repository credential's key is a URL with a non-empty path:

```
https://repo.host.com/repos/repo1
```

A host credential's is an URL with an empty path:

```
https://repo.host.com
```

pak always looks for repository credentials first. If it does not find any credentials for a repository then it drops the path and looks for host credentials.
Because `netrc` files only store domain names and not URLs, they can only contain host credentials.

Testing:
To test that authentication works, use the `repo_status()` function:

```
repo_status()
#> x Did not find credentials for repo <http://username@127.0.0.1:59571/>, keyring
#> lookup failed (macos backend).
#> # A data frame: 12 x 12
#>   name  url   type  bioc_version username has_password platform path  r_version
#>   <chr> <chr> <chr> <chr>        <chr>    <lgl>        <chr>    <chr> <chr>
#>  1 CRAN  http~ cran  <NA>         username FALSE        source   src/~ 4.4
#>  2 CRAN  http~ cran  <NA>         username FALSE        aarch64~ bin/~ 4.4
#>  3 BioC~ http~ bioc  3.20         <NA>     NA           source   src/~ 4.4
#>  4 BioC~ http~ bioc  3.20         <NA>     NA           aarch64~ bin/~ 4.4
#>  5 BioC~ http~ bioc  3.20         <NA>     NA           source   src/~ 4.4
#>  6 BioC~ http~ bioc  3.20         <NA>     NA           aarch64~ bin/~ 4.4
#>  7 BioC~ http~ bioc  3.20         <NA>     NA           source   src/~ 4.4
#>  8 BioC~ http~ bioc  3.20         <NA>     NA           aarch64~ bin/~ 4.4
#>  9 BioC~ http~ bioc  3.20         <NA>     NA           source   src/~ 4.4
#> 10 BioC~ http~ bioc  3.20         <NA>     NA           aarch64~ bin/~ 4.4
```

```
#> 11 BioC~ http~ bioc  3.20         <NA>     NA           source   src/~ 4.4
#> 12 BioC~ http~ bioc  3.20         <NA>     NA           aarch64~ bin/~ 4.4
#> # i 3 more variables: ok <lgl>, ping <dbl>, error <list>
```

The output of `repo_status()` has extra columns, compared to `repo_get()`, and it also has a separate row for each platform. If everything works, then the `has_password` column is `TRUE` for authenticated repositories, and the `ok` column is `TRUE` if `repo_status()` was able to perform an (authenticated) HTTP HEAD request to the metadata file of a platform in a repository.

If you need even more information about repo authentication, e.g. because `repo_status()` shows some failures, then use the `repo_auth()` function:

```
repo_auth()
#> v Found credentials for repo <http://username@127.0.0.1:59571/> (keyring:env).
#> # A data frame: 1 x 11
#>   name  url          type  r_version bioc_version username has_password auth_domains auth_domain
#> * <chr> <chr>        <chr> <chr>    <chr>     <chr>    <lgl>        <I<list>>   <chr>
#> 1 CRAN  http://username~ cran  *      <NA>      username TRUE         <chr [4]>   http://127~
#> # i 2 more variables: auth_source <chr>, auth_error <chr>
```

The output of `repo_auth()` has the following extra columns:

- `auth_domains`: these are the URLs that pak tries to use as the `service` when looking for credentials in the keyring. For our proxy, it tries these URLs:

  ```
  repo_auth()$auth_domains
  #> v Found credentials for repo <http://username@127.0.0.1:59571/> (keyring:env).
  #> [[1]]
  #> [1] "http://username@127.0.0.1:59571/" "http://127.0.0.1:59571/"
  #> [3] "http://username@127.0.0.1:59571"  "http://127.0.0.1:59571"
  ```

- `auth_domain`: one of `auth_domains`, the URL for which pak found credentials in the keyring. If no credentials were found, then this is `NA`.
- `auth_source`: a short string that explains where pak found the credentials (or `NA` if not credentials were found). For examples `netrc` means the user's `netrc` file, and `keyring:macos` means the macOS system credential store.
- `auth_error`: `NA` for successful credential search, otherwise a short error message on why the search failed. Typically it would fail is the credentials are not in the credential store.

Usage:

Once you set up your authenticated repositories, and stored the required passwords in the system credential store, you can use them like any other repository. Operations that need authentication will always include reassuring messages for successful authentications, and warning messages for failed ones. Function calls that do not perform any HTTP requests, e.g. because they list cached data, do not display such messages.

```
meta_update()
#> Checking for 15 new metadata files
#> ! Cannot find credentials for URL <http://username@127.0.0.1:59571//bin/macosx/big-sur-arm64/cont
#> failed. Keyring backend: "macos".
#> Checking for 15 new metadata files
x Did not find credentials for repo <http://username@127.0.0.1:59571/>, keyring
#> lookup failed (macos backend).
```

```
#>
#> i R 4.4 aarch64-apple-darwin20 packages are missing from CRAN: Failed to connect to 127.0.0.1 port 5
#> i source packages are missing from CRAN: Failed to connect to 127.0.0.1 port 59571 after 0 ms: Couldr
#> i Updating metadata database
#> v Updating metadata database ... done

meta_list()
#> # A data frame: 50,701 x 33
#>   package   version depends suggests license md5sum sha256sum needscompilation
#>     <chr>      <chr>   <chr>   <chr>    <chr>   <chr>  <chr>     <chr>
#>  1 A3         1.0.0   R (>= ~ randomF~ GPL (>~ 929a4~ "\n     ~ no
#>  2 AATtools   0.0.3   R (>= ~ <NA>     GPL-3   de2ec~ "\n     ~ no
#>  3 ABACUS     1.0.0   R (>= ~ rmarkdo~ GPL-3   28795~ "\n     ~ no
#>  4 ABC.RAP    0.9.0   R (>= ~ knitr, ~ GPL-3   0158e~ "\n     ~ no
#>  5 ABCanalys~ 1.2.1   R (>= ~ <NA>     GPL-3   4cbe1~ "\n     ~ no
#>  6 ABCoptim   0.15.0  <NA>    testtha~ MIT + ~ a294d~ "\n     ~ yes
#>  7 ABCp2      1.2     MASS    <NA>     GPL-2   d049b~  <NA>     no
#>  8 ABHgenoty~ 1.0.1   <NA>    knitr, ~ GPL-3   fce25~ "\n     ~ no
#>  9 ABM        0.4.3   <NA>    <NA>     GPL (>~ 7aaae~ "\n     ~ yes
#> 10 ABPS       0.3     <NA>    testthat GPL (>~ d3f00~ "\n     ~ no
#> # i 50,691 more rows
#> # i 25 more variables: imports <chr>, linkingto <chr>, archs <chr>,
#> #   enhances <chr>, license_restricts_use <chr>, priority <chr>, os_type <chr>,
#> #   license_is_foss <chr>, repodir <chr>, rversion <chr>, platform <chr>,
#> #   ref <chr>, type <chr>, direct <lgl>, status <chr>, target <chr>,
#> #   mirror <chr>, sources <list>, filesize <int>, sha256 <chr>, sysreqs <chr>,
#> #   built <chr>, published <dttm>, deps <list>, path <chr>
```

E.g. here `meta_update()` outputs an authentication message, but `meta_list()` does not.

See Also

Other authenticated repositories: `repo_auth()`, `repo_auth_key_get()`

`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 sufficient 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 `pkg = 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?:
Since version 1.0.0 renv has official support for using pak. This needs to be enabled with the `renv.config.pak.enabled` option or the `RENV_CONFIG_PAK_ENABLED` environment variable set to `TRUE`. For more information see the renv documentation.

`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 corresponds 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)
```

`upgrade = 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 PPM:
pak can use PPM (Posit Public Package Manager) 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-contained 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
FreeBSD 13.x or later	x86_64	R 4.4.x
OpenBSD 7.4, 7.5, 7.6	x86_64	R 4.2.x (7.4, 7.5), R 4.4.x (7.6)
NetBSD 10.0	x86_64	R 4.4.x
DragonFly BSD 6.4	x86_64	R 4.3.x

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 pak.
- `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. `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()`, `pkg_sysreqs()`

---

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 recommended 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 recommended 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 recommended 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

[Deprecated]

Note that these functions are now *deprecated*, in favor of `pkg_sysreqs()` and the `sysreqs_*` functions, which are more powerful, as they work for all package sources (packages at Github, GitLab, URLs, etc.) and they have more detailed output.

Instead of

```
pak::pkg_system_requirement("curl")
```

call

```
pak::pkg_sysreqs("curl")$install_scripts
```

and the equivalent of

```
pak::local_system_requirements()
```

is

```
pak::pkg_sysreqs("local::.", dependencies = TRUE)$install_script
```

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", "centos", "redhat". See supported_os_versions() for all full list of supported operating systems. If NULL, the default, these will be looked up.
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.

Details

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).

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 recommended 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

```
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

```
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()`, `pkg_sysreqs()`

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 default 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:

`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 reload 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 = TRUE`.

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,
- platform the package was built on, and the current platform,
- the current library path,
- versions of dependencies,
- whether dependencies can be loaded.

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. `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_sysreqs()

---

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_sysreqs()

---

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: 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_sysreqs()

---

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 recommended 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_sysreqs()

---

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, * 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_sysreqs()

---

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.
...	Arguments passed on to pkgsearch::pkg_search from Where to start listing the results, for pagination. size The number of results to list.

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. 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(), pkg_sysreqs()

pkg_sysreqs *Calculate system requirements of one of more packages*

Description

Calculate system requirements of one of more packages

Usage

```
pkg_sysreqs(pkg, upgrade = TRUE, dependencies = NA, sysreqs_platform = 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.
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.
sysreqs_platform	System requirements platform. If NULL, then the sysreqs_platform man_config_link("configuration option") is used, which defaults to the current platform. Set this option if to one of platforms() if .packageName fails to correctly detect your platform or if you want to see the system requirements for a different platform.

Value

List with entries:

- os: character string. Operating system.
- distribution: character string. Linux distribution, NA if the OS is not Linux.
- version: character string. Distribution version, NA is the OS is not Linux.
- pre_install: character vector. Commands to run before the installation of system packages.
- install_scripts: character vector. Commands to run to install the system packages.
- post_install: character vector. Commands to run after the installation of system packages.
- packages: data frame. Information about the system packages that are needed. It has columns:

- sysreq: string, cross-platform name of the system requirement.
- packages: list column of character vectors. The names of the R packages that have this system requirement.
- pre_install: list column of character vectors. Commands run before the package installation for this system requirement.
- system_packages: list column of character vectors. Names of system packages to install.
- post_install: list column of character vectors. Commands run after the package installation for this system requirement.

See Also

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

Other system requirements functions: sysreqs_check_installed(), sysreqs_db_list(), sysreqs_db_match(), sysreqs_db_update(), sysreqs_is_supported(), sysreqs_list_system_packages(), sysreqs_platforms()

---

ppm_has_binaries *Does PPM build binary packages for the current platform?*

Description

Does PPM build binary packages for the current platform?

Usage

```
ppm_has_binaries()
```

Value

TRUE or FALSE.

See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/.

Other PPM functions: ppm_platforms(), ppm_r_versions(), ppm_repo_url(), ppm_snapshots()

Examples

```
system_r_platform()
ppm_has_binaries()
```

ppm_platforms *List all platforms supported by Posit Package Manager (PPM)*

Description

List all platforms supported by Posit Package Manager (PPM)

Usage

```
ppm_platforms()
```

Value

Data frame with columns:

- `name`: platform name, this is essentially an identifier,
- `os`: operating system, `linux`, `windows` or `macOS` currently,
- `binary_url`: the URL segment of the binary repository URL of this platform, see `ppm_snapshots()`.
- `distribution`: for Linux platforms the name of the distribution,
- `release`: for Linux platforms, the name of the release,
- `binaries`: whether PPM builds binaries for this platform.

See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/.

Other PPM functions: `ppm_has_binaries()`, `ppm_r_versions()`, `ppm_repo_url()`, `ppm_snapshots()`

Examples

```
ppm_platforms()
```

ppm_repo_url *Returns the current Posit Package Manager (PPM) repository URL*

Description

Returns the current Posit Package Manager (PPM) repository URL

Usage

```
ppm_repo_url()
```

Details

This URL has the form {base}/{repo}, e.g. https://packagemanager.posit.co/all.

To configure a hosted PPM instance, set the PKGCACHE_PPM_URL environment variable to the base URL (e.g. https://packagemanager.posit.co).

To use repo_add() with PPM snapshots, you may also set the PKGCACHE_PPM_REPO environment variable to the name of the default repository.

On Linux, instead of setting these environment variables, you can also add a PPM repository to the repos option, see base::options(). In the environment variables are not set, then ppm_repo_url() will try extract the PPM base URL and repository name from this option.

If the PKGCACHE_PPM_URL environment variable is not set, and the repos option does not contain a PPM URL (on Linux), then pak uses the public PPM instance at https://packagemanager.posit.co, with the cran repository.

Value

String scalar, the repository URL of the configured PPM instance. If no PPM instance is configured, then the URL of the Posit Public Package Manager instance. It includes the repository name, e.g. https://packagemanager.posit.co/all.

See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/.

repo_resolve() and repo_add() to find and configure PPM snapshots.

Other PPM functions: ppm_has_binaries(), ppm_platforms(), ppm_r_versions(), ppm_snapshots()

Examples

```
ppm_repo_url()
```

---

ppm_r_versions *List all R versions supported by Posit Package Manager (PPM)*

Description

List all R versions supported by Posit Package Manager (PPM)

Usage

```
ppm_r_versions()
```

Value

Data frame with columns:

- r_version: minor R versions, i.e. version numbers containing the first two components of R versions supported by this PPM instance.

See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/.

Other PPM functions: ppm_has_binaries(), ppm_platforms(), ppm_repo_url(), ppm_snapshots()

Examples

```
ppm_r_versions()
```

`ppm_snapshots` *List all available Posit Package Manager (PPM) snapshots*

Description

List all available Posit Package Manager (PPM) snapshots

Usage

```
ppm_snapshots()
```

Details

The repository URL of a snapshot has the following form on Windows:

```
{base}/{repo}/{id}
```

where `{base}` is the base URL for PPM (see ppm_repo_url()) and `{id}` is either the date or id of the snapshot, or `latest` for the latest snapshot. E.g. these are equivalent:

```
https://packagemanager.posit.co/cran/5
https://packagemanager.posit.co/cran/2017-10-10
```

On a Linux distribution that has PPM support, the repository URL that contains the binary packages looks like this:

```
{base}/{repo}/__linux__/{binary_url}/{id}
```

where `{id}` is as before, and `{binary_url}` is a code name for a release of a supported Linux distribution. See the `binary_url` column of the result of ppm_platforms() for these code names.

Value

Data frame with two columns:

- `date`: the time the snapshot was taken, a `POSIXct` vector,
- `id`: integer id of the snapshot, this can be used in the repository URL.

See Also

The 'pkgcache and Posit Package Manager on Linux' article at https://r-lib.github.io/pkgcache/.

Other PPM functions: ppm_has_binaries(), ppm_platforms(), ppm_r_versions(), ppm_repo_url()

Examples

```
ppm_snapshots()
```

---

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, username = NULL)

repo_resolve(spec, username = NULL)
```

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.
username	User name to set, for authenticated repositories, see repo_auth().
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`

- `PPM@latest`, PPM (Posit Package Manager, formerly RStudio Package Manager), the latest snapshot.
- `PPM@<date>`, PPM (Posit Package Manager, formerly RStudio Package Manager) snapshot, at the specified date.
- `PPM@<package>-<version>` PPM snapshot, for the day after the release of `<version>` of `<package>`.
- `PPM@R-<version>` PPM snapshot, for the day after R `<version>` was released.

Still works for dates starting from 2017-10-10, but now deprecated, because MRAN is discontinued:

- `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 PPM at https://packagemanager.posit.co/client/#/.
- The `RSPM@` prefix is still supported and treated the same way as `PPM@`.
- The MRAN service is now retired, see `https://techcommunity.microsoft.com/blog/azuresqlblog/microsoft-` for details.
- `MRAN@...` repository specifications now resolve to PPM, but note that PPM snapshots are only available from 2017-10-10. See more about this at https://posit.co/blog/migrating-from-mran-to-posit-pack
- All dates (or times) can be specified in the ISO 8601 format.
- If PPM 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 PPM snapshot will trigger an error.
- Unknown R or package versions will trigger an error.

Examples

```
repo_add(PPMdplyr100 = "PPM@dplyr-1.0.0")
repo_get()

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

repo_resolve("PPM@dplyr-1.0.0")

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

See Also

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

`repo_auth` *Authenticated repositories*

Description

pak supports HTTP basic authentication when interacting with CRAN-like repositories. To use authentication, include a username in the repo URL:

```
https://<username>@<repo-host>/<repo-path>
```

Usage

```
repo_auth(
  r_version = getRversion(),
  bioc = NULL,
  cran_mirror = NULL,
  check_credentials = TRUE
)
```

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.
`check_credentials`	Whether to check that credentials are available for authenticated repositories.

Details

pak will look up the password for this url and username from the the user's `.netrc` file and from the system credential store using the keyring package.

'.netrc" files:

First pak searches in the `.netrc` file. If the `NETRC` environment variable is set, pak uses its value to determine the location of the `netrc` file.

Otherwise pak looks for the `netrc` file in current user's home directory, at `~/.netrc`. On Windows it also looks for `~/_netrc` if the file starting with a dot does not exist.

If you create a `netrc` file, make sure that is only readable by you. E.g. on Unix run

```
chmod 600 ~/.netrc
```

`netrc` files are simple text files that can store passwords for multiple hosts. They may contain three types of tokens:

`machine <hostname>`:
A host name, without the protocol. Subsequent `login` and `password` tokens belong to this host, until another `machine` token is found, or the end of file.

`login <username>`:
User name. It must be preceded by a `machine` token.

`password <password>`:
Password. It must be preceded by a `machine` and a `login` token.
Whitespace is ignored in `netrc` files. You may include multiple tokens on the same line, or have one token per line. Here is an example:

```
machine myhost.mydomain.com login myuser password secret
machine myhost2.mydomain.com
login myuser
password secret
login anotheruser
password stillsecret
```

If you need to include whitespace in a password, put the password in double quotes.

The system credential store:
pak currently supports the following keyring backends:

- Windows credential store,
- macOS Keychain,
- Linux Secret Service via libsecret, if built with libsecret support,
- environment variables.

For the URL above it tries the following keyring keys, in this order:

```
https://<username>@repo-host/<repo-path>
https://repo-host/<repo-path>
https://<username>@repo-host
https://repo-host
```

To add an authenticated repository use `repo_add()` with the `username` argument. Alternatively, you can set the `repos` option directly using `base::options()` and including the username in the repository URL.
`repo_auth()` lists authentication information for all configured repositories.

Value

Data frame with columns:

- all columns from the output of `repo_get()`,
- `auth_domains`: authentication domains. pak tries to find the credentials for these domains, until the search is successful or all domains fail.
- `auth_domain`: if the credential lookup is successful, then this is the authentication domain that was used to get the credentials.
- `auth_source`: where the credentials were found. E.g. `keyring:<backend>` means it was in the default macos keyring.
- `auth_error`: for failed credential searches this is the description of why the search failed. E.g. maybe the keyring package is not installed, or pak found no credentials for any of the authentication domains.

See Also

Authenticated repositories.

Other authenticated repositories: Authenticated repositories, repo_auth_key_get()

---

repo_auth_key_get *Query or set repository password in the system credential store*

Description

Use pak's keyring functions to query or set a repository password in the system credential store.

Usage

```
repo_auth_key_get(url, username = NULL)

repo_auth_key_set(url, password, username = NULL)

repo_auth_unlock(keyring_password)
```

Arguments

url	Repository URL. It may contain a username, in which case username may be NULL.
username	User name, if it is not included in url.
password	Password (key) to set.
keyring_password	Password to unlock the keyring.

Details

repo_auth_key_get() retrieves a password from the default keyring. It errors if it cannot find the credentials for url.

repo_auth_key_set() adds or updates a password in the system credential store.

repo_auth_unlock() unlocks the default keyring, if it is locked. You might need this if the keyring is locked. If you are using ecrypted files to store the keys, then you typically need to call this function in each session. You typically don't need to do that if you are using the native Windows, macOS or Linux (Secret Service) backends.

Value

repo_auth_key_get() returns a single string, the repository password.

See Also

Other authenticated repositories: Authenticated repositories, repo_auth()

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 = NULL, 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.

Value

repo_get() returns a data frame with columns:

- name: repository name. Names are informational only.
- url: repository URL.
- type: repository type. This is also informational, currently it can be cran for CRAN, bioc for a Bioconductor repository, and cranlike: for other repositories.
- r_version: R version that is supposed to be used with this repository. This is only set for Bioconductor repositories. It is * for others. This is also informational, and not used when retrieving the package metadata.
- bioc_version: Bioconductor version. Only set for Bioconductor repositories, and it is NA for others.
- username: user name, for authenticated repositories.
- has_password: whether repo_get() could find the password for this repository. Call repo_auth() for more information if the credential lookup failed.

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 = NULL,
  cran_mirror = NULL
)

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

Arguments

platforms	Platforms to use, default is the current platform, plus source packages, via the pkg.platforms option.
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. Defaults to the pkg.use_bioconductor option.
cran_mirror	The CRAN mirror to use. Defaults to the pkg.cran_mirror option.

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 pak. 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.
- `username`: Included if at least one repository is authenticated. `NA_character_` for repositories without authentication. See `repo_auth()`.
- `has_password`: `TRUE` is the function could retrieve the password for the authenticated repository. It is `NA` for repositories without authentication. This column is included only if at least one repository has authentication. See `repo_auth()`.
- `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()`

scan_deps *Scan R code for dependent packages*

Description

Scan all R files of a project or directory for packages used within them. It parses R code to find `library(package)`, `package::func()`, and similar calls that imply package dependencies. See details below.

Usage

```
scan_deps(path = NULL, root = NULL)
```

Arguments

`path`	Files and/or directories to scan. Defaults to the current project, detected by finding the first parent directory of the current working directory, that contains a file or directory called DESCRIPTION, .git, .Rproj.user, renv.lock, or renv. (Note that this is different from `renv::dependencies()`, which only scans the current working directory by default!) If `path` is not `NULL`, then only the specified files and directories are scanned, the directories recursively. In this case the `root` argument is used as the project root, to find `.gitignore` and `.renvignore` files. All entries of `path` must be within the `root`, the project root.
`root`	The root directory of the project. It is used to find the `.gitignore` and `.renvignore` files. By default the same algorithm is used to detect this as for `path`. If `path` is specified and it is not within the detected or specified `root`, `scan_path()` throws an error.

Value

Data frame with columns:

- `path`: Path to the file in which the dependencies was found.
- `package`: Detected package dependency. Typically a package name, but it can also be a package reference, e.g. a package from GitHub.
- `type`: Dependency type. It is `"prod"`, `"test"` or `"dev"`. See 'Dependency types' below.
- `code`: The piece of code the dependency was extracted from.
- `start_row`: Start row of the code the dependency was extracted from.
- `start_column`: Start column of the code the dependency was extracted from.
- `start_byte`: Start byte of the code the dependency was extracted from.

Note the data frame may contain the same package multiple times, if it was detected multiple times, e.g. multiple `library()` calls load the same package.

Detected dependencies

`scan_deps()` detects package dependencies from these R expressions:

- `library()`, `require()`, `loadNamespace()` and `requireNamespace` calls.
- `::` and `:::` operators.
- Any of the calls in this list in R code from R markdown or quarto `R` and `Rscript` (case insensitive) code blocks or inline R code.
- A dependency on the methods package is inferred from finding `setClass()` and/or `setGeneric()` calls.
- `xfun::pkg_attach()` and `xfun::pkg_attach2()` calls.
- `pacman::p_load()` calls.
- `modules::import()` and `modules::module()` calls.
- `import::from()`, `import::here()` and `import::into()` calls.
- `box::use()` calls.
- `targets::tar_option_set(packages = ...)` calls.
- Any of the calls in this list in R code from `glue::glue()` strings.
- A dependency on the svglite package is inferred from `ggplot2::ggsave()` calls saving `.svg` files.
- Dependencies from `parsnip::set_engine()` calls, the default engine to package mapping is:
  - `"glm"` -> stats,
  - `"glmnet"` -> glmnet,
  - `"keras"` -> keras,
  - `"kknn"` -> kknn,
  - `"nnet"` -> nnet,
  - `"rpart"` -> rpart,
  - `"spark"` -> sparklyr,
  - `"stan"` -> rstanarm. You can override the default mapping by setting the `renv.parsnip.engines` option to a named list.
- A dependency on the xml2 package is inferred from using the "Junit" reporter (`JunitReporter`) from the testthat package.
- A dependency on the ragg package is inferred from setting the default knitr device (`dev` option) to `"ragg_png"`.
- A dependency on the hexbin package is inferred from using `ggplot2::geom_hex()`.
- A custom symbol name to package name mapping can be defined in the `renv.dependencies.database` option. This must be a named list of named lists, where the outer names are package names, the inner names are function or object names, and the values are package names. E.g.

```
options(renv.dependencies.database = list(
  ggplot2 = list(geom_hex = "hexbin"),
  testthat = list(JunitReporter = "xml2")
))
```

Dependency types

scan_deps() classifies package dependencies into three groups, based on which files they were found:

- Production dependencies: "prod".
- Test dependencies: "test".
- Development dependencies: "dev".

Examples

```
scan_deps("myproject")
scan_deps("myproject")[]
```

---

sysreqs_check_installed

Check if installed packages have all their system requirements

---

Description

sysreqs_check_installed() checks if the system requirements of all packages (or a subset of packages) are installed.

sysreqs_fix_installed() installs the missing system packages.

Usage

```
sysreqs_check_installed(packages = NULL, library = .libPaths()[1])
sysreqs_fix_installed(packages = NULL, library = .libPaths()[1])
```

Arguments

packages	If not NULL, then only these packages are checked. If a package in packages is not installed, then pak throws a warning.
library	Library or libraries to check.

Details

These functions use the sysreqs_platform configuration option, see man_config_link("Configuration"). Set this if pak does not detect your platform correctly.

Value

Data frame with a custom print and format method, and a `pkg_sysreqs_check_result` class. Its columns are:

- `system_package`: string, name of the required system package.
- `installed`: logical, whether the system package is correctly installed.
- `packages`: list column of character vectors. The names of the installed R packages that need this system package.
- `pre_install`: list column of character vectors. Commands to run before the installation of the the system package.
- `post_install`: list column of character vectors. Commands to run after the installation of the system package.

The data frame also have two attributes with additional data:

- `sysreqs_records`: the raw system requirements records, and
- `system_packages`: the list of the installed system packages.

`sysreqs_fix_packages()` returns the same value, but invisibly.

See Also

Other system requirements functions: `pkg_sysreqs()`, `sysreqs_db_list()`, `sysreqs_db_match()`, `sysreqs_db_update()`, `sysreqs_is_supported()`, `sysreqs_list_system_packages()`, `sysreqs_platforms()`

Examples

```
# This only works on supported platforms
sysreqs_check_installed()
```

---

sysreqs_db_list *List contents of the system requirements DB, for a platform*

Description

It also tries to update the system dependency database, if it is outdated. (I.e. older than allowed in the `metadata_update_after` `man_config_link("configuration option")`.

Usage

```
sysreqs_db_list(sysreqs_platform = NULL)
```

Arguments

`sysreqs_platform`
: System requirements platform. If `NULL`, then the `sysreqs_platform` `man_config_link("configuratio option")` is used, which defaults to the current platform. Set this option if `.packageName` does not detect your platform correctly.

Value

Data frame with columns:

- `name`: cross platform system dependency name in the database.
- `patterns`: one or more regular expressions to match to `SystemRequirements` fields.
- `packages`: one or more system package names to install.
- `pre_install`: command(s) to run before installing the packages.
- `post_install`:: command(s) to run after installing the packages.

See Also

Other system requirements functions: `pkg_sysreqs()`, `sysreqs_check_installed()`, `sysreqs_db_match()`, `sysreqs_db_update()`, `sysreqs_is_supported()`, `sysreqs_list_system_packages()`, `sysreqs_platforms()`

Examples

```
sysreqs_db_list(sysreqs_platform = "ubuntu-22.04")
```

`sysreqs_db_match` *Match system requirement descriptions to the database*

Description

In the usual workflow pak matches the `SystemRequirements` fields of the `DESCRIPTION` files to the database.

Usage

```
sysreqs_db_match(specs, sysreqs_platform = NULL)
```

Arguments

`specs`	Character vector of system requirements descriptions.
`sysreqs_platform`	System requirements platform. If `NULL`, then the `sysreqs_platform` `man_config_link("configuratio option")` is used, which defaults to the current platform. Set this option if `.packageName` does not detect your platform correctly.

Details

The `sysreqs_db_match()` function lets you match any string, and it is mainly useful for debugging.

Value

Data frame with columns:

- spec: the input specs.
- sysreq: name of the system library or tool.
- packages: system packages, list column of character vectors. Rarely it can be an empty string, e.g. if a pre_install script performs the installation.
- pre_install: list column of character vectors. Shell script(s) to run before the installation.
- post_install: list column of character vectors. Shell script(s) to run after the installation.

See Also

Other system requirements functions: pkg_sysreqs(), sysreqs_check_installed(), sysreqs_db_list(), sysreqs_db_update(), sysreqs_is_supported(), sysreqs_list_system_packages(), sysreqs_platforms()

Examples

```
sysreqs_db_match(
  c("Needs libcurl", "Java, libssl"),
  sysreqs_platform = "ubuntu-22.04"
)
```

---

sysreqs_db_update *Update the cached copy of the system requirements database*

Description

Update the cached copy of the system requirements database

Usage

```
sysreqs_db_update()
```

Details

If the the cached copy is recent, then no update is attempted. See the metadata_update_after man_config_link("configuration option").

See Also

Other system requirements functions: pkg_sysreqs(), sysreqs_check_installed(), sysreqs_db_list(), sysreqs_db_match(), sysreqs_is_supported(), sysreqs_list_system_packages(), sysreqs_platforms()

sysreqs_is_supported *Check if a platform has system requirements support*

Description

Check if a platform has system requirements support

Usage

```
sysreqs_is_supported(sysreqs_platform = NULL)
```

Arguments

sysreqs_platform	System requirements platform. If NULL, then the sysreqs_platform man_config_link("configuratio option") is used, which defaults to the current platform. Set this option if .packageName does not detect your platform correctly.

Value

Logical scalar.

See Also

The sysreqs_platform man_config_link("configuration option").

Other system requirements functions: pkg_sysreqs(), sysreqs_check_installed(), sysreqs_db_list(), sysreqs_db_match(), sysreqs_db_update(), sysreqs_list_system_packages(), sysreqs_platforms()

Examples

```
sysreqs_is_supported()
```

sysreqs_list_system_packages *List installed system packages*

Description

List installed system packages

Usage

```
sysreqs_list_system_packages()
```

Details

This function uses the `sysreqs_platform` configuration option, see `man_config_link("Configuration")`. Set this if pak does not detect your platform correctly.

Value

Data frame with columns:

- `status`. two or three characters, the notation of `dpkg` on Debian based systems. `"ii"` means the package is correctly installed. On `RPM` based systems it is always `"ii"` currently.
- `package`: name of the system package.
- `version`: installed version of the system package.
- `capabilities`: list column of character vectors, the capabilities provided by the package.

See Also

Other system requirements functions: `pkg_sysreqs()`, `sysreqs_check_installed()`, `sysreqs_db_list()`, `sysreqs_db_match()`, `sysreqs_db_update()`, `sysreqs_is_supported()`, `sysreqs_platforms()`

Examples

```
sysreqs_list_system_packages()[1:10,]
```

`sysreqs_platforms` *List platforms with system requirements support*

Description

List platforms with system requirements support

Usage

```
sysreqs_platforms()
```

Value

Data frame with columns:

- `name`: human readable OS name.
- `os`: OS name, e.g. `linux`.
- `distribution`: OS id, e.g. `ubuntu` or `redhat`.
- `version`: distribution version. A star means that all versions are supported, that are also supported by the vendor.
- `update_command`: command to run to update the system package metadata.
- `install_command`: command to run to install packages.
- `query_command`: name of the tool to use to query system package information.

See Also

Other system requirements functions: `pkg_sysreqs()`, `sysreqs_check_installed()`, `sysreqs_db_list()`, `sysreqs_db_match()`, `sysreqs_db_update()`, `sysreqs_is_supported()`, `sysreqs_list_system_packages()`

Examples

```
sysreqs_platforms()
```

`System requirements` *System requirements*

Description

pak takes care of your system requirements.

Introduction

Many R packages need external software to be present on the machine, otherwise they do not work, or not even load. For example the RPostgres R package uses the PostgreSQL client library, and by default dynamically links to it on Linux systems. This means that you (or the administrators of your system) need to install this library, typically in the form of a system package: `libpq-dev` on Ubuntu and Debian systems, or `postgresql-server-devel` or `postgresql-devel` on RedHat, Fedora, etc. systems.

The good news is that pak helps you with this: - it looks up the required system packages when installing R packages, - it checks if the required system packages are installed, and - it installs them automatically, if you are a superuser, or you can use password-less `sudo` to start a superuser shell.

In addition, pak also has some functions to query system requirements and system packages.

Requirements, supported platforms

Call `pak::sysreqs_platforms()` to list all platforms that support system requirements:

```
pak::sysreqs_platforms()
```

Call `pak::sysreqs_is_supported()` to see if your system is supported:

```
pak::sysreqs_is_supported()
```

This vignette was built on Ubuntu 22.04.2 LTS, which is a platform pak does support. So in the following you will see the output of the code.

R package installation

If you are using pak as a superuser, on a supported platform, then pak will look up system requirements, and install the missing ones. Here is an example:

```
pak::pkg_install("RPostgres")
```

Running R as a regular user:

If you don't want to use R as the superuser, but you can set up `sudo` without a password, that works as well. pak will automatically detect the password-less `sudo` capability, and use it to install system packages, as needed.

If you run R as a regular (not root) user, and password-less `sudo` is not available, then pak will print the system requirements, but it will not try to install or update them. If you are installing source packages that need to link to system libraries, then their installation will probably fail, until you install these system packages. If you are installing binary R packages, then the installation typically succeeds, but you won't be able to load these packages into R, until you install the required system packages. Here is an example, on a system that does not have the required system package installed for RPostgres. If you are installing a source R package, the installation already fails:

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

On the other hand, if you are installing binary packages, e.g. from the Posit Package Manager, then the installation typically succeeds, but then loading the package fails:

```
pak::pkg_install("RPostgres")
library(RPostgres)
```

Query system requirements without installation

If you only want to query system requirements, without installing any packages, use the `pkg_sysreqs()` function. This is similar to `pkg_deps()` but in addition to looking up package dependencies, it also looks up system dependencies, and only reports the latter:

```
pak::pkg_sysreqs(c("curl", "xml2", "devtools", "CHRONOS"))
```

See the manual of `pkg_sysreqs()` to see how to programmatically extract information from its return value.

Other queries

In addition to the automatic system package lookup and installation, pak also has some other functions to help you with system dependencies. The `sysreqs_db_list()` function lists all system requirements pak knows about.

```
pak::sysreqs_db_list()
```

`sysreqs_db_match()` manually matches `SystemRequirements` fields against these system requirements:

```
sq <- pak::sysreqs_db_match("Needs libcurl and also Java.")
sq
```

```
sq[[1]]$packages
```

You can also use it to query system requirements for other platforms:

```
sqrhel9 <- pak::sysreqs_db_match("Needs libcurl and also Java.", "redhat-9")
sqrhel9
```

```
sqrhel9[[1]]$packages
```

`sysreqs_list_system_packages()` is a cross-platform way of listing all installed system packages and capabilities:

```
pak::sysreqs_list_system_packages()
```

`sysreqs_check_installed()` is a handy function that checks if all system requirements are installed for some or all R packages that are installed in your library:

```
pak::sysreqs_check_installed()
```

`sysreqs_fix_installed()` goes one step further and also tries to install the missing system requirements.

Build-time and run-time dependencies

The system requirements database that pak uses does not currently differentiate between build-time and run-time dependencies. A build-time dependency is a system package that you need when *installing* an R package from source. A run-time dependency is a system package that you need when *using* an R package. Most Linux distribution create (at least) two packages for each software library: a runtime package and a development package. For an R package that uses such a software library, the runtime package is a run-time dependency and the development package is a build-time dependency. However, pak does not currently know the difference between build-time and run-time dependencies, and it will install both types of dependencies, always. This means that pak usually installs system packages that are not strictly necessary. These are typically development packages of libraries, i.e. header files, and typically do not cause any issues. If you are short on disk space, then you can try removing them.

How it works

pak uses the database of system requirements at https://github.com/rstudio/r-system-requirements. It has its own copy of the database embedded into the package, and it also tries to download updated versions of the database from GitHub, if its current copy is older than one day. You can explicitly update the database from GitHub using the `sysreqs_db_update()` function.

For CRAN packages, it downloads the `SystemRequirements` fields from `https://cran.r-pkg.org/metadata`, which is a database updated daily. For Bioconductor packages, it downloads then from GitHub. (We are planning on moving CRAN database to GitHub as well.)

For packages sources that require pak to obtain a package DESCRIPTION file (e.g. github::, git::, etc.), pak obtains SystemRequirements directly from the DESCRIPTION file.

Once having the SystemRequirements fields, pak matches them to the database, to obtain the canonized list of system requirements.

Then pak queries the local platform, to see the exact system packages needed. It also queries the installed system packages, to avoid trying to install system packages that are already installed.

Configuration

There are several pak configuration options you can use to adjust how system requirements are handled. We will list some of them here, please see the options with a sysreqs prefix in the ?pak-config manual page for a complete and current list.

- sysreqs: whether to install system requirements. The default is TRUE if the platform is supported and the user can install system packages, either because it is the superuser, or via sudo. If it is FALSE (or the user cannot install system packages), but the platform is supported, system requirements are printed, but not installed.
- sysreqs_db_update: whether to try to update the system requirements database from GitHub.
- sysreqs_db_update_timeout: timeout for the system requirements update from GitHub.
- sysreqs_dry_run: if TRUE then pak only prints the install commands, but does not actually run them.
- sysreqs_platform: the platform name to use for determining system requirements. Defaults to the current platform. If you are using a Linux distribution that is compatible with some distribution that pak supports, then you can set this option manually. E.g. Ubuntu-based distros can set it to ubuntu-22.04, or the appropriate Ubuntu version.
- sysreqs_sudo: whether to use sudo to install system packages. If this is not set, then pak tries to auto-detect if sudo is needed or not.
- sysreqs_update: whether to try to update system packages that are already installed. pak does not know which version of a system package is required, and it does not try to update system packages by default. If you think that you need newer system packages, then you can set this option to TRUE.
- sysreqs_verbose: whether to print the output of the system package installation commands. Useful for debugging, and it is TRUE by default in a CI environment.

About other OSes

Windows:

While the system requirements database has some information about system dependencies on Windows, pak does not use this information and it does not try to install system software on Windows. CRAN, PPM and Bioconductor have Windows binary packages available for the majority of R packages they serve, and these packages practically always link to system libraries statically, so they don't need any external software.

If you wish to compile Windows packages from source, then you need to install the appropriate version of Rtools, and possibly extra packages using the pacman tool of Rtools4x.

Rtools42 and newer Rtools versions bundle lots of libraries, so most likely no extra pacman packages are needed. Rtools40 has a leaner default installation, and you'll probably need to

install packages manually: https://github.com/r-windows/docs/blob/master/rtools40.md#readme

We are planning on adding better Windows system software support to pak in the future.

macOS:

pak does not currently have system requirement information for macOS. macOS is similar to Windows, in that most repositories will serve statically linked macOS binary packages that do not need system software.

If you do need to compile packages from source, then you possibly need to install some system libraries, either via Homebrew, or by downloading CRAN's static library builds from https://mac.r-project.org/bin/

We are planning on adding better macOS system software support to pak in the future.

`system_r_platform` *R platforms*

Description

R platforms

Usage

```
system_r_platform()

system_r_platform_data()
```

Details

`system_r_platform()` detects the platform of the current R version. `system_r_platform_data()` is similar, but returns the raw data instead of a character scalar.

By default pak works with source packages and binary packages for the current platform. You can change this, by providing different platform names in the `pkg.platforms` option or the `PKG_PLATFORMS` environment variable.

This option may contain the following platform names:

- `"source"` for source packages,
- `"macos"` for macOS binaries that are appropriate for the R versions pak is working with. Packages for incompatible CPU architectures are dropped (defaulting to the CPU of the current macOS machine and x86_64 on non-macOS systems). The macOS Darwin version is selected based on the CRAN macOS binaries. E.g. on R 3.5.0 macOS binaries are built for macOS El Capitan.
- `"windows"` for Windows binaries for the default CRAN architecture. This is currently Windows Vista for all supported R versions, but it might change in the future. The actual binary packages in the repository might support both 32 bit and 64 builds, or only one of them. In practice 32-bit only packages are very rare. CRAN builds before and including R 4.1 have both architectures, from R 4.2 they are 64 bit only. `"windows"` is an alias to `i386+x86_64-w64-mingw32` currently.

- A platform string like `R.version$platform`, but on Linux the name and version of the distribution are also included. Examples:
  - `x86_64-apple-darwin17.0`: macOS High Sierra.
  - `aarch64-apple-darwin20`: macOS Big Sur on arm64.
  - `x86_64-w64-mingw32`: 64 bit Windows.
  - `i386-w64-mingw32`: 32 bit Windows.
  - `i386+x86_64-w64-mingw32`: 64 bit + 32 bit Windows.
  - `i386-pc-solaris2.10`: 32 bit Solaris. (Some broken 64 Solaris builds might have the same platform string, unfortunately.)
  - `x86_64-pc-linux-gnu-debian-10`: Debian Linux 10 on x86_64.
  - `x86_64-pc-linux-musl-alpine-3.14.1`: Alpine Linux.
  - `x86_64-pc-linux-gnu-unknown`: Unknown Linux Distribution on x86_64.
  - `s390x-ibm-linux-gnu-ubuntu-20.04`: Ubuntu Linux 20.04 on S390x.
  - `amd64-portbld-freebsd12.1`: FreeBSD 12.1 on x86_64.

Value

`system_r_platform()` returns a character scalar.

`system_r_platform_data()` returns a data frame with character scalar columns:

- `cpu`,
- `vendor`,
- `os`,
- `distribution` (only on Linux),
- `release` (only on Linux),
- `platform`: the concatenation of the other columns, separated by a dash.

See Also

These function call `pkgcache::current_r_platform()` and `pkgcache::current_r_platform_data()`.

Examples

```
system_r_platform()
system_r_platform_data()
```

`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