Package 'rix'

January 21, 2025

Title Reproducible Data Science Environments with 'Nix'

Version 0.14.4

Description Simplifies the creation of reproducible data science environments
using the 'Nix' package manager, as described in
Dolstra (2006) <ISBN 90-393-4130-3>. The included `rix()`
function generates a complete description of the environment as a
`default.nix` file, which can then be built using 'Nix'. This results in
project specific software environments with pinned versions of R, packages,
linked system dependencies, and other tools. Additional helpers make it easy
to run R code in 'Nix' software environments for testing and production.

License GPL (>= 3)

URL <https://docs.ropensci.org/rix/>

BugReports <https://github.com/ropensci/rix/issues>

Depends R (>= 2.10)

Imports codetools, curl, jsonlite, sys, utils

Suggests knitr, rmarkdown, testthat

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Bruno Rodrigues [aut, cre] (<<https://orcid.org/0000-0002-3211-3689>>),
Philipp Baumann [aut] (<<https://orcid.org/0000-0002-3194-8975>>),
David Watkins [rev] (David reviewed the package (v. 0.9.1) for
rOpenSci, see
<<https://github.com/ropensci/software-review/issues/625>>),
Jacob Wujiciak-Jens [rev] (<<https://orcid.org/0000-0002-7281-3989>>,
Jacob reviewed the package (v. 0.9.1) for rOpenSci, see
<<https://github.com/ropensci/software-review/issues/625>>),
Richard J. Acton [ctb] (<<https://orcid.org/0000-0002-2574-9611>>),
Jordi Rosell [ctb] (<<https://orcid.org/0000-0002-4349-1458>>),

Elio Campitelli [ctb] (<<https://orcid.org/0000-0002-7742-9230>>),
László Kupcsik [ctb] (<<https://orcid.org/0000-0003-3535-5496>>),
Michael Heming [ctb] (<<https://orcid.org/0000-0002-9568-2790>>)

Maintainer Bruno Rodrigues <bruno@brodrigues.co>

Contents

---

available_dates	*List available dates.*

Description

List available dates.

Usage

```
available_dates()
```

Value

A character vector containing the available dates

Examples

```
available_dates()
```

---

available_df *Return data frame with R, Bioc versions and supported platforms*

Description

Return data frame with R, Bioc versions and supported platforms

Usage

```
available_df()
```

Value

A data frame

Examples

```
available_dates()
```

---

available_r *List available R versions from the rstats-on-nix fork of Nixpkgs*

Description

List available R versions from the rstats-on-nix fork of Nixpkgs

Usage

```
available_r()
```

Value

A character vector containing the available R versions.

Examples

```
available_r()
```

---

ga_cachix	*ga_cachix Build an environment on GitHub Actions and cache it on*
	Cachix

Description

ga_cachix Build an environment on GitHub Actions and cache it on Cachix

Usage

```
ga_cachix(cache_name, path_default)
```

Arguments

cache_name	String, name of your cache.
path_default	String, relative path (from the root directory of your project) to the `default.nix` to build.

Details

This function puts a `.yaml` file inside the `.github/workflows/` folders on the root of your project. This workflow file will use the projects `default.nix` file to generate the development environment on GitHub Actions and will then cache the created binaries in Cachix. Create a free account on Cachix to use this action. Refer to `vignette("z-binary_cache")` for detailed instructions. Make sure to give read and write permissions to the GitHub Actions bot.

Value

Nothing, copies file to a directory.

Examples

```
## Not run:
ga_cachix("my-cachix", path_default = "default.nix")

## End(Not run)
```

---

nix_build	*Invoke shell command* nix-build *from an R session*

Description

Invoke shell command `nix-build` from an R session

Usage

```
nix_build(
  project_path = getwd(),
  message_type = c("simple", "quiet", "verbose")
)
```

Arguments

project_path	Path to the folder where the default.nix file resides.
message_type	Character vector with messaging type, Either "simple" (default), "quiet" for no messaging, or "verbose".

Details

The nix-build command line interface has more arguments. We will probably not support all of them in this R wrapper, but currently we have support for the following nix-build flags:

- --max-jobs: Maximum number of build jobs done in parallel by Nix. According to the official docs of Nix, it defaults to 1, which is one core. This option can be useful for shared memory multiprocessing or systems with high I/O latency. To set --max-jobs used, you can declare with options(rix.nix_build_max_jobs = <integer>). Once you call nix_build() the flag will be propagated to the call of nix-build.

Value

integer of the process ID (PID) of nix-build shell command launched, if nix_build() call is assigned to an R object. Otherwise, it will be returned invisibly.

Examples

```
## Not run:
nix_build()

## End(Not run)
```

---

read_renv_lock	*read_renv_lock*

Description

Reads renv.lock if it exists and can be parsed as json.

Usage

```
read_renv_lock(renv_lock_path = "renv.lock")
```

Arguments

renv_lock_path location of the renv.lock file, defaults to "renv.lock"

Value

the result of reading renv.lock with jsonlite::read_json

---

renv2nix   *renv2nix*

Description

renv2nix

Usage

```
renv2nix(
  renv_lock_path = "renv.lock",
  project_path,
  return_rix_call = FALSE,
  method = c("fast", "accurate"),
  override_r_ver = NULL,
  ...
)
```

Arguments

renv_lock_path Character, path of the renv.lock file, defaults to "renv.lock"

project_path   Character, where to write default.nix, for example "/home/path/to/project". The file will thus be written to the file "/home/path/to/project/default.nix". If the folder does not exist, it will be created.

return_rix_call

Logical, return the generated rix function call instead of evaluating it this is for debugging purposes, defaults to FALSE

method   Character, the method of generating a nix environment from an renv.lock file. "fast" is an inexact conversion which simply extracts the R version and a list of all the packages in an renv.lock file and adds them to the r_pkgs argument of rix(). This will use a snapshot of nixpkgs that should contain package versions that are not too different from the ones defined in the renv.lock file. For packages installed from GitHub or similar, an attempt is made to handle them and pass them to the git_pkgs argument of rix(). Currently defaults to "fast", "accurate" is not yet implemented.

override_r_ver Character, defaults to NULL, override the R version defined in the renv.lock file with another version. This is especially useful if the renv.lock file lists a version of R not (yet) available through Nix, or if the R version included in the renv.lock is too old compared to the package versions.

...   Arguments passed on to [rix](rix)

system_pkgs Vector of characters. List further software you wish to install that are not R packages such as command line applications for example. You can look for available software on the NixOS website https://search.nixos.org/packages?channel=unstable&from=0&size=50&sort=relevance&type=packages&query= # nolint

local_r_pkgs List. A list of local packages to install. These packages need to be in the `.tar.gz` or `.zip` formats and must be in the same folder as the generated "default.nix" file.

tex_pkgs Vector of characters. A set of TeX packages to install. Use this if you need to compile `.tex` documents, or build PDF documents using Quarto. If you don't know which package to add, start by adding "amsmath". See the Vignette "Authoring LaTeX documents" for more details.

ide Character, defaults to "other". If you wish to use RStudio to work interactively use "rstudio" or "rserver" for the server version. Use "code" for Visual Studio Code. You can also use "radian", an interactive REPL. For other editors, use "other". This has been tested with RStudio, VS Code and Emacs. If other editors don't work, please open an issue.

overwrite Logical, defaults to FALSE. If TRUE, overwrite the `default.nix` file in the specified path.

print Logical, defaults to FALSE. If TRUE, print `default.nix` to console.

message_type Character. Message type, defaults to "simple", which gives minimal but sufficient feedback. Other values are currently "quiet, which generates the files without message, and "verbose", displays all the messages.

shell_hook Character of length 1, defaults to NULL. Commands added to the `shellHook` variable are executed when the Nix shell starts. So by default, using `nix-shell default.nix` will start a specific program, possibly with flags (separated by space), and/or do shell actions. You can for example use `shell_hook = R`, if you want to directly enter the declared Nix R session when dropping into the Nix shell.

Details

In order for this function to work properly, we recommend not running it inside the same folder as an existing {renv} project. Instead, run it from a new, empty directory which path you pass to `project_path`, and use `renv_lock_path` to point to the `renv.lock` file in the original {renv} folder. We recommend that you start from an empty folder to hold your new Nix project, and copy the `renv.lock` file only (not any of the other files and folders generated by {renv}) and then call `renv2nix()` there. For more details, see `vignette("f-renv2nix")`.

Value

Nothing, this function is called for its side effects only, unless `return_rix_call = TRUE` in which case an unevaluated call to `rix()` is returned

Examples

```
## Not run:
```

```
# if the lock file is in another folder
renv2nix(
  renv_lock_path = "path/to/original/renv_project/renv.lock",
  project_path = "path/to/rix_project"
)
# you could also copy the renv.lock file in the folder of the Nix
# project (don't copy any other files generated by `{renv}`)
renv2nix(
  renv_lock_path = "path/to/rix_project/renv.lock",
  project_path = "path/to/rix_project"
)

## End(Not run)
```

---

renv_lock_r_ver   *renv_lock_r_ver*

Description

renv_lock_r_ver

Usage

```
renv_lock_r_ver(renv_lock, override_r_ver = NULL)
```

Arguments

renv_lock   renv.lock file from which to get the R version

override_r_ver   Character, override the R version defined in the renv.lock file with another
                   version. This is especially useful if the renv.lock file lists a version of R not
                   (yet) available through Nix.

Value

a length 1 character vector with the version of R recorded in renv.lock

Examples

```
## Not run:
rix(r_ver = renv_lock_r_ver())

## End(Not run)
```

---

renv_remote_pkgs   *renv_remote_pkgs*

Description

Construct a list to be passed the git_pkgs argument of rix The list returned contains the information necessary to have nix attempt to build the packages from their external repositories.

Usage

```
renv_remote_pkgs(renv_lock_remote_pkgs, host = NULL)
```

Arguments

renv_lock_remote_pkgs

the list of package information from an renv.lock file.

host   the host of remote package, defaults to NULL meaning the RemoteHost of the renv entry will be used. currently supported hosts: 'api.github.com' 'gitlab.com' see remotes for more.

Value

a list of lists with three elements named: "package_name", "repo_url", "commit"

Examples

```
## Not run:
renv_remote_pkgs(read_renv_lock()$Packages)

## End(Not run)
```

---

rix   *Generate a Nix expression that builds a reproducible development environment*

Description

Generate a Nix expression that builds a reproducible development environment

Usage

```
rix(
  r_ver = NULL,
  date = NULL,
  r_pkgs = NULL,
  system_pkgs = NULL,
  git_pkgs = NULL,
  local_r_pkgs = NULL,
  tex_pkgs = NULL,
  ide = c("other", "code", "radian", "rstudio", "rserver"),
  project_path,
  overwrite = FALSE,
  print = FALSE,
  message_type = "simple",
  shell_hook = NULL
)
```

Arguments

r_ver	Character. The required R version, for example "4.0.0". You can check which R versions are available using available_r(), and for more details check available_df(). For reproducibility purposes, you can also provide a nixpkgs revision directly. For older versions of R, nix-build might fail with an error stating 'this derivation is not meant to be built'. In this case, simply drop into the shell with nix-shell instead of building it first. It is also possible to provide either "bleeding-edge" or "frozen-edge" if you need an environment with bleeding edge packages. Read more in the "Details" section below.
date	Character. Instead of providing r_ver, it is also possible to provide a date. This will build an environment containing R and R packages (and other dependencies) as of that date. You can check which dates are available with available_dates(). For more details about versions check available_df().
r_pkgs	Vector of characters. List the required R packages for your analysis here.
system_pkgs	Vector of characters. List further software you wish to install that are not R packages such as command line applications for example. You can look for available software on the NixOS website https://search.nixos.org/packages?channel=unstable&from=0&size=50&sort=relevance&type=packages&query= # nolint
git_pkgs	List. A list of packages to install from Git. See details for more information.
local_r_pkgs	List. A list of local packages to install. These packages need to be in the .tar.gz or .zip formats and must be in the same folder as the generated "default.nix" file.
tex_pkgs	Vector of characters. A set of TeX packages to install. Use this if you need to compile .tex documents, or build PDF documents using Quarto. If you don't know which package to add, start by adding "amsmath". See the Vignette "Authoring LaTeX documents" for more details.

ide	Character, defaults to "other". If you wish to use RStudio to work interactively use "rstudio" or "rserver" for the server version. Use "code" for Visual Studio Code. You can also use "radian", an interactive REPL. For other editors, use "other". This has been tested with RStudio, VS Code and Emacs. If other editors don't work, please open an issue.
project_path	Character, where to write default.nix, for example "/home/path/to/project". The file will thus be written to the file "/home/path/to/project/default.nix". If the folder does not exist, it will be created.
overwrite	Logical, defaults to FALSE. If TRUE, overwrite the default.nix file in the specified path.
print	Logical, defaults to FALSE. If TRUE, print default.nix to console.
message_type	Character. Message type, defaults to "simple", which gives minimal but sufficient feedback. Other values are currently "quiet, which generates the files without message, and "verbose", displays all the messages.
shell_hook	Character of length 1, defaults to NULL. Commands added to the shellHook variable are executed when the Nix shell starts. So by default, using nix-shell default.nix will start a specific program, possibly with flags (separated by space), and/or do shell actions. You can for example use shell_hook = R, if you want to directly enter the declared Nix R session when dropping into the Nix shell.

Details

This function will write a default.nix and an .Rprofile in the chosen path. Using the Nix package manager, it is then possible to build a reproducible development environment using the nix-build command in the path. This environment will contain the chosen version of R and packages, and will not interfere with any other installed version (via Nix or not) on your machine. Every dependency, including both R package dependencies but also system dependencies like compilers will get installed as well in that environment.

It is possible to use environments built with Nix interactively, either from the terminal, or using an interface such as RStudio. If you want to use RStudio, set the ide argument to "rstudio". Please be aware that RStudio is not available for macOS through Nix. As such, you may want to use another editor on macOS. To use Visual Studio Code (or Codium), set the ide argument to "code", which will add the {languageserver} R package to the list of R packages to be installed by Nix in that environment. You can use the version of Visual Studio Code or Codium you already use, or also install it using Nix (by adding "vscode" or "vscodium" to the list of system_pkgs). For non-interactive use, or to use the environment from the command line, or from another editor (such as Emacs or Vim), set the ide argument to "other". We recommend reading the vignette("e-interactive-use") for more details.

Packages to install from GitHub or Gitlab must be provided in a list of 3 elements: "package_name", "repo_url" and "commit". To install several packages, provide a list of lists of these 3 elements, one per package to install. It is also possible to install old versions of packages by specifying a version. For example, to install the latest version of {AER} but an old version of {ggplot2}, you could write: r_pkgs = c("AER", "ggplot2@2.2.1"). Note however that doing this could result in dependency hell, because an older version of a package might need older versions of its dependencies, but other packages might need more recent versions of the same dependencies. If instead you want to use an environment as it would have looked at the time of {ggplot2}'s version 2.2.1 release, then use the Nix revision closest to that date, by setting r_ver = "3.1.0", which was the version of R

current at the time. This ensures that Nix builds a completely coherent environment. For security purposes, users that wish to install packages from GitHub/Gitlab or from the CRAN archives must provide a security hash for each package. {rix} automatically precomputes this hash for the source directory of R packages from GitHub/Gitlab or from the CRAN archives, to make sure the expected trusted sources that match the precomputed hashes in the default.nix are downloaded. If Nix is available, then the hash will be computed on the user's machine, however, if Nix is not available, then the hash gets computed on a server that we set up for this purposes. This server then returns the security hash as well as the dependencies of the packages. It is possible to control this behaviour using options(rix.sri_hash=x), where x is one of "check_nix" (the default), "locally" (use the local Nix installation) or "api_server" (use the remote server to compute and return the hash).

Note that installing packages from Git or old versions using the "@" notation or local packages, does not leverage Nix's capabilities for dependency solving. As such, you might have trouble installing these packages. If that is the case, open an issue on {rix}'s GitHub repository.

By default, the Nix shell will be configured with "en_US.UTF-8" for the relevant locale variables (LANG, LC_ALL, LC_TIME, LC_MONETARY, LC_PAPER, LC_MEASUREMENT). This is done to ensure locale reproducibility by default in Nix environments created with rix(). If there are good reasons to not stick to the default, you can set your preferred locale variables via options(rix.nix_locale_variables = list(LANG = "c and the aforementioned locale variable names.

It is possible to use "bleeding-edge" or "frozen-edge" as the value for the r_ver argument. This will create an environment with the very latest R packages. "bleeding-edge" means that every time you will build the environment, the packages will get updated. This is especially useful for environments that need to be constantly updated, for example when developing a package. In contrast, "frozen-edge" will create an environment that will remain stable at build time. So if you create a default.nix file using "bleeding-edge", each time you build it using nix-build that environment will be up-to-date. With "frozen-edge" that environment will be up-to-date on the date that the default.nix will be generated, and then each subsequent call to nix-build will result in the same environment. "bioc-devel" is the same as "bleeding-edge", but also adds the development version of Bioconductor. "r-devel" is the same as bleeding edge, but with the R development version instead of the latest stable version and "r-devel-bioc-devel" is the same as "r-devel" but with Bioconductor on the development version. We highly recommend you read the vignette titled "z - Advanced topic: Understanding the rPackages set release cycle and using bleeding edge packages".

Value

Nothing, this function only has the side-effect of writing two files: default.nix and .Rprofile in the working directory. default.nix contains a Nix expression to build a reproducible environment using the Nix package manager, and .Rprofile ensures that a running R session from a Nix environment cannot access local libraries, nor install packages using install.packages() (nor remove nor update them).

Examples

```
## Not run:
# Build an environment with the latest version of R available from Nixpkgs
# and the dplyr and ggplot2 packages
rix(
  r_ver = "latest-upstream",
```

```
    r_pkgs = c("dplyr", "ggplot2"),
    system_pkgs = NULL,
    git_pkgs = NULL,
    local_r_pkgs = NULL,
    ide = "code",
    project_path = path_default_nix,
    overwrite = TRUE,
    print = TRUE,
    message_type = "simple",
    shell_hook = NULL
  )

  ## End(Not run)
```

| rix_init | *Initiate and maintain an isolated, project-specific, and runtime-pure R setup via Nix.* |

Description

Creates an isolated project folder for a Nix-R configuration. `rix::rix_init()` also adds, appends, or updates with or without backup a custom `.Rprofile` file with code that initializes a startup R environment without system's user libraries within a Nix software environment. Instead, it restricts search paths to load R packages exclusively from the Nix store. Additionally, it makes Nix utilities like `nix-shell` available to run system commands from the system's RStudio R session, for both Linux and macOS.

Usage

```
rix_init(
  project_path,
  rprofile_action = c("create_missing", "create_backup", "overwrite", "append"),
  message_type = c("simple", "quiet", "verbose")
)
```

Arguments

project_path   Character with the folder path to the isolated nix-R project. If the folder does not exist yet, it will be created.

rprofile_action

Character. Action to take with `.Rprofile` file destined for `project_path` folder. Possible values include `"create_missing"`, which only writes `.Rprofile` if it does not yet exist (otherwise does nothing) - this is the action set when using `rix()` - ; `"create_backup"`, which copies the existing `.Rprofile` to a new backup file, generating names with POSIXct-derived strings that include the time zone information. A new `.Rprofile` file will be written with default code from `rix::rix_init()`; `"overwrite"` overwrites the `.Rprofile` file if it does exist; `"append"` appends the existing file with code that is tailored to an isolated Nix-R project setup.

message_type   Character. Message type, defaults to ″simple″, which gives minimal but suf-
                ficient feedback. Other values are currently ″quiet, which writes .Rprofile
                without message, and ″verbose″, which displays the mechanisms implemented
                to achieve fully controlled R project environments in Nix.

Details

Enhancement of computational reproducibility for Nix-R environments:

The primary goal of rix::rix_init() is to enhance the computational reproducibility of Nix-R
environments during runtime. Concretely, if you already have a system or user library of R packages
(if you have R installed through the usual means for your operating system), using rix::rix_init()
will prevent Nix-R environments to load packages from the user library which would cause issues.
Notably, no restart is required as environmental variables are set in the current session, in addition
to writing an .Rprofile file. This is particularly useful to make [with_nix()](#) evaluate custom R
functions from any "Nix-to-Nix" or "System-to-Nix" R setups. It introduces two side-effects that
take effect both in a current or later R session setup:

1. **Adjusting** R_LIBS_USER **path:** By default, the first path of R_LIBS_USER points to the user li-
   brary outside the Nix store (see also [base::.libPaths()](#)). This creates friction and potential
   impurity as R packages from the system's R user library are loaded. While this feature can be
   useful for interactively testing an R package in a Nix environment before adding it to a .nix
   configuration, it can have undesired effects if not managed carefully. A major drawback is
   that all R packages in the R_LIBS_USER location need to be cleaned to avoid loading packages
   outside the Nix configuration. Issues, especially on macOS, may arise due to segmentation
   faults or incompatible linked system libraries. These problems can also occur if one of the
   (reverse) dependencies of an R package is loaded along the process.

2. **Make Nix commands available when running system commands from RStudio:** In a
   host RStudio session not launched via Nix (nix-shell), the environmental variables from
   ~/.zshrc or ~/.bashrc may not be inherited. Consequently, Nix command line interfaces
   like nix-shell might not be found. The .Rprofile code written by rix::rix_init() en-
   sures that Nix command line programs are accessible by adding the path of the "bin" directory
   of the default Nix profile, ″/nix/var/nix/profiles/default/bin″, to the PATH variable in
   an RStudio R session.

These side effects are particularly recommended when working in flexible R environments, espe-
cially for users who want to maintain both the system's native R setup and utilize Nix expressions
for reproducible development environments. This init configuration is considered pivotal to en-
hance the adoption of Nix in the R community, particularly until RStudio in Nixpkgs is packaged
for macOS. We recommend calling rix::rix_init() prior to comparing R code ran between two
software environments with rix::with_nix().

rix::rix_init() is called automatically by rix::rix() when generating a default.nix file,
and when called by rix::rix() will only add the .Rprofile if none exists. In case you have a
custom .Rprofile that you wish to keep using, but also want to benefit from what rix_init()
offers, manually call it and set the rprofile_action to ″append″.

Value

Nothing, this function only has the side-effect of writing a file called ".Rprofile" to the specified
path.

See Also

with_nix()

Examples

```
## Not run:
# create an isolated, runtime-pure R setup via Nix
project_path <- "./sub_shell"
if (!dir.exists(project_path)) dir.create(project_path)
rix_init(
  project_path = project_path,
  rprofile_action = "create_missing",
  message_type = c("simple")
)

## End(Not run)
```

---

setup_cachix   *setup_cachix Setup up the rstats-on-nix binary repository*

Description

setup_cachix Setup up the rstats-on-nix binary repository

Usage

```
setup_cachix(nix_conf_path = "~/.config/nix")
```

Arguments

nix_conf_path   Character, path to folder containing 'nix.conf' file. Defaults to "~/.config/nix".

Details

This function edits `~/.config/nix/nix.conf` to add the `rstats-on-nix` public cache as a substituter. The `rstats-on-nix` public cache, hosted on Cachix, contains many prebuild binaries of R and R packages for x86_64 Linux and macOS (Intel architectures for packages released before 2021 and Apple Silicon from 2021 onwards). This function automatically performs a backup of `~/.config/nix/nix.conf`, or creates one if there is no `nix.conf` file. If you installed Nix by following the instructions provided in the "Getting started" vignette and also followed the instructions to install Cachix and configure the cache, you don't need to run this. This function is useful in mainly two cases: if you somehow mess up `~/.config/nix/nix.conf` and need to generate a new one from scratch, or if you're using Nix inside Docker, write a `RUN Rscript -e 'rix::setup_cachix()'` statement to configure the cache there. Because Docker runs using `root` by default no need to install the `cachix` client to configure the cache, running `setup_cachix()` is enough. See the 'z - Advanced topic: Using Nix inside Docker' vignette for more details.

Value

Nothing; changes a file in the user's home directory.

Examples

```
## Not run:
setup_cachix()

## End(Not run)
```

---

tar_nix_ga   *tar_nix_ga Run a {targets} pipeline on GitHub Actions.*

Description

tar_nix_ga Run a {targets} pipeline on GitHub Actions.

Usage

```
tar_nix_ga()
```

Details

This function puts a `.yaml` file inside the `.github/workflows/` folders on the root of your project. This workflow file will use the projects `default.nix` file to generate the development environment on GitHub Actions and will then run the projects {targets} pipeline. Make sure to give read and write permissions to the GitHub Actions bot.

Value

Nothing, copies file to a directory.

Examples

```
## Not run:
tar_nix_ga()

## End(Not run)
```

---

with_nix   *Evaluate function in R or shell command via* nix-shell *environment*

Description

This function needs an installation of Nix. with_nix() has two effects to run code in isolated and reproducible environments.

1. Evaluate a function in R or a shell command via the nix-shell environment (Nix expression for custom software libraries; involving pinned versions of R and R packages via Nixpkgs)

2. If no error, return the result object of expr in with_nix() into the current R session.

Usage

```
with_nix(
  expr,
  program = c("R", "shell"),
  project_path = ".",
  message_type = c("simple", "quiet", "verbose")
)
```

Arguments

expr   Single R function or call, or character vector of length one with shell command and possibly options (flags) of the command to be invoked. For program = R, you can both use a named or an anonymous function. The function provided in expr should not evaluate when you pass arguments, hence you need to wrap your function call like function() your_fun(arg_a = "a", arg_b = "b"), to avoid evaluation and make sure expr is a function (see details and examples).

program   String stating where to evaluate the expression. Either "R", the default, or "shell". where = "R" will evaluate the expression via RScript and where = "shell" will run the system command in nix-shell.

project_path   Path to the folder where the default.nix file resides. The default is ".", which is the working directory in the current R session. This approach also useful when you have different subfolders with separate software environments defined in different default.nix files.

message_type   String how detailed output is. Currently, there is either "simple" (default), "quiet or "verbose", which shows the script that runs via nix-shell.

Details

with_nix() gives you the power of evaluating a main function expr and its function call stack that are defined in the current R session in an encapsulated nix-R session defined by Nix expression (default.nix), which is located in at a distinct project path (project_path).

with_nix() is very convenient because it gives direct code feedback in read-eval-print-loop style, which gives a direct interface to the very reproducible infrastructure-as-code approach offered by

Nix and Nixpkgs. You don't need extra efforts such as setting up DevOps tooling like Docker and domain specific tools like {renv} to control complex software environments in R and any other language. It is for example useful for the following purposes.

1. test compatibility of custom R code and software/package dependencies in development and production environments

2. directly stream outputs (returned objects), messages and errors from any command line tool offered in Nixpkgs into an R session.

3. Test if evolving R packages change their behavior for given unchanged R code, and whether they give identical results or not.

`with_nix()` can evaluate both R code from a nix-R session within another nix-R session, and also from a host R session (i.e., on macOS or Linux) within a specific nix-R session. This feature is useful for testing the reproducibility and compatibility of given code across different software environments. If testing of different sets of environments is necessary, you can easily do so by providing Nix expressions in custom `.nix` or `default.nix` files in different subfolders of the project.

`rix_init()` is run automatically to generate a custom `.Rprofile` file for the subshell in `project_dir`. The defaults in that file ensure that only R packages from the Nix store, that are defined in the sub-shell `.nix` file are loaded and system's libraries are excluded.

To do its job, `with_nix()` heavily relies on patterns that manipulate language expressions (aka computing on the language) offered in base R as well as the {codetools} package by Luke Tierney.

Some of the key steps that are done behind the scene:

1. recursively find, classify, and export global objects (globals) in the call stack of `expr` as well as propagate R package environments found.

2. Serialize (save to disk) and deserialize (read from disk) dependent data structures as `.Rds` with necessary function arguments provided, any relevant globals in the call stack, packages, and `expr` outputs returned in a temporary directory.

3. Use pure `nix-shell` environments to execute a R code script reconstructed catching expressions with quoting; it is launched by commands like this via {sys} by Jeroen Ooms: `nix-shell --pure --run "Rscript --vanilla"`.

Value

- if `program = "R"`, R object returned by function given in `expr` when evaluated via the R environment in `nix-shell` defined by Nix expression.

- if `program = "shell"`, list with the following elements:
    - `status`: exit code
    - `stdout`: character vector with standard output
    - `stderr`: character vector with standard error of `expr` command sent to a command line interface provided by a Nix package.

Examples

```
## Not run:
# create an isolated, runtime-pure R setup via Nix
project_path <- "./sub_shell"
```

```
rix_init(
  project_path = project_path,
  rprofile_action = "create_missing"
)
# generate nix environment in `default.nix`
rix(
  r_ver = "4.2.0",
  project_path = project_path
)
# evaluate function in Nix-R environment via `nix-shell` and `Rscript`,
# stream messages, and bring output back to current R session
out <- with_nix(
  expr = function(mtcars) nrow(mtcars),
  program = "R", project_path = project_path,
  message_type = "simple"
)


# There no limit in the complexity of function call stacks that `with_nix()`
# can possibly handle; however, `expr` should not evaluate and
# needs to be a function for `program = "R"`. If you want to pass the
# a function with arguments, you can do like this
get_sample <- function(seed, n) {
  set.seed(seed)
  out <- sample(seq(1, 10), n)
  return(out)
}

out <- with_nix(
  expr = function() get_sample(seed = 1234, n = 5),
  program = "R",
  project_path = ".",
  message_type = "simple"
)

## You can also attach packages with `library()` calls in the current R
## session, which will be exported to the nix-R session.
## Other option: running system commands through `nix-shell` environment.

## End(Not run)
```

Index