Package: tic (via r-universe)

July 17, 2024

Type Package

Title Tasks Integrating Continuously: CI-Agnostic Workflow Definitions

Version 0.14.0

Description Provides a way to describe common build and deployment workflows for R-based projects: packages, websites (e.g. blogdown, pkgdown), or data processing (e.g. research compendia). The recipe is described independent of the continuous integration tool used for processing the workflow (e.g. 'GitHub Actions' or 'Circle CI'). This package has been peer-reviewed by rOpenSci (v0.3.0.9004).

License GPL (>= 2)

URL https://github.com/ropensci/tic

BugReports https://github.com/ropensci/tic/issues

Depends R (>= 3.2.0)

Imports cli (>= 3.2.0), crayon, git2r, lifecycle, magrittr, memoise, methods, pak, R6, remotes, rlang (>= 1.0.0), usethis, withr

Suggests base64enc, blogdown, bookdown, callr, circle, covr, desc, devtools, drat, fansi, gh (>= 1.1.0), knitr, openssl, pkgdepends, pkgdown, purrr, rcmdcheck, rmarkdown, rprojroot, sodium, stats, stringr, testthat (>= 2.1.0), utils

VignetteBuilder knitr

RdMacros lifecycle

ByteCompile No

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

SystemRequirements sodium harfbuzz fribidi libgit2

Collate 'base64.R' 'ci.R' 'circleci.R' 'droneci.R' 'dsl-storage.R' 'dsl.R' 'gh-actions.R' 'git2r.R' 'helpers_github.R' 'install.R' 'local.R' 'macro.R' 'macro-package-checks.R' 'macro-pkgdown.R' 'macro-blogdown.R' 'macro-bookdown.R' 'macro-drat.R' 'macro-readme-rmd.R' 'mock.R' 'print.R' 'repo.R' 'run.R' 'stage.R' 'steps-base.R' 'steps-blogdown.R' 'steps-bookdown.R' 'steps-code.R' 'steps-drat.R' 'steps-git.R' 'steps-install.R' 'steps-rcmdcheck.R' 'steps-pkgdown.R' 'steps-session-info.R' 'steps-ssh.R' 'steps-write-text-file.R' 'tic-package.R' 'update-yaml-helpers.R' 'update-yaml.R' 'use-badge.R' 'use-yaml.R' 'use_tic.R' 'utils.R'

Repository https://ropensci.r-universe.dev

RemoteUrl https://github.com/ropensci/tic

RemoteRef main

RemoteSha 380b98b52a1d3c42c177d28f0827d148971d33e9

Contents

`tic-package` *tic: Tasks Integrating Continuously: CI-Agnostic Workflow Definitions*

Description

Provides a way to describe common build and deployment workflows for R-based projects: packages, websites (e.g. blogdown, pkgdown), or data processing (e.g. research compendia). The recipe is described independent of the continuous integration tool used for processing the workflow (e.g. 'GitHub Actions' or 'Circle CI'). This package has been peer-reviewed by rOpenSci (v0.3.0.9004).

Details

The `use_tic()` function prepares a code repository for use with this package. See DSL for an overview of **tic**'s domain-specific language for defining stages and steps, `step_hello_world()` and the links therein for available steps, and macro for an overview over the available macros that bundle several steps.

Author(s)

Maintainer: Patrick Schratz <`patrick.schratz@gmail.com`> (ORCID)

Authors:

- Kirill Müller (ORCID)
- Mika Braginsky <`mika.br@gmail.com`>

- Karthik Ram <karthik.ram@gmail.com>
- Jeroen Ooms <jeroen.ooms@stat.ucla.edu>

Other contributors:

- Max Held (Max reviewed the package for ropensci, see <https://github.com/ropensci/software-review/issues/305>) [reviewer]
- Anna Krystalli (Anna reviewed the package for ropensci, see <https://github.com/ropensci/software-review/issues/305>) [reviewer]
- Laura DeCicco (Laura reviewed the package for ropensci, see <https://github.com/ropensci/software-review/issues/305>) [reviewer]
- rOpenSci [funder]

See Also

Useful links:

- https://github.com/ropensci/tic
- Report bugs at https://github.com/ropensci/tic/issues

---

base64serialize   *Helpers for converting R objects to strings and back*

Description

`base64serialize()` converts an R object into a string suitable for storing in an environment variable. Use this function for encoding entire R objects (such as OAuth tokens).

`base64unserialize()` is the inverse operation to `base64serialize()`. Use this function in your `tic.R` to access the R object previously encoded by `base64serialize()`.

Usage

```
base64serialize(x, compression = "gzip")

base64unserialize(x, compression = "gzip")
```

Arguments

`x`	Object to serialize or deserialize
`compression`	Passed on as `type` argument to `memCompress()` or `memDecompress()`.

Examples

```
serial <- base64serialize(1:10)
base64unserialize(serial)
```

`ci` *The current CI environment*

Description

Functions that return environment settings that describe the CI environment. The value is retrieved only once and then cached.

`ci_get_branch()`: Returns the current branch. Returns nothing if operating on a tag.

`ci_is_tag()`: Returns the current tag name. Returns nothing if a branch is selected.

`ci_get_slug()`: Returns the repo slug in the format `user/repo` or `org/repo`

`ci_get_build_number()`: Returns the CI build number.

`ci_get_build_url()`: Returns the URL of the current build.

`ci_get_commit()`: Returns the SHA1 of the current commit.

`ci_get_env()`: Return an environment or configuration variable.

`ci_is_env()`: Checks if an environment or configuration variable is set to a particular value.

`ci_has_env()`: Checks if an environment or configuration variable is set to any value.

`ci_can_push()`: Checks if push deployment is possible. Always true for local environments, CI environments require an environment variable (by default `TIC_DEPLOY_KEY`).

`ci_is_interactive()`: Returns whether the current build is run interactively or not. Global setup operations shouldn't be run on interactive CIs.

`ci_cat_with_color()`: Colored output targeted to the CI log. The code argument can be an unevaluated call to a crayon function, the style will be applied even if it normally wouldn't be.

`ci_on_circle()`: Are we running on Circle CI?

`ci_on_ghactions()`: Are we running on GitHub Actions?

`ci()`: Return the current CI environment

Usage

```
ci_get_branch()

ci_is_tag()

ci_get_slug()

ci_get_build_number()

ci_get_build_url()

ci_get_commit()

ci_get_env(env)
```

```
ci_is_env(env, value)

ci_has_env(env)

ci_can_push(private_key_name = "TIC_DEPLOY_KEY")

ci_is_interactive()

ci_cat_with_color(code)

ci_on_circle()

ci_on_ghactions()

ci()
```

Arguments

`env`	Name of the environment variable to check.
`value`	Value for the environment variable to compare against.
`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.
`code`	Code that should be colored.

`Deprecated` *Deprecated functions*

Description

`add_package_checks()` has been replaced by `do_package_checks()`.

Usage

```
add_package_checks(
  ...,
  warnings_are_errors = NULL,
  notes_are_errors = NULL,
  args = c("--no-manual", "--as-cran"),
  build_args = "--force",
  error_on = "warning",
  repos = repo_default(),
  timeout = Inf
)
```

Arguments

`...`	Ignored, used to enforce naming of arguments.
`warnings_are_errors`, `notes_are_errors`	`[flag]` Deprecated, use `error_on`.
`args`	`[character]` Passed to `rcmdcheck::rcmdcheck()`. Default for local runs: `c("--no-manual", "--as-cran")`. Default for Windows: `c("--no-manual", "--as-cran", "--no-vignettes", "--no-build-vignettes", "--no-multiarch")`. On GitHub Actions option "–no-manual" is always used (appended to custom user input) because LaTeX is not available and installation is time consuming and error prone.
`build_args`	`[character]` Passed to `rcmdcheck::rcmdcheck()`. Default for local runs: `"--force"`. Default for Windows: `c("--no-build-vignettes", "--force")`.
`error_on`	`[character]` Whether to throw an error on R CMD check failures. Note that the check is always completed (unless a timeout happens), and the error is only thrown after completion. If "never", then no errors are thrown. If "error", then only ERROR failures generate errors. If "warning", then WARNING failures generate errors as well. If "note", then any check failure generated an error.
`repos`	`[character]` Passed to `rcmdcheck::rcmdcheck()`, default: `repo_default()`.
`timeout`	`[numeric]` Passed to `rcmdcheck::rcmdcheck()`, default: `Inf`.

`do_blogdown` *Build a blogdown site*

Description

`do_blogdown()` adds default steps related to package checks to the `"install"`, `"before_deploy"`, `"script"` and `"deploy"` stages.

1. `step_install_deps()` in the `"install"` stage
2. `blogdown::install_hugo()` in the `"install"` stage to install the latest version of HUGO.
3. `step_session_info()` in the `"install"` stage.
4. `step_setup_ssh()` in the `"before_deploy"` to setup the upcoming deployment (if `deploy` is set),

5. step_setup_push_deploy() in the "before_deploy" stage (if deploy is set),
6. step_build_blogdown() in the "deploy" stage, forwarding all ... arguments.
7. step_do_push_deploy() in the "deploy" stage.

By default, the public/ directory is deployed to the gh-pages branch, keeping the history. If the output directory of your blog/theme is not "public" you need to change the "path" argument.

Usage

```
do_blogdown(
  ...,
  deploy = NULL,
  orphan = FALSE,
  checkout = TRUE,
  path = "public",
  branch = "gh-pages",
  remote_url = NULL,
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE,
  private_key_name = "TIC_DEPLOY_KEY",
  cname = NULL
)
```

Arguments

...	Passed on to step_build_blogdown()
deploy	[flag] If TRUE, deployment setup is performed before building the blogdown site, and the site is deployed after building it. Set to FALSE to skip deployment. By default (if deploy is NULL), deployment happens if the following conditions are met: 1. The repo can be pushed to (see ci_can_push()). 2. The branch argument is NULL (i.e., if the deployment happens to the active branch), or the current branch is the default repo branch (see ci_get_branch()).
orphan	[flag] Create and force-push an orphan branch consisting of only one commit? This can be useful e.g. for path = "docs", branch = "gh-pages", but cannot be applied for pushing to the current branch.
checkout	[flag] Check out the current contents of the repository? Defaults to TRUE, set to FALSE if the build process relies on existing contents or if you deploy to a different branch.
path	[string] Path to the repository, default "." which means setting up the current repository.
branch	[string] Target branch, default: current branch.

`remote_url`	`[string]` The URL of the remote Git repository to push to, defaults to the current GitHub repository.
`commit_message`	`[string]` Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
`commit_paths`	`[character]` Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.
`force`	`[logical]` Add `--force` flag to git commands?
`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.
`cname`	`(character(1)` An optional URL for redirecting the created website A `CNAME` file containing the given URL will be added to the root of the directory specified in argument `path`.

See Also

Other macros: `do_bookdown()`, `do_drat()`, `do_package_checks()`, `do_pkgdown()`, `do_readme_rmd()`, `list_macros()`

Examples

```
## Not run:
dsl_init()

do_blogdown()

dsl_get()

## End(Not run)
```

do_bookdown *Build a bookdown book*

Description

`do_bookdown()` adds default steps related to package checks to the `"install"`, `"before_deploy"`, `"script"` and `"deploy"` stages.

1. `step_install_deps()` in the `"install"` stage
2. `step_session_info()` in the `"install"` stage.

3. `step_setup_ssh()` in the `"before_deploy"` to setup the upcoming deployment (if `deploy` is set),
4. `step_setup_push_deploy()` in the `"before_deploy"` stage (if `deploy` is set),
5. `step_build_bookdown()` in the `"deploy"` stage, forwarding all `...` arguments.
6. `step_do_push_deploy()` in the `"deploy"` stage.

By default, the `_book/` directory is deployed to the `gh-pages` branch, keeping the history.

Usage

```
do_bookdown(
  ...,
  deploy = NULL,
  orphan = FALSE,
  checkout = TRUE,
  path = "_book",
  branch = "gh-pages",
  remote_url = NULL,
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE,
  private_key_name = "TIC_DEPLOY_KEY",
  cname = NULL
)
```

Arguments

`...`	Passed on to `step_build_bookdown()`
`deploy`	`[flag]` If `TRUE`, deployment setup is performed before building the bookdown site, and the site is deployed after building it. Set to `FALSE` to skip deployment. By default (if `deploy` is `NULL`), deployment happens if the following conditions are met: 1. The repo can be pushed to (see `ci_can_push()`). 2. The `branch` argument is `NULL` (i.e., if the deployment happens to the active branch), or the current branch is the default repo branch (usually "master") (see `ci_get_branch()`).
`orphan`	`[flag]` Create and force-push an orphan branch consisting of only one commit? This can be useful e.g. for `path = "docs", branch = "gh-pages"`, but cannot be applied for pushing to the current branch.
`checkout`	`[flag]` Check out the current contents of the repository? Defaults to `TRUE`, set to `FALSE` if the build process relies on existing contents or if you deploy to a different branch.
`path`	`[string]` Path to the repository, default `"."` which means setting up the current repository.

branch	[string] Target branch, default: current branch.
remote_url	[string] The URL of the remote Git repository to push to, defaults to the current GitHub repository.
commit_message	[string] Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
commit_paths	[character] Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.
force	[logical] Add --force flag to git commands?
private_key_name	string Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or use_tic(), pass this name here.
cname	(character(1) An optional URL for redirecting the created website A CNAME file containing the given URL will be added to the root of the directory specified in argument path.

See Also

Other macros: do_blogdown(), do_drat(), do_package_checks(), do_pkgdown(), do_readme_rmd(), list_macros()

Examples

```
## Not run:
dsl_init()

do_bookdown()

dsl_get()

## End(Not run)
```

---

do_drat *Build and deploy drat repository*

Description

do_drat() builds and deploys R packages to a drat repository and adds default steps to the "install", "before_deploy" and "deploy" stages:

1. step_setup_ssh() in the "before_deploy" to setup the upcoming deployment

2. step_setup_push_deploy() in the "before_deploy" stage (if deploy is set),
3. step_add_to_drat() in the "deploy"
4. step_do_push_deploy() in the "deploy" stage.

Usage

```
do_drat(
  repo_slug = NULL,
  orphan = FALSE,
  checkout = TRUE,
  path = "~/git/drat",
  branch = NULL,
  remote_url = NULL,
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE,
  private_key_name = "TIC_DEPLOY_KEY",
  deploy_dev = FALSE
)
```

Arguments

repo_slug	[string] The name of the drat repository to deploy to in the form :owner/:repo.
orphan	[flag] Create and force-push an orphan branch consisting of only one commit? This can be useful e.g. for path = "docs", branch = "gh-pages", but cannot be applied for pushing to the current branch.
checkout	[flag] Check out the current contents of the repository? Defaults to TRUE, set to FALSE if the build process relies on existing contents or if you deploy to a different branch.
path, branch	By default, this macro deploys the default repo branch (usually "master") of the drat repository. An alternative option is "gh-pages".
remote_url	[string] The URL of the remote Git repository to push to, defaults to the current GitHub repository.
commit_message	[string] Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
commit_paths	[character] Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.
force	[logical] Add --force flag to git commands?
private_key_name	string Only needed when deploying from builds on GitHub Actions. If you have set

	a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.
`deploy_dev`	`[logical]` Should development versions of packages also be deployed to the drat repo? By default only "major", "minor" and "patch" releases are build and deployed.

Deployment

Deployment can only happen to the default repo branch (usually "master") or `gh-pages` branch because the GitHub Pages functionality from GitHub is used to access the drat repository later on. You need to enable this functionality when creating the drat repository on GitHub via `Settings -> GitHub pages` and set it to the chosen setting here.

To build and deploy Windows and macOS binaries, builds with deployment permissions need to be triggered. Have a look at `https://docs.ropensci.org/tic/articles/deployment.html` for more information and instructions.

See Also

Other macros: `do_blogdown()`, `do_bookdown()`, `do_package_checks()`, `do_pkgdown()`, `do_readme_rmd()`, `list_macros()`

Examples

```
## Not run:
dsl_init()

do_drat()

dsl_get()

## End(Not run)
```

`do_package_checks` *Add default checks for packages*

Description

`do_package_checks()` adds default steps related to package checks to the `"before_install"`, `"install"`, `"script"` and `"after_success"` stages:

This macro is only available for R packages.

1. `step_install_deps()` in the `"install"` stage, using the `repos` argument.
2. `step_session_info()` in the `"install"` stage.
3. `step_rcmdcheck()` in the `"script"` stage, using the `warnings_are_errors`, `notes_are_errors`, `args`, and `build_args` arguments.
4. A call to `covr::codecov()` in the `"after_success"` stage (only if the `codecov` flag is set)

Usage

```
do_package_checks(
  ...,
  codecov = !ci_is_interactive(),
  warnings_are_errors = NULL,
  notes_are_errors = NULL,
  args = NULL,
  build_args = NULL,
  error_on = "warning",
  repos = repo_default(),
  dependencies = TRUE,
  timeout = Inf,
  check_dir = "check"
)
```

Arguments

`...` Ignored, used to enforce naming of arguments.

`codecov` `[flag]`
Whether to include a step running `covr::codecov(quiet = FALSE)` (default: only for non-interactive CI, see `ci_is_interactive()`).

`warnings_are_errors, notes_are_errors`
`[flag]`
Deprecated, use `error_on`.

`args` `[character]`
Passed to `rcmdcheck::rcmdcheck()`.

Default for local runs: `c("--no-manual", "--as-cran")`.

Default for Windows: `c("--no-manual", "--as-cran", "--no-vignettes", "--no-build-vignettes", "--no-multiarch")`.

On GitHub Actions option "–no-manual" is always used (appended to custom user input) because LaTeX is not available and installation is time consuming and error prone.

`build_args` `[character]`
Passed to `rcmdcheck::rcmdcheck()`.
Default for local runs: `"--force"`.
Default for Windows: `c("--no-build-vignettes", "--force")`.

`error_on` `[character]`
Whether to throw an error on R CMD check failures. Note that the check is always completed (unless a timeout happens), and the error is only thrown after completion. If "never", then no errors are thrown. If "error", then only ERROR failures generate errors. If "warning", then WARNING failures generate errors as well. If "note", then any check failure generated an error.

`repos` `[character]`
Passed to `rcmdcheck::rcmdcheck()`, default: `repo_default()`.

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.
timeout	[numeric] Passed to rcmdcheck::rcmdcheck(), default: Inf.
check_dir	[character] Path specifying the directory for R CMD check. Defaults to "check" for easy upload of artifacts.

See Also

Other macros: do_blogdown(), do_bookdown(), do_drat(), do_pkgdown(), do_readme_rmd(), list_macros()

Examples

```
dsl_init()

do_package_checks()

dsl_get()
```

---

do_pkgdown *Build pkgdown documentation*

Description

do_pkgdown() builds and optionally deploys a pkgdown site and adds default steps to the "install", "before_deploy" and "deploy" stages:

1. step_install_deps() in the "install" stage
2. step_session_info() in the "install" stage.
3. step_setup_ssh() in the "before_deploy" to setup the upcoming deployment (if deploy is set and only on GitHub Actions),
4. step_setup_push_deploy() in the "before_deploy" stage (if deploy is set),
5. step_build_pkgdown() in the "deploy" stage, forwarding all ... arguments.
6. step_do_push_deploy() in the "deploy" stage.

By default, the docs/ directory is deployed to the gh-pages branch, keeping the history.

Usage

```
do_pkgdown(
  ...,
  deploy = NULL,
  orphan = FALSE,
  checkout = TRUE,
  path = "docs",
  branch = "gh-pages",
  remote_url = NULL,
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE,
  private_key_name = "TIC_DEPLOY_KEY"
)
```

Arguments

`...`	Passed on to `step_build_pkgdown()`
`deploy`	`[flag]` If `TRUE`, deployment setup is performed before building the pkgdown site, and the site is deployed after building it. Set to `FALSE` to skip deployment. By default (if `deploy` is `NULL`), deployment happens if the following conditions are met: 1. The repo can be pushed to (see `ci_can_push()`). account for old default "id_rsa" 2. The `branch` argument is `NULL` (i.e., if the deployment happens to the active branch), or the current branch is the default branch, or contains "cran-" in its name (for compatibility with **fledge**) (see `ci_get_branch()`).
`orphan`	`[flag]` Create and force-push an orphan branch consisting of only one commit? This can be useful e.g. for `path = "docs", branch = "gh-pages"`, but cannot be applied for pushing to the current branch.
`checkout`	`[flag]` Check out the current contents of the repository? Defaults to `TRUE`, set to `FALSE` if the build process relies on existing contents or if you deploy to a different branch.
`path, branch`	By default, this macro deploys the `docs` directory to the `gh-pages` branch. This is different from `step_push_deploy()`.
`remote_url`	`[string]` The URL of the remote Git repository to push to, defaults to the current GitHub repository.
`commit_message`	`[string]` Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
`commit_paths`	`[character]` Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.

force	[logical] Add --force flag to git commands?
private_key_name	string Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or use_tic(), pass this name here.

See Also

Other macros: do_blogdown(), do_bookdown(), do_drat(), do_package_checks(), do_readme_rmd(), list_macros()

Examples

```
## Not run:
dsl_init()

do_pkgdown()

dsl_get()

## End(Not run)
```

---

do_readme_rmd *Render a R Markdown README and deploy to Github*

Description

[Experimental]

do_readme_rmd() renders an R Markdown README and deploys the rendered README.md file to Github. It adds default steps to the "before_deploy" and "deploy" stages:

1. step_setup_ssh() in the "before_deploy" to setup the upcoming deployment
2. step_setup_push_deploy() in the "before_deploy" stage
3. rmarkdown::render() in the "deploy" stage
4. step_do_push_deploy() in the "deploy" stage.

Usage

```
do_readme_rmd(
  checkout = TRUE,
  remote_url = NULL,
  commit_message = NULL,
  force = FALSE,
  private_key_name = "TIC_DEPLOY_KEY"
)
```

Arguments

`checkout`	`[flag]` Check out the current contents of the repository? Defaults to `TRUE`, set to `FALSE` if the build process relies on existing contents or if you deploy to a different branch.
`remote_url`	`[string]` The URL of the remote Git repository to push to, defaults to the current GitHub repository.
`commit_message`	`[string]` Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
`force`	`[logical]` Add `--force` flag to git commands?
`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.

See Also

Other macros: `do_blogdown()`, `do_bookdown()`, `do_drat()`, `do_package_checks()`, `do_pkgdown()`, `list_macros()`

Examples

```
## Not run:
dsl_init()

do_readme_rmd()

dsl_get()

## End(Not run)
```

---

dsl *tic's domain-specific language*

Description

Functions to define stages and their constituent steps. The macros combine several steps and assign them to relevant stages. See `dsl_get()` for functions to access the storage for the stages and their steps.

`get_stage()` returns a `TicStage` object for a stage given by name. This function can be called directly in the `tic.R` configuration file, which is processed by `dsl_load()`.

`add_step()` adds a step to a stage, see `step_hello_world()` and the links therein for available steps.

`add_code_step()` is a shortcut for `add_step(step_run_code(...))`.

Usage

```
get_stage(name)

add_step(stage, step)

add_code_step(stage, call = NULL, prepare_call = NULL)
```

Arguments

`name`	`[string]` The name for the stage.
`stage`	`[TicStage]` A `TicStage` object as returned by `get_stage()`.
`step`	`[function]` An object of class TicStep, usually created by functions with the `step_` prefix like `step_hello_world()`.
`call`	`[call]` An arbitrary R expression executed during the stage to which this step is added. The default is useful if you only pass `prepare_call`.
`prepare_call`	`[call]` An optional arbitrary R expression executed during preparation.

Examples

```
dsl_init()

get_stage("script")

get_stage("script") %>%
  add_step(step_hello_world())

get_stage("script")

get_stage("script") %>%
  add_code_step(print("Hi!"))

get_stage("script")
```

`dsl_get` *Stages and steps*

Description

tic works in a declarative way, centered around the `tic.R` file created by `use_tic()`. This file contains the *definition* of the steps to be run in each stage: calls to `get_stage()` and `add_step()`, or macros like `do_package_checks()`.

Normally, this file is never executed directly. Running these functions in an interactive session will **not** carry out the respective actions. Instead, a description of the code that would have been run is printed to the console. Edit `tic.R` to configure your CI builds. See `vignette("build-lifecycle", package = "tic")` for more details.

Usage

```
dsl_get()

dsl_load(path = "tic.R", force = FALSE, quiet = FALSE)

dsl_init(quiet = FALSE)
```

Arguments

`path`	`[string]` Path to the stage definition file, default: `"tic.R"`.
`force`	`[flag]` Set to `TRUE` to force loading from file even if a configuration exists. By default an existing configuration is not overwritten by `dsl_load()`.
`quiet`	`[flag]` Set to `TRUE` to turn off verbose output.

Details

Stages and steps defined using tic's DSL are stored in an internal object in the package. The stages are accessible through `dsl_get()`. When running the stages, by default a configuration defined in the `tic.R` file is loaded with `dsl_load()`. See `use_tic()` for setting up a `tic.R` file.

For interactive tests, an empty storage can be initialized with `dsl_init()`. This happens automatically the first time `dsl_get()` is called (directly or indirectly).

Value

A named list of opaque stage objects with a `"class"` attribute and a corresponding `print()` method for pretty output. Use the high-level `get_stage()` and `add_step()` functions to configure, and the stages functions to run.

Examples

```
## Not run:
dsl_init()
dsl_get()

dsl_load(system.file("templates/package/tic.R", package = "tic"))
dsl_load(system.file("templates/package/tic.R", package = "tic"),
  force =
    TRUE
)
dsl_get()
```

```
## End(Not run)
```

---

gha_add_secret *Add a GitHub Actions secret to a repository*

Description

Encrypts the supplied value using `libsodium` and adds it as a secret to the given GitHub repository. Secrets can be be used in GitHub Action runs as environment variables. A common use case is to encrypt Personal Access Tokens (PAT) or API keys.

This is the same as adding a secret manually in GitHub via `"Settings"` -> `"Secrets"` -> `"New repository secret"`

Usage

```
gha_add_secret(
  secret,
  name,
  repo_slug = NULL,
  remote = "origin",
  visibility = "all",
  selected_repositories = NULL
)
```

Arguments

`secret`	`[character]` The value which should be encrypted (e.g. a Personal Access Token).
`name`	`[character]` The name of the secret as which it will appear in the "Secrets" overview of the repository.
`repo_slug`	`[character]` Repository slug of the repository to which the secret should be added. Must follow the form `owner/repo`.
`remote`	`[character]` If `repo_slug` = `NULL`, the `repo_slug` is determined by the respective git remote.
`visibility`	`[character]` The level of visibility for the secret. One of `"all"`, `"private"`, or `"selected"`. See https://developer.github.com/v3/actions/secrets/#create-or-update-an-organization-secret for more information.
`selected_repositories`	`[character]` Vector of repository ids for which the secret is accessible. Only applies if `visibility` = `"selected"` was set.

Examples

```
## Not run:
gha_add_secret("supersecret", name = "MY_SECRET", repo = "ropensci/tic")

## End(Not run)
```

`github_helpers` *Github API helpers*

Description

- `auth_github()`: Creates a `GITHUB_TOKEN` and asks to store it in your `.Renviron` file.
- `get_owner()`: Returns the owner of a Github repo.
- `get_repo()`: Returns the repo name of a Github repo for a given remote.
- `get_repo_slug()`: Returns the repo slug of a Github repo (`<owner>/<repo>`).

Usage

```
auth_github()

get_owner(remote = "origin")

get_user()

get_repo(remote = "origin")

get_repo_slug(remote = "origin")
```

Arguments

`remote`	`[string]` The Github remote which should be used. Defaults to "origin".

`github_repo` *Github information*

Description

`github_repo()` returns the true repository name as string.

Retrieves metadata about a Git repository from GitHub.

`github_info()` returns a list as obtained from the GET "/repos/:repo" API.

Usage

```
github_repo(
  path = usethis::proj_get(),
  info = github_info(path, remote = remote),
  remote = "origin"
)

github_info(path = usethis::proj_get(), remote = "origin")

uses_github(path = usethis::proj_get())
```

Arguments

`path`	`[string]` The path to a GitHub-enabled Git repository (or a subdirectory thereof).
`info`	`[list]` GitHub information for the repository, by default obtained through `github_info()`.
`remote`	`[string]` The Github remote which should be used. Defaults to "origin".

`list_macros` *List available macros*

Description

Lists available macro functions of the `tic` package.

Usage

```
list_macros()
```

Value

character

See Also

Other macros: `do_blogdown()`, `do_bookdown()`, `do_drat()`, `do_package_checks()`, `do_pkgdown()`, `do_readme_rmd()`

`macro` *Macros*

Description

The DSL offers a fine-grained interface to the individual stages of a CI run. Macros are tic's way of adding several related steps to the relevant stages. All macros use the `do_` prefix.

The `do_package_checks()` macro adds default checks for R packages, including installation of dependencies and running a test coverage analysis.

The `do_pkgdown()` macro adds the necessary steps for building and deploying **pkgdown** documentation for a package.

The `do_blogdown()` macro adds the necessary steps for building and deploying a **blogdown** blog.

The `do_bookdown()` macro adds the necessary steps for building and deploying a **bookdown** book.

The `do_drat()` macro adds the necessary steps for building and deploying a drat repository to host R package sources.

The `do_readme_rmd()` macro renders an R Markdown README and deploys the rendered README.md file to Github.

`prepare_all_stages` *Prepare all stages*

Description

Run the `prepare()` method for all defined steps for which the `check()` method returns `TRUE`.

Usage

```
prepare_all_stages(stages = dsl_load())
```

Arguments

`stages`	`[named list]` A named list of `TicStage` objects as returned by `dsl_load()`, by default loaded from `tic.R`.

See Also

TicStep

Other runners: `run_all_stages()`, `run_stage()`

`repo` *Shortcuts for accessing CRAN-like repositories*

Description

These functions can be used as convenient shortcuts for the `repos` argument to e.g. `do_package_checks()` and `step_install_deps()`.

`repo_default()` returns the value of the `"repos"` option, or `repo_cloud()` if the option is not set.

`repo_cloud()` returns RStudio's CRAN mirror.

`repo_cran()` returns the master CRAN repo.

`repo_bioc()` returns Bioconductor repos from `remotes::bioc_install_repos()`, in addition to the default repo.

Usage

```
repo_default()

repo_cloud()

repo_cran()

repo_bioc(base = repo_default())
```

Arguments

`base`	The base repo to use, defaults to `repo_default()`. Pass `NULL` to install only from Bioconductor repos.

`run_all_stages` *Emulate a CI run locally*

Description

Runs predefined stages similarly to the chosen CI provider. The run aborts on error, the `after_failure` stage is never run.

Usage

```
run_all_stages(stages = dsl_load())
```

Arguments

`stages`	`[named list]` A named list of `TicStage` objects as returned by `dsl_load()`, by default loaded from `tic.R`.

Details

The stages are run in the following order:

1. `before_install()`
2. `install()`
3. `after_install()`
4. `before_script()`
5. `script()`
6. `after_success()`
7. `before_deploy()`
8. `deploy()`
9. `after_deploy()`
10. `after_script()`

See Also

Other runners: `prepare_all_stages()`, `run_stage()`

`run_stage` *Run a stage*

Description

Run the `run_all()` method for all defined steps of a stage for which the `check()` method returns `TRUE`.

Usage

```
run_stage(name, stages = dsl_load())
```

Arguments

`name`	`[string]` The name of the stage to run.
`stages`	`[named list]` A named list of `TicStage` objects as returned by `dsl_load()`, by default loaded from `tic.R`.

See Also

TicStep

Other runners: `prepare_all_stages()`, `run_all_stages()`

ssh_key_helpers *SSH key helpers*

Description

SSH key helpers

Usage

```
github_add_key(
  pubkey,
  repo = get_repo(remote),
  user = get_user(),
  title = "ghactions",
  remote = "origin",
  check_role = TRUE
)

check_admin_repo(owner, user, repo)

get_role_in_repo(owner, user, repo)

get_public_key(key)

encode_private_key(key)

check_private_key_name(string)
```

Arguments

`pubkey`	The public key of the SSH key pair
`repo`	`[string]` The repository slug to use. Must follow the "`user/repo`" structure.
`user`	The name of the user account
`title`	The title of the key to add
`remote`	`[string]` The Github remote which should be used. Defaults to "origin".
`check_role`	Whether to check if the current user has the permissions to add a key to the repo. Setting this to `FALSE` makes it possible to add keys to other repos than just the one from which the function is called.
`owner`	The owner of the repository
`key`	The SSH key pair object
`string`	String to check

`stages` *Predefined stages*

Description

Stages available in the CI provider, for which shortcuts have been defined. All these functions call `run_stage()` with the corresponding stage name.

Usage

```
before_install(stages = dsl_load())

install(stages = dsl_load())

after_install(stages = dsl_load())

before_script(stages = dsl_load())

script(stages = dsl_load())

after_success(stages = dsl_load())

after_failure(stages = dsl_load())

before_deploy(stages = dsl_load())

deploy(stages = dsl_load())

after_deploy(stages = dsl_load())

after_script(stages = dsl_load())
```

Arguments

`stages`	`[named list]` A named list of `TicStage` objects as returned by `dsl_load()`, by default loaded from `tic.R`.

`step_add_to_drat` *Step: Add built package to a drat*

Description

Builds a package (binary on OS X or Windows) and inserts it into an existing **drat** repository via `drat::insertPackage()`.

Usage

```
step_add_to_drat(repo_slug = NULL, deploy_dev = FALSE)
```

Arguments

`repo_slug`	`[string]` The name of the drat repository to deploy to in the form `:owner/:repo`.
`deploy_dev`	`[logical]` Should development versions of packages also be deployed to the drat repo? By default only "major", "minor" and "patch" releases are build and deployed.

See Also

Other steps: `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_add_to_drat())

dsl_get()
```

step_add_to_known_hosts

Step: Add to known hosts

Description

Adds a host name to the `~/.ssh/known_hosts` file to allow subsequent SSH access. Requires `ssh-keyscan` on the system `PATH`.

Usage

```
step_add_to_known_hosts(host = "github.com")
```

Arguments

`host`	`[string]` The host name to add to the `known_hosts` file, default: `github.com`.

See Also

Other steps: step_add_to_drat(), step_build_pkgdown(), step_do_push_deploy(), step_hello_world(), step_install_pkg, step_install_ssh_keys(), step_push_deploy(), step_run_code(), step_session_info(), step_setup_push_deploy(), step_setup_ssh(), step_test_ssh(), step_write_text_file()

Examples

```
dsl_init()

get_stage("before_deploy") %>%
  add_step(step_add_to_known_hosts("gitlab.com"))

dsl_get()
```

---

step_build_blogdown *Step: Build a Blogdown Site*

Description

Build a Blogdown site using blogdown::build_site().

Usage

```
step_build_blogdown(...)
```

Arguments

...	Arguments passed on to blogdown::build_site `local` Whether to build the website locally. This argument is passed to hugo_build(), and `local = TRUE` is mainly for serving the site locally via serve_site(). `run_hugo` Whether to run `hugo_build()` after R Markdown files are compiled. `build_rmd` Whether to (re)build R Markdown files. By default, they are not built. See 'Details' for how `build_rmd = TRUE` works. Alternatively, it can take a vector of file paths, which means these files are to be (re)built. Or you can provide a function that takes a vector of paths of all R Markdown files under the '`content/`' directory, and returns a vector of paths of files to be built, e.g., `build_rmd = blogdown::filter_timestamp`. A few aliases are currently provided for such functions: `build_rmd = 'newfile'` is equivalent to `build_rmd = blogdown::filter_newfile`, `build_rmd = 'timestamp'` is equivalent to `build_rmd = blogdown::filter_timestamp`, and `build_rmd = 'md5sum'` is equivalent to `build_rmd = blogdown::filter_md5sum`.

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_build_blogdown("."))

dsl_get()
```

step_build_bookdown *Step: Build a bookdown book*

Description

Build a bookdown book using `bookdown::render_book()`.

Usage

```
step_build_bookdown(...)
```

Arguments

`...`	See bookdown::render_book.

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_build_bookdown("."))

dsl_get()
```

step_build_pkgdown *Step: Build pkgdown documentation*

Description

Builds package documentation with the **pkgdown** package. Calls `pkgdown::clean_site()` and then `pkgdown::build_site(...)`.

Usage

```
step_build_pkgdown(...)
```

Arguments

`...`	Arguments passed on to `pkgdown::build_site`

`pkg` Path to package.

`examples` Run examples?

`run_dont_run` Run examples that are surrounded in \dontrun?

`seed` Seed used to initialize so that random examples are reproducible.

`lazy` If `TRUE`, will only rebuild articles and reference pages if the source is newer than the destination.

`override` An optional named list used to temporarily override values in `_pkgdown.yml`

`preview` If `TRUE`, or `is.na(preview) && interactive()`, will preview freshly generated section in browser.

`devel` Use development or deployment process?
If `TRUE`, uses lighter-weight process suitable for rapid iteration; it will run examples and vignettes in the current process, and will load code with `pkgload::load_all()`.
If `FALSE`, will first install the package to a temporary library, and will run all examples and vignettes in a new process.
`build_site()` defaults to `devel = FALSE` so that you get high fidelity outputs when you building the complete site; `build_reference()`, `build_home()` and friends default to `devel = TRUE` so that you can rapidly iterate during development.

`new_process` If `TRUE`, will run `build_site()` in a separate process. This enhances reproducibility by ensuring nothing that you have loaded in the current process affects the build process.

`install` If `TRUE`, will install the package in a temporary library so it is available for vignettes.

`document` **Deprecated** Use `devel` instead.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_build_pkgdown())

dsl_get()
```

`step_do_push_deploy` *Step: Perform push deploy*

Description

Commits and pushes to a repo prepared by `step_setup_push_deploy()`.

Deployment usually requires setting up SSH keys with `use_tic()`.

Usage

```
step_do_push_deploy(
  path = ".",
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE
)
```

Arguments

`path`	`[string]` Path to the repository, default `"."` which means setting up the current repository.
`commit_message`	`[string]` Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.
`commit_paths`	`[character]` Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.
`force`	`[logical]` Add `--force` flag to git commands?

Details

It is highly recommended to restrict the set of files touched by the deployment with the `commit_paths` argument: this step assumes that it can freely overwrite all changes to all files below `commit_paths`, and will not warn in case of conflicts.

To mitigate conflicts race conditions to the greatest extent possible, the following strategy is used:

- The changes are committed to the branch
- Before pushing, new commits are fetched, and the changes are cherry-picked on top of the new commits

If no new commits were pushed after the CI run has started, this strategy is equivalent to committing and pushing. In the opposite case, if the remote repo has new commits, the deployment is safely applied to the current tip.

See Also

Other deploy steps: step_push_deploy(), step_setup_push_deploy()

Other steps: step_add_to_drat(), step_add_to_known_hosts(), step_build_pkgdown(), step_hello_world(), step_install_pkg, step_install_ssh_keys(), step_push_deploy(), step_run_code(), step_session_info(), step_setup_push_deploy(), step_setup_ssh(), step_test_ssh(), step_write_text_file()

Examples

```
## Not run:
dsl_init()

# Deployment only works if a companion step_setup_push_deploy() is added
get_stage("deploy") %>%
  add_step(step_setup_push_deploy(path = "docs", branch = "gh-pages")) %>%
  add_step(step_build_pkgdown())

if (rlang::is_installed("git2r") && git2r::in_repository()) {
  get_stage("deploy") %>%
    add_step(step_do_push_deploy(path = "docs"))
}

dsl_get()

## End(Not run)
```

---

step_hello_world *Step: Hello, world!*

Description

The simplest step possible: prints "Hello, world!" to the console when run, does not require any preparation. This step may be useful to test a **tic** setup or as a starting point when implementing a custom step.

Usage

```
step_hello_world()
```

See Also

Other steps: step_add_to_drat(), step_add_to_known_hosts(), step_build_pkgdown(), step_do_push_deploy(), step_install_pkg, step_install_ssh_keys(), step_push_deploy(), step_run_code(), step_session_info(), step_setup_push_deploy(), step_setup_ssh(), step_test_ssh(), step_write_text_file()

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_hello_world())

dsl_get()
```

step_install_pkg *Step: Install packages*

Description

These steps are useful if your CI run needs additional packages. Usually they are declared as dependencies in your `DESCRIPTION`, but it is also possible to install dependencies manually. By default, binary versions of packages are installed if possible, even if the CRAN version is ahead.

A `step_install_deps()` step installs all package dependencies declared in `DESCRIPTION`, using `pak::local_install_dev_deps()`. This includes upgrading outdated packages.

This step can only be used if a DESCRIPTION file is present in the repository root.

A `step_install_cran()` step installs one package from CRAN via `install.packages()`, but only if it's not already installed.

A `step_install_github()` step installs one or more packages from GitHub via `pak::pkg_install()`, the packages are only installed if their GitHub version is different from the locally installed version.

Usage

```
step_install_deps(dependencies = TRUE)

step_install_cran(package = NULL, ...)

step_install_github(repo = NULL, ...)
```

Arguments

`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.
`package`	Package(s) to install
`...`	Passed on to `pak::pkg_install()`.
`repo`	Package to install in the "user/repo" format.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
dsl_init()

get_stage("install") %>%
  add_step(step_install_deps())

dsl_get()
dsl_init()

get_stage("install") %>%
  add_step(step_install_cran("magick"))

dsl_get()
dsl_init()

get_stage("install") %>%
  add_step(step_install_github("rstudio/gt"))

dsl_get()
```

`step_install_ssh_keys` *Step: Install an SSH key*

Description

Writes a private SSH key encoded in an environment variable to a file in `~/.ssh`. Only run in non-interactive settings and if the environment variable exists and is non-empty. `use_ghactions_deploy()` and `use_tic()` functions encode a private key as an environment variable for use with this function.

Usage

```
step_install_ssh_keys(private_key_name = "TIC_DEPLOY_KEY")
```

Arguments

`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.

See Also

use_tic(), use_ghactions_deploy()

Other steps: step_add_to_drat(), step_add_to_known_hosts(), step_build_pkgdown(), step_do_push_deploy(), step_hello_world(), step_install_pkg, step_push_deploy(), step_run_code(), step_session_info(), step_setup_push_deploy(), step_setup_ssh(), step_test_ssh(), step_write_text_file()

Examples

```
dsl_init()

get_stage("before_deploy") %>%
  add_step(step_install_ssh_keys())

dsl_get()
```

---

step_push_deploy — *Step: Setup and perform push deploy*

Description

Clones a repo, initializes author information, sets up remotes, commits, and pushes. Combines step_setup_push_deploy() with checkout = FALSE and a suitable orphan argument, and step_do_push_deploy().

Deployment usually requires setting up SSH keys with use_tic().

Usage

```
step_push_deploy(
  path = ".",
  branch = NULL,
  remote_url = NULL,
  commit_message = NULL,
  commit_paths = ".",
  force = FALSE
)
```

Arguments

path	[string] Path to the repository, default "." which means setting up the current repository.
branch	[string] Target branch, default: current branch.
remote_url	[string] The URL of the remote Git repository to push to, defaults to the current GitHub repository.
commit_message	[string] Commit message to use, defaults to a useful message linking to the CI build and avoiding recursive CI runs.

commit_paths	[character] Restrict the set of directories and/or files added to Git before deploying. Default: deploy all files.
force	[logical] Add --force flag to git commands?

Details

Setup and deployment are combined in one step, the files to be deployed must be prepared in a previous step. This poses some restrictions on how the repository can be initialized, in particular for a nonstandard path argument only orphan = TRUE can be supported (and will be used).

For more control, create two separate steps with step_setup_push_deploy() and step_do_push_deploy(), and create the files to be deployed in between these steps.

See Also

Other deploy steps: step_do_push_deploy(), step_setup_push_deploy()

Other steps: step_add_to_drat(), step_add_to_known_hosts(), step_build_pkgdown(), step_do_push_deploy(), step_hello_world(), step_install_pkg, step_install_ssh_keys(), step_run_code(), step_session_info(), step_setup_push_deploy(), step_setup_ssh(), step_test_ssh(), step_write_text_file()

Examples

```
## Not run:
dsl_init()

get_stage("script") %>%
  add_step(step_push_deploy(commit_paths = c("NAMESPACE", "man")))

dsl_get()

## End(Not run)
```

step_rcmdcheck   *Step: Check a package*

Description

Check a package using rcmdcheck::rcmdcheck(), which ultimately calls R CMD check.

Usage

```
step_rcmdcheck(
  ...,
  warnings_are_errors = NULL,
  notes_are_errors = NULL,
  args = NULL,
  build_args = NULL,
  error_on = "warning",
  repos = repo_default(),
  timeout = Inf,
  check_dir = "check"
)
```

Arguments

`...`	Ignored, used to enforce naming of arguments.
`warnings_are_errors`, `notes_are_errors`	`[flag]` Deprecated, use `error_on`.
`args`	`[character]` Passed to `rcmdcheck::rcmdcheck()`. Default for local runs: `c("--no-manual", "--as-cran")`. Default for Windows: `c("--no-manual", "--as-cran", "--no-vignettes", "--no-build-vignettes", "--no-multiarch")`. On GitHub Actions option "–no-manual" is always used (appended to custom user input) because LaTeX is not available and installation is time consuming and error prone.
`build_args`	`[character]` Passed to `rcmdcheck::rcmdcheck()`. Default for local runs: `"--force"`. Default for Windows: `c("--no-build-vignettes", "--force")`.
`error_on`	`[character]` Whether to throw an error on R CMD check failures. Note that the check is always completed (unless a timeout happens), and the error is only thrown after completion. If "never", then no errors are thrown. If "error", then only ERROR failures generate errors. If "warning", then WARNING failures generate errors as well. If "note", then any check failure generated an error.
`repos`	`[character]` Passed to `rcmdcheck::rcmdcheck()`, default: `repo_default()`.
`timeout`	`[numeric]` Passed to `rcmdcheck::rcmdcheck()`, default: `Inf`.
`check_dir`	`[character]` Path specifying the directory for R CMD check. Defaults to `"check"` for easy upload of artifacts.

Updating of (dependency) packages

Packages shipped with the R-installation will not be updated as they will be overwritten by the R-installer in each build. If you want these package to be updated, please add the following step to your workflow: `add_code_step(remotes::update_packages("<pkg>"))`.

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_rcmdcheck(error_on = "note", repos = repo_bioc()))

dsl_get()
```

step_run_code *Step: Run arbitrary R code*

Description

Captures the expression and executes it when running the step. An optional preparatory expression can be provided that is executed during preparation. If the top-level expression is a qualified function call (of the format `package::fun()`), the package is installed during preparation.

Usage

```
step_run_code(call = NULL, prepare_call = NULL)
```

Arguments

`call`	`[call]` An arbitrary R expression executed during the stage to which this step is added. The default is useful if you only pass `prepare_call`.
`prepare_call`	`[call]` An optional arbitrary R expression executed during preparation.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file(`

Examples

```
dsl_init()

get_stage("install") %>%
  add_step(step_run_code(update.packages(ask = FALSE)))

# Will install covr from CRAN during preparation:
get_stage("after_success") %>%
  add_code_step(covr::codecov())

dsl_get()
```

`step_session_info` *Step: Print the current Session Info*

Description

Prints out the package information of the current session via `sessioninfo::session_info()`.

Usage

```
step_session_info()
```

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
dsl_init()

get_stage("install") %>%
  add_step(step_session_info())

dsl_get()
```

`step_setup_push_deploy` *Step: Setup push deploy*

Description

Clones a repo, inits author information, and sets up remotes for a subsequent `step_do_push_deploy()`.

Usage

```
step_setup_push_deploy(
  path = ".",
  branch = NULL,
  orphan = FALSE,
  remote_url = NULL,
  checkout = TRUE
)
```

Arguments

`path`	`[string]` Path to the repository, default `"."` which means setting up the current repository.
`branch`	`[string]` Target branch, default: current branch.
`orphan`	`[flag]` Create and force-push an orphan branch consisting of only one commit? This can be useful e.g. for `path = "docs", branch = "gh-pages"`, but cannot be applied for pushing to the current branch.
`remote_url`	`[string]` The URL of the remote Git repository to push to, defaults to the current GitHub repository.
`checkout`	`[flag]` Check out the current contents of the repository? Defaults to `TRUE`, set to `FALSE` if the build process relies on existing contents or if you deploy to a different branch.

See Also

Other deploy steps: `step_do_push_deploy()`, `step_push_deploy()`

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_ssh()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
## Not run:
dsl_init()

get_stage("deploy") %>%
  add_step(step_setup_push_deploy(path = "docs", branch = "gh-pages")) %>%
  add_step(step_build_pkgdown())

# This example needs a Git repository
if (rlang::is_installed("git2r") && git2r::in_repository()) {
  # Deployment only works if a companion step_do_push_deploy() is added
  get_stage("deploy") %>%
    add_step(step_do_push_deploy(path = "docs"))
}

dsl_get()

## End(Not run)
```

step_setup_ssh *Step: Setup SSH*

Description

Adds to known hosts, installs private key, and tests the connection. Chaining `step_install_ssh_keys()`, `step_add_to_known_hosts()` and `step_test_ssh()`. `use_tic()` encodes a private key as an environment variable for use with this function.

Usage

```
step_setup_ssh(
  private_key_name = "TIC_DEPLOY_KEY",
  host = "github.com",
  url = paste0("git@", host),
  verbose = ""
)
```

Arguments

`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.
`host`	`[string]` The host name to add to the `known_hosts` file, default: `github.com`.
`url`	`[string]` URL to establish SSH connection with, by default `git@github.com`
`verbose`	`[string]` Verbosity, by default `""`. Use `-v` or `"-vvv"` for more verbosity.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_test_ssh()`, `step_write_text_file()`

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_setup_ssh(host = "gitlab.com"))

dsl_get()
```

step_test_ssh *Step: Test SSH connection*

Description

Establishes an SSH connection. This step doesn't fail if the connection cannot be established, but prints verbose output by default. It is useful for troubleshooting deployment problems.

Usage

```
step_test_ssh(
  url = "git@github.com",
  verbose = "",
  private_key_name = "TIC_DEPLOY_KEY"
)
```

Arguments

`url`	`[string]` URL to establish SSH connection with, by default `git@github.com`
`verbose`	`[string]` Verbosity, by default `""`. Use `-v` or `"-vvv"` for more verbosity.
`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_write_text_file(`

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_test_ssh(verbose = "-vvv"))

dsl_get()
```

step_write_text_file *Step: Write a text file*

Description

Creates a text file with arbitrary contents

Usage

```
step_write_text_file(..., path)
```

Arguments

`...`	`[character]` Contents of the text file.
`path`	`[string]` Path to the new text file.

See Also

Other steps: `step_add_to_drat()`, `step_add_to_known_hosts()`, `step_build_pkgdown()`, `step_do_push_deploy()`, `step_hello_world()`, `step_install_pkg`, `step_install_ssh_keys()`, `step_push_deploy()`, `step_run_code()`, `step_session_info()`, `step_setup_push_deploy()`, `step_setup_ssh()`, `step_test_ssh()`

Examples

```
dsl_init()

get_stage("script") %>%
  add_step(step_write_text_file("Hi!", path = "hello.txt"))

dsl_get()
```

TicStep *The base class for all steps*

Description

Override this class to create a new step.

Methods

Public methods:

- `TicStep$new()`
- `TicStep$run()`
- `TicStep$prepare()`
- `TicStep$check()`

Method `new()`: Create a `TicStep` object.

Usage:

```
TicStep$new()
```

Method `run()`: This method must be overridden, it is called when running the stage to which a step has been added.

Usage:

```
TicStep$run()
```

Method `prepare()`: This is just a placeholder. This method is called when preparing the stage to which a step has been added. It auto-install all packages which are needed for a certain step. For example, `step_build_pkgdown()` requires the *pkgdown* package.

For `add_code_step()`, it autodetects any package calls in the form of `pkg::fun` and tries to install these packages from CRAN. If a steps `prepare_call` is not empty, the `$prepare` method is skipped for this step. This can be useful if a package should be installed from non-standard repositories, e.g. from GitHub.

Usage:

```
TicStep$prepare()
```

Method `check()`: This method determines if a step is prepared and run. Return `FALSE` if conditions for running this step are not met.

Usage:

```
TicStep$check()
```

`update_yml` *Update tic YAML Templates*

Description

Updates YAML templates to their latest versions. Currently only GitHub Actions and Circle CI templates are supported.

Usage

```
update_yml(template_in = NULL, template_out = NULL)
```

Arguments

`template_in`	`[character]` Path to template which should be updated. By default all standard template paths of GitHub Actions or Circle CI will be searched and updated if they exist. Alternatively a full path to a single template can be passed.
`template_out`	`[character]` Where the updated template should be written to. This is mainly used for internal testing purposes and should not be set by the user.

Details

By default all workflow files starting with `tic` are matched. This means that you can have multiple YAML files with update support, e.g. `"tic.yml"` and `"tic-db.yml"`.

Formatting requirements of tic YAML templates

To ensure that updating of tic templates works, ensure the following points:

- Your template contains the type (e.g. linux-matrix-deploy) and the revision date in its first two lines.
- When inserting comments into custom code blocks, only one-line comments are allowed. Otherwise the update heuristic gets in trouble.

See Also

yaml_templates

Examples

```
## Not run:
# auto-search
update_yml()

update_yml("tic.yml")

# custom named templates
update_yml("custom-name.yml")

# full paths
update_yml("~/path/to/repo/.github/workflows/tic.yml")

## End(Not run)
```

`use_ghactions_deploy` *Setup deployment for GitHub Actions*

Description

[Experimental]

Creates a public-private key pair, adds the public key to the GitHub repository via `github_add_key()`, and stores the private key as a "secret" in the GitHub repo.

Usage

```
use_ghactions_deploy(
  path = usethis::proj_get(),
  repo = get_repo_slug(remote),
  key_name_private = "TIC_DEPLOY_KEY",
  key_name_public = "Deploy key for GitHub Actions",
  remote = "origin"
)
```

Arguments

`path`	`[string]` The path to the repository.
`repo`	`[string]` The repository slug to use. Must follow the "`user/repo`" structure.
`key_name_private`	`[string]` The name of the private key of the SSH key pair which will be created. If not supplied, `"TIC_DEPLOY_KEY"` will be used.
`key_name_public`	`[string]` The name of the private key of the SSH key pair which will be created. If not supplied, `"Deploy key for GitHub Actions"` will be used.
`remote`	`[string]` The GitHub remote which should be used. Defaults to "origin".

`use_tic` *Initialize CI testing using tic*

Description

Prepares a repo for building and deploying supported by **tic**.

Usage

```
use_tic(
  wizard = interactive(),
  linux = "ghactions",
  mac = "ghactions",
  windows = "ghactions",
  deploy = "ghactions",
  matrix = "none",
  private_key_name = "TIC_DEPLOY_KEY",
  quiet = FALSE
)
```

Arguments

`wizard`	`[flag]` Interactive operation? If `TRUE`, a menu will be shown.
`linux`	`[string]` Which CI provider(s) to use to test on Linux. Possible options are `"circle"`, `"ghactions"`, `"none"`/`NULL` and `"all"`.
`mac`	`[string]` Which CI provider(s) to use to test on macOS Possible options are `"none"`/`NULL` and `"ghactions"`.
`windows`	`[string]` Which CI provider(s) to use to test on Windows Possible options are `"none"`/`NULL`, and `"ghactions"`.
`deploy`	`[string]` Which CI provider(s) to use to deploy artifacts such as pkgdown documentation. Possible options are "circle"`, `"ghactions"`, `"none"`/`NULL`and`"all"‘.
`matrix`	`[string]` For which CI provider(s) to set up matrix builds. Possible options are `"circle"`, `"ghactions"`, `"none"`/`NULL` and `"all"`.
`private_key_name`	`string` Only needed when deploying from builds on GitHub Actions. If you have set a custom name for the private key during creation of the SSH key pair via tic::use_ghactions_deploy()] or `use_tic()`, pass this name here.
`quiet`	`[flag]` Less verbose output? Default: `FALSE`.

Details

1. Query information which CI providers should be used
2. Setup permissions for providers selected for deployment
3. Create YAML files for selected providers
4. Create a default `tic.R` file depending on the repo type (package, website, bookdown, ...)

Examples

```
# Requires interactive mode
if (FALSE) {
  use_tic()

  # Pre-specified settings favoring Circle CI:
  use_tic(
    wizard = FALSE,
    linux = "circle",
    mac = "ghactions",
    windows = "ghactions",
    deploy = "circle",
    matrix = "all"
  )
}
```

`use_tic_badge` *Add a CI Status Badge to README files*

Description

Adds a CI status badge to `README.Rmd` or `README.md`. By default the label is `"tic"`.

A custom branch can be specified via argument `branch`.

Usage

```
use_tic_badge(provider, branch = NULL, label = "tic")
```

Arguments

`provider`	`character(1)` The CI provider to generate a badge for. Only `ghactions` is currently supported
`branch`	`character(1)` Which branch should the badge represent? Defaults to the default repo branch.
`label`	`character(1)` Text to use for the badge.

Examples

```
## Not run:
use_tic_badge(provider = "ghactions")

# use a different branch
use_tic_badge(provider = "ghactions", branch = "develop")

## End(Not run)
```

use_tic_r *Add a tic.R file to the repo*

Description

Adds a `tic.R` file to containing the macros/steps/stages to be run during CI runs.

The content depends on the repo type (detected automatically when used within `use_tic()`).

Usage

```
use_tic_r(repo_type, deploy_on = "none")
```

Arguments

`repo_type`	`(character(1))` Which type of template should be used. Possible values are `"package"`, `"site"`, `"blogdown"`, `"bookdown"` or `"unknown"`.
`deploy_on`	`(character(1))` Which CI provider should perform deployment? Defaults to `NULL` which means no deployment will be done. Possible values are `"ghactions"` or `"circle"`.

See Also

yaml_templates, `use_tic_badge()`

Examples

```
## Not run:
use_tic_r("package")
use_tic_r("package", deploy_on = "ghactions")
use_tic_r("blogdown", deploy_on = "all")

## End(Not run)
```

use_update_tic *Update tic Templates*

Description

Adds a GitHub Actions workflow (`update-tic.yml`) to check for tic template updates once a day.

Internally, `update_yml()` is called. A Pull Request will be opened if a newer upstream version of the local tic template is found.

This workflow relies on a GITHUB_PAT with "workflow" scopes if GitHub Actions templates should be updated. Generate a GITHUB PAT and add it as a secret to your repo with `gha_add_secret()`.

Usage

```
use_update_tic()
```

Examples

```
## Not run:
use_update_tic()

## End(Not run)
```

`yaml_templates` *Use CI YAML templates*

Description

Installs YAML templates for various CI providers. These functions are also used within `use_tic()`.

If you want to update an existing template use `update_yml()`.

Usage

```
use_circle_yml(type = "linux-deploy", write = TRUE, quiet = FALSE)

use_ghactions_yml(type = "linux-deploy", write = TRUE, quiet = FALSE)
```

Arguments

`type`	`[character]` Which template to use. The string should be given following the logic `<platform>-<action>`. See details for more.
`write`	`[logical]` Whether to write the template to disk (`TRUE`) or just return it (`FALSE`).
`quiet`	`[logical]` Whether to print informative messages.

pkgdown

If `type` contains "deploy", tic by default also sets the environment variable `BUILD_PKGDOWN=true`. This triggers a call to `pkgdown::build_site()` via the do_pkgdown macro in `tic.R` for the respective runners.

If a setting includes "matrix" and builds on multiple R versions, the job building on R release is chosen to build the pkgdown site.

YAML Type

`tic` supports a variety of different YAML templates which follow the `<platform>-<action>` pattern. The first one is mandatory, the others are optional.

- Possible values for `<platform>` are `linux`, and `macos`, `windows`.
- Possible values for `<action>` are `matrix` and `deploy`.

Special types are `custom` and `custom-deploy`. These should be used if the runner matrix is completely user-defined. This is mainly useful in `update_yml()`.

For backward compatibility `use_ghactions_yml()` will be default build and deploy on all platforms.

Here is a list of all available combinations:

Provider	Operating system	Deployment	multiple R versions	Call
Circle	Linux	no	no	`use_circle_yml("linux")`
	Linux	yes	no	`use_circle_yml("linux-deploy")`
	Linux	no	yes	`use_circle_yml("linux-matrix")`
	Linux	no	yes	`use_circle_yml("linux-deploy-matri`
———-	———————	———-	——————-	—————————————
GH Actions	Linux	no	no	`use_ghactions_yml("linux")`
	Linux	yes	no	`use_ghactions_yml("linux-deploy")`
	custom	no	no	`use_ghactions_yml("custom")`
	custom-deploy	yes	no	`use_ghactions_yml("custom-deploy")`
	macOS	no	no	`use_ghactions_yml("macos")`
	macOS	yes	no	`use_ghactions_yml("macos-deploy")`
	Windows	no	no	`use_ghactions_yml("windows")`
	Windows	yes	no	`use_ghactions_yml("windows-deploy"`
	Linux + macOS	no	no	`use_ghactions_yml("linux-macos")`
	Linux + macOS	yes	no	`use_ghactions_yml("linux-macos-dep`
	Linux + Windows	no	no	`use_ghactions_yml("linux-windows")`
	Linux + Windows	yes	no	`use_ghactions_yml("linux-windows-d`
	macOS + Windows	no	no	`use_ghactions_yml("macos-windows")`
	macOS + Windows	yes	no	`use_ghactions_yml("macos-windows-d`
	Linux + macOS + Windows	no	no	`use_ghactions_yml("linux-macos-win`
	Linux + macOS + Windows	yes	no	`use_ghactions_yml("linux-macos-win`

Index