Package: containr (via r-universe)

October 10, 2026

Title Containerize Your 'R' Project

Version 0.2.0.9000

Description Provides tools for containerizing 'R' projects. Reads an 'renv' lock file, generates a ready-to-use 'Dockerfile', builds the container image, and pushes it to a registry. Generated 'COPY' instructions preserve the local directory structure inside the container. Cross-platform builds are supported via 'Docker' buildx for targeting 'linux/amd64' from ARM hosts. Designed to help researchers build portable, reproducible workflows that can be reliably shared, archived, and rerun across systems. See R Core Team (2025) <https://www.R-project.org/>, Ushey et al. (2025) <https://CRAN.R-project.org/package=renv>, and Docker Inc. (2025) <https://www.docker.com/>.

License Apache License (>= 2)

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

URL https://github.com/erwinlares/containr, https://erwinlares.github.io/containr/

BugReports https://github.com/erwinlares/containr/issues

Suggests knitr, rmarkdown, spelling, testthat (>= 3.0.0), withr

Imports cli, fs, glue, httr2, jsonlite, purrr, readr, remotes, renv, yaml

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

VignetteBuilder knitr

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

Date/Publication 2026-10-10 00:17:49 UTC

RemoteUrl https://github.com/erwinlares/containr

RemoteRef HEAD

RemoteSha 63a2b06295e698bd087f1e2e1a97aebf6a0a658c

Contents

build_image 2
generate_dockerfile 5
list_images 10
push_image 12

Index **16**

`build_image` *Build a container image from a Dockerfile*

Description

`build_image()` builds a container image from a `Dockerfile` using either `podman` or `docker`. It auto-detects which tool is available on the system unless `tool_preference` is set to a single value. Use `dry_run = TRUE` to preview the exact command that would be run without executing it.

Usage

```
build_image(
  dockerfile = "Dockerfile",
  tag = NULL,
  platform = "linux/amd64",
  tool_preference = c("podman", "docker"),
  dry_run = FALSE,
  verbose = FALSE,
  comments = FALSE
)
```

Arguments

`dockerfile`	A character string. Path to the `Dockerfile` to build from. Defaults to `"Dockerfile"` in the current working directory.
`tag`	A character string or `NULL`. The full image tag to assign to the built image, including the registry prefix, e.g. `"registry.doit.wisc.edu/namespace/myimage"`. If `NULL`, no tag is applied and the image is identified only by its image ID. Defaults to `NULL`.
`platform`	A character string or `NULL`. The target platform for the container image. Defaults to `"linux/amd64"`, which is the architecture used by most HPC and HTC clusters. Set to `"linux/arm64"` for ARM-based systems (Apple Silicon, AWS Graviton). Set to `NULL` to let the container tool build for the host architecture.

`tool_preference`
: A non-empty character vector of container tools to try, in order. Defaults to `c("podman", "docker")` – Podman first, then Docker. Supply a single value (e.g. `"docker"`) to require that specific tool rather than auto-detecting.

`dry_run`
: Logical. If `TRUE`, prints the command that would be run without executing it. Useful for verifying the command before committing to a potentially slow build. Defaults to `FALSE`.

`verbose`
: Logical. If `TRUE`, prints progress messages at each step. Defaults to `FALSE`.

`comments`
: Logical. If `TRUE`, prints explanatory context before each step – what the command does, why it is needed, and common pitfalls. Useful for first-time users learning the container build workflow. Defaults to `FALSE`. This matches how `comments` is used across `submitr` – printed guidance, nothing written to a file. The one exception in this family is `generate_dockerfile()`'s own `comments` argument (C15), which writes annotations into the generated `Dockerfile` instead.

Details

When the target `platform` differs from the host architecture (e.g. building `linux/amd64` on an Apple Silicon Mac), `build_image()` automatically uses `docker buildx build` instead of `docker build`, and includes `--load` to ensure the image is available in the local store. For `podman`, `--platform` is passed directly to `podman build`.

Value

If `tag` was supplied, the same character string, returned invisibly – so it can be captured or piped straight into `push_image(image_id = )` without retyping it (C-G1). If `tag` is `NULL` (the default), there is no name to hand back yet; returns `invisible(NULL)`, and `list_images()` is how you find the built image's ID.

Prerequisites

Before calling `build_image()`, ensure the following are in place:

1. A `Dockerfile` exists at `dockerfile`. Use `generate_dockerfile()` to create one if needed.
2. An `renv.lock` file is present in the build context – the generated `Dockerfile` uses it to restore the R package environment inside the container.
3. Either `podman` or `docker` is installed and the daemon (for `docker`) or the Podman service is running. Verify with `podman info` or `docker info` in a terminal.

Cross-platform builds

Building for a different architecture than the host requires emulation. On Apple Silicon Macs, building `linux/amd64` images uses QEMU emulation under Podman, which can be slow and unstable. Docker Desktop handles cross-platform builds more reliably via `buildx` and Rosetta 2.

If builds fail with QEMU segfaults, consider:

- Using Docker Desktop instead of Podman (`tool_preference = "docker"`)

- Building on a native x86_64 machine (e.g. via GitHub Actions)
- Building directly on the target cluster if it supports container builds

Tagging convention for the DoIT GitLab Container Registry

For UW-Madison's DoIT GitLab Container Registry (the registry `push_image()` currently supports), the full tag format is: `registry.doit.wisc.edu/<namespace>/<image-name>:<version>`

For example: `registry.doit.wisc.edu/erwin.lares/my-analysis:1.0.0`

The version tag defaults to `latest` if omitted. Using explicit version tags (e.g. `1.0.0`) is recommended for reproducibility – `latest` will be overwritten each time you push.

Examples

```
## Not run:
# Build for linux/amd64 (default) with auto-detected tool
build_image()

# Build and tag for the DoIT GitLab Container Registry
build_image(tag = "registry.doit.wisc.edu/namespace/my-analysis:1.0.0")

# Build for the host architecture (no --platform flag)
build_image(platform = NULL)

# Build for ARM64 (e.g. local use on Apple Silicon)
build_image(platform = "linux/arm64")

# Preview the build command without running it
build_image(
  tag     = "registry.doit.wisc.edu/namespace/my-analysis:1.0.0",
  dry_run = TRUE
)

# Guided build for first-time users
build_image(
  tag      = "registry.doit.wisc.edu/namespace/my-analysis:1.0.0",
  verbose  = TRUE,
  comments = TRUE
)

# Chain straight into push_image() -- build_image() returns its own
# tag argument invisibly, and push_image()'s first parameter is
# image_id, so a simple local tag flows through the pipe into the
# full registry destination with nothing retyped or captured in an
# intermediate variable (C-G1)
build_image(tag = "my-analysis:dev") |>
  push_image(namespace = "erwin.lares", project = "my-analysis", tag = "1.0.0")

## End(Not run)
```

generate_dockerfile *Generate a reproducible Dockerfile for an R project*

Description

`generate_dockerfile()` inspects an R project's dependencies via an `renv` lockfile and writes a ready-to-use `Dockerfile` to the specified output directory. It supports multiple Rocker base images, automatic system library detection, Quarto installation, file copying, user creation, and inline documentation comments.

Usage

```
generate_dockerfile(
  r_version = "current",
  r_mode = "base",
  auto_syslibs = TRUE,
  install_syslibs = NULL,
  output = ".",
  data_file = NULL,
  code_file = NULL,
  misc_file = NULL,
  add_user = NULL,
  home_dir = "/home",
  expose_port = "8787",
  install_quarto = FALSE,
  quarto_version = "latest",
  os_version = NULL,
  comments = FALSE,
  verbose = FALSE,
  config = NULL,
  binary_packages = TRUE,
  ppm_snapshot = "latest"
)
```

Arguments

`r_version`	A character string specifying the R version to use, e.g. `"4.3.0"`. Defaults to `"current"`, which resolves to the version of R running in the current session.
`r_mode`	A character string selecting the Rocker base image. Inspired by the Rocker Project. One of `"base"` for plain R, `"tidyverse"` for R with the tidyverse, `"rstudio"` for RStudio Server, `"verse"` for tidyverse plus TeX Live and publishing-related packages, `"shiny_server"` for serving Shiny apps, or `"rstudio_shiny"` for RStudio Server with Shiny Server layered on top. Defaults to `"base"`.
`auto_syslibs`	Logical. If `TRUE` (the default), reads `renv.lock` from the current working directory, queries the Posit Package Manager sysreqs database via `remotes::system_requirements()`, and automatically includes the system libraries required by all packages in the

lock file. Warns and continues without auto-detection if the lookup fails. Set to FALSE to skip auto-detection entirely.

install_syslibs

A character vector or NULL. Additional system libraries to install beyond those auto-detected from renv.lock. Each element should be a valid apt package name, e.g. c("libuv1-dev", "libwebp-dev"). Defaults to NULL.

output

A character string. Directory path where the Dockerfile will be written. Defaults to ".", the current working directory – the same directory build_image() treats as the build context by default, so the two functions' defaults compose without either argument having to be supplied. A Dockerfile written somewhere else (tempdir(), say) would have to be moved into the build context before build_image() could find it, since the build context is always getwd(). output is created automatically, along with any missing parent directories, if it does not already exist.

data_file

A character vector or NULL. Path(s) to data file(s) and/or directories to copy into the container – a single path, a vector of paths, or a directory (copied whole, with its contents) may all be mixed freely in the same vector. The local directory structure is preserved under home_dir for "base", "tidyverse", "rstudio", and "verse" (e.g. with the default home_dir = "/home", "data-raw/sample.csv" becomes /home/data-raw/sample.csv, and a directory "data-raw/" is copied to /home/data-raw/ in full), or under /srv/shiny-server/ for "shiny_server" and "rstudio_shiny", matching Shiny Server's own default app directory regardless of home_dir. Every path must be inside the current working directory (the build context). Defaults to NULL.

code_file

A character vector or NULL. Path(s) to script file(s) (e.g. .R, .qmd, .rmd) and/or directories to copy into the container – see data_file for vector and directory behavior. The local directory structure is preserved under the mode's copy root (home_dir for four of the six modes) – see data_file. Every path must be inside the current working directory. Defaults to NULL. Leave it out for an image that will run cluster jobs: in that workflow the analysis script travels with each job (submitr uploads it alongside the data subset), so the image holds the environment and the data, and editing the script never means rebuilding the image. Pass code_file when the image itself has to run the code – a Shiny app, or a self-contained image handed to a collaborator. config never fills it in; see config.

misc_file

A character vector or NULL. Path(s) to miscellaneous file(s) (e.g. images, shell scripts, or branding assets) and/or directories to copy into the container – see data_file for vector and directory behavior. The local directory structure is preserved under the mode's copy root (home_dir for four of the six modes) – see data_file. Every path must be inside the current working directory. Defaults to NULL. If the project was scaffolded with toolero::init_project() using branding = TRUE or branding = "uw-madison", the generated .qmd will reference assets/styles.css, assets/header.html, and assets/footer.html at render time. Those files must be present inside the container or Quarto will error on render. Pass misc_file = "assets/" to copy the entire branding folder in one step. Additional files and directories can be combined freely in the same vector, e.g. misc_file = c("assets/", "extra-script.sh").

add_user — A character string. Name of a Linux user to create inside the container with sudo access. Defaults to NULL.

home_dir — A character string. The working directory set inside the container via WORKDIR. For "base", "tidyverse", "rstudio", and "verse", this is also where data_file, code_file, and misc_file are copied (C24) – there is nowhere else a script running from WORKDIR could resolve its relative paths against. "shiny_server" and "rstudio_shiny" are the exception: their copy destination is fixed at /srv/shiny-server regardless of home_dir – see data_file. Defaults to "/home".

expose_port — A character string. Overrides the port exposed when r_mode is "rstudio". Defaults to "8787". Ignored for every other r_mode – "shiny_server" and "rstudio_shiny" expose their own fixed port(s) ("3838", and "8787"/"3838" respectively), since a single override value can't address more than one port.

install_quarto — Logical. If TRUE, downloads and installs the Quarto CLI inside the container. Defaults to FALSE. See quarto_version to pin a specific release rather than always installing whatever is currently latest.

quarto_version — Character string. Either "latest" (the default) or an explicit Quarto version, e.g. "1.5.57". Ignored unless install_quarto = TRUE. When "latest", the actual version is resolved at generation time via the Quarto releases API and recorded in the generated Dockerfile as ENV QUARTO_VERSION=..., so a later rebuild from the same Dockerfile reproduces the same Quarto version rather than whatever happens to be current at build time – consistent with how r_version and renv.lock are pinned elsewhere in the image. An explicit version is validated against the Quarto releases API and errors if no matching release exists.

os_version — A character string or NULL. The Ubuntu version to query against when looking up system requirements for auto_syslibs (C14). When NULL (the default), it is derived from the resolved r_version via the Rocker Project's own R-version-to-Ubuntu-release mapping – see .resolve_os_version() – rather than left at a single hardcoded value that only matched the Ubuntu release actually backing some R versions and not others. Supplying a value overrides the derivation entirely, for a project that needs to query against a different Ubuntu release than the one its r_version would normally resolve to. If remotes does not accept the release (its list lags behind Ubuntu: remotes 2.5.0 stops at "22.04", while R 4.4.2 and later are backed by "24.04"), the lookup uses the newest release remotes does accept that is not newer than this one, and says so in a message. Ignored when auto_syslibs = FALSE, since no sysreqs lookup happens in that case.

comments — Logical. If TRUE, annotates each Dockerfile instruction with an explanatory comment, written on the line above the instruction it describes. Useful for learning or sharing. Defaults to FALSE. Note (C15): this is the one comments argument in the family that writes into a generated file rather than printing to the console – build_image() and push_image() in this same package, and every comments argument in submitr, use it to print explanatory guidance to the console instead. Keep that distinction in mind when moving between these functions. The comment-then-instruction ordering matches submitr's own generated .sh and .sub files (S09), so a reader moving between a generated Dockerfile and a submitr-generated script reads both the same way round: explanation first, instruction second.

verbose — Logical. If TRUE, prints progress messages as each section of the Dockerfile is written. Defaults to FALSE.

config — A character string. Path to a _toolero.yml project config, such as the one toolero::init_project() writes. When supplied, fills in data_file and misc_file from the config's declared folders: – but only an argument left at its own NULL default. An argument you do supply always wins; config never overrides an explicit call. data_file is derived from "data-raw" when present, the folder this family's own documentation uses as the canonical home for input data – there is no dedicated config key naming it, so this one is containr's own convention rather than something the file states explicitly; misc_file is derived from "assets" when present, the branding folder init_project(branding = ...) creates. code_file is deliberately never derived, and the config's script_dir convention is not consulted: the analysis script travels with each cluster job rather than living in the image (see code_file), so a script copied in because the project happens to have an R/ folder would be a second, stale copy that nothing runs. Pass code_file yourself when the image does need it. Reading _toolero.yml is not a dependency on toolero: the schema is the contract, and a config written by hand is as valid an input as one init_project() created. In verbose mode, reports which of data_file and misc_file came from config rather than from the call, and notes that code_file did not, since a generated Dockerfile whose COPY lines came from somewhere invisible to the caller undermines the reproducibility this package exists to support. A schema_version the file does not declare is treated as schema 1; any other declared value produces a warning, not an abort, and the file is still read on a best-effort basis either way. When supplied, config also adds a RUN mkdir -p instruction, right after WORKDIR, creating every folder the config's folders: declares under home_dir (C07) – so a script's first write into one of them (via toolero::save_output(), or a bare ggsave()/write.csv() call) does not fail the way it would on a fresh checkout with no config supplied. This is unconditional on which folders those are; a folder this version of containr has no other special meaning for is still created. Defaults to NULL, so nothing about this argument changes the behavior of a call that does not use it.

binary_packages — Logical. If TRUE (the default), the image restores R packages from Posit Package Manager's precompiled Linux binaries instead of compiling each one from source. A source restore compiles every package in renv.lock, which is slow, especially under cross-architecture emulation; a binary restore skips the compilation. It does not skip the system libraries: the binaries are still linked against them, so the apt-get block that auto_syslibs and install_syslibs produce matters just as much as it does for a source restore (the -dev packages in it also pull in the matching runtime libraries). The Package Manager repository is chosen for the Ubuntu release that backs r_version, not for os_version, which only steers the system-requirements lookup, since a binary built for one release is not guaranteed to load on another. A package version the repository has no binary for is installed from source instead. If no repository exists for the image's Ubuntu release, a warning says so and the restore stays a source restore. Set to FALSE to restore from the repositories recorded in renv.lock.

ppm_snapshot — A character string. Which Package Manager snapshot binary_packages = TRUE

restores from: "latest" (the default), or a date in "YYYY-MM-DD" form. "latest" serves the current version of each package, so a lockfile written some time ago may record versions with no binary left in that snapshot; renv then installs those from the CRAN archive, which compiles them and needs their system libraries. Pin a date only when you want the repository frozen at a known day. Ignored when binary_packages = FALSE.

Value

Called for its side effects. Writes a Dockerfile to output. Returns invisible(NULL).

Prerequisites

generate_dockerfile() requires an renv.lock file in the current working directory. Create one with renv::snapshot() before calling this function. If the lock file is out of sync with your project library, a warning is issued – run renv::snapshot() to update it before building the image.

If the project uses Quarto with branding assets (i.e. toolero::create_qmd() was called with use_style = TRUE), the assets/ folder must be copied into the container alongside the .qmd file or Quarto will be unable to resolve the CSS and HTML includes at render time. The simplest way to ensure this is to pass misc_file = "assets/" – or misc_file = c("assets/", other_files) if additional files are needed – when calling generate_dockerfile(). No code change is required; misc_file already accepts directories and copies them whole.

Examples

```
## Not run:
# Requires renv.lock in the current working directory.
# Run renv::snapshot() first if you don't have one.

# Generate a minimal Dockerfile using a pinned R version. output defaults
# to ".", so this writes to the current working directory -- the same
# place build_image() looks by default.
generate_dockerfile(r_version = "4.4.0")

# Pin a specific R version with the tidyverse image
generate_dockerfile(r_version = "4.3.0", r_mode = "tidyverse")

# Add extra system libraries on top of auto-detected ones
generate_dockerfile(
  r_version       = "4.4.0",
  install_syslibs = c("libuv1-dev", "libwebp-dev"),
  output          = "."
)

# Include a data file -- directory structure is preserved in the container.
# No code_file: for cluster jobs the script travels with each job.
generate_dockerfile(
  r_version = "4.3.0",
  data_file = "data-raw/penguins.csv",
  comments  = TRUE,
  output    = "."
```

```
)

# Several data files and a whole assets folder -- pass assets/ via
# misc_file so that branding files (styles.css, header.html, footer.html)
# are present inside the container when Quarto renders the .qmd
generate_dockerfile(
  r_version = "4.3.0",
  data_file = c("data-raw/penguins.csv", "data-raw/islands.csv"),
  misc_file = "assets/",
  output    = "."
)

# Serve a Shiny app -- the image runs the app itself, so its code goes in
# via code_file. Files land under /srv/shiny-server/ automatically
generate_dockerfile(
  r_version = "4.3.0",
  r_mode    = "shiny_server",
  code_file = "app.R",
  output    = "."
)

# RStudio Server plus Shiny Server in the same image
generate_dockerfile(
  r_version = "4.3.0",
  r_mode    = "rstudio_shiny",
  code_file = "app.R",
  output    = "."
)

# Install Quarto, pinned to a specific release rather than whatever is
# currently latest -- the resolved version is recorded as
# ENV QUARTO_VERSION in the generated Dockerfile either way
generate_dockerfile(
  r_version      = "4.3.0",
  install_quarto = TRUE,
  quarto_version = "1.5.57",
  output         = "."
)

# Fill in data_file and misc_file from a toolero project config instead
# of retyping paths the project already declares (code_file is never
# filled in from config)
generate_dockerfile(
  r_version = "4.3.0",
  config    = "_toolero.yml",
  output    = "."
)

## End(Not run)
```

`list_images` *List locally available container images*

Description

`list_images()` returns a data frame of container images currently stored in the local image store, as reported by `podman image ls` or `docker image ls`. This is useful for finding the image ID to pass to `push_image()` after building an image with `build_image()`.

Usage

```
list_images(tool_preference = c("podman", "docker"), verbose = FALSE)
```

Arguments

`tool_preference`	A non-empty character vector of container tools to try, in order. Defaults to `c("podman", "docker")` – Podman first, then Docker. Supply a single value (e.g. `"docker"`) to require that specific tool rather than auto-detecting.
`verbose`	Logical. If `TRUE`, prints a progress message before querying the local image store, and prints the resulting data frame (C17) – matching every other user-facing message in the package, which is `verbose`-gated rather than unconditional. Defaults to `FALSE`, so `imgs <- list_images()` assigns quietly rather than also printing as a side effect.

Value

A data frame with five columns: `repository`, `tag`, `image_id`, `created`, and `size`. Rows where both `repository` and `tag` are `<none>` correspond to untagged images produced by `build_image()` when no `tag` argument was supplied. Returns an empty data frame if no images are found.

Finding your image ID

After calling `build_image()`, run `list_images()` to find the image ID of the image you just built. Untagged images appear with `<none>` in the `repository` and `tag` columns – the `image_id` column contains the hash you need to pass to `push_image()`:

```
imgs <- list_images()
push_image(
  image_id  = imgs$image_id[1],
  namespace = "erwin.lares",
  project   = "my-analysis"
)
```

Examples

```
## Not run:
# List all local images and print them (verbose = TRUE)
list_images(verbose = TRUE)

# Capture the result for programmatic use, without printing (C17)
imgs <- list_images()
imgs$image_id[1]
```

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

---

push_image *Tag and push a container image to a registry*

Description

push_image() tags a locally built container image with a full registry path and pushes it to a container registry. It handles both the podman tag and podman push steps in a single call. Auto-detects which container tool is available unless tool_preference is set to a single value. Use dry_run = TRUE to preview the exact commands without executing them. The full path format is {registry}/{namespace}/{project}:{tag} – for the default registry, e.g. registry.doit.wisc.edu/<namespace>/<pr

Usage

```
push_image(
  image_id = NULL,
  namespace = NULL,
  project = NULL,
  tag = "latest",
  registry = "registry.doit.wisc.edu",
  tool_preference = c("podman", "docker"),
  check_login = TRUE,
  dry_run = FALSE,
  verbose = FALSE,
  comments = FALSE
)
```

Arguments

image_id	A character string. The local image ID or name to push, as shown in podman image ls or docker image ls. This is typically a 12-character hash (e.g. "974123909a36") or a locally assigned name if the image was built with a tag via build_image().
namespace	A character string. The account or organization the image is scoped under, used to construct the full registry path. For the default "registry.doit.wisc.edu" registry, this is your UW-Madison NetID (e.g. "erwin.lares"). For other registries, it's the equivalent identifier – your GitHub username or organization for ghcr.io, or your Quay namespace for quay.io.
project	A character string. The image's name – the third segment of the full registry path, e.g. "my-analysis" in registry.doit.wisc.edu/erwin.lares/my-analysis. For the default DoIT GitLab registry, the project *is* the image: name it after the analysis it holds (C18), rather than after a single shared project set up to host the registry generally.
tag	A character string. The version tag to assign to the image. Defaults to "latest". Using explicit version tags (e.g. "1.0.0") is recommended for reproducibility – "latest" is overwritten on every push.

`registry` A character string. The registry hostname. Defaults to `"registry.doit.wisc.edu"`, UW-Madison DoIT's GitLab Container Registry – the only registry `push_image()` currently supports.

`tool_preference` A non-empty character vector of container tools to try, in order. Defaults to `c("podman", "docker")` – Podman first, then Docker. Supply a single value (e.g. `"docker"`) to require that specific tool rather than auto-detecting.

`check_login` Logical. If `TRUE` (the default), verifies that you are logged in to `registry` before attempting the push. If not logged in, the function errors with instructions on how to authenticate.

`dry_run` Logical. If `TRUE`, prints the commands that would be run without executing them. Defaults to `FALSE`.

`verbose` Logical. If `TRUE`, prints progress messages at each step. Defaults to `FALSE`.

`comments` Logical. If `TRUE`, prints explanatory context before each step – what the command does, why it is needed, and common pitfalls. Useful for first-time users learning the container push workflow. Defaults to `FALSE`. This matches how `comments` is used across `submitr` – printed guidance, nothing written to a file. The one exception in this family is `generate_dockerfile()`'s own `comments` argument (C15), which writes annotations into the generated `Dockerfile` instead.

Value

The full registry URI the image was tagged and pushed as (or would be, under `dry_run = TRUE`) – `"{registry}/{namespace}/{project}:{tag}"`, e.g. `"registry.doit.wisc.edu/erwin.lares/my-analysis:1.0.0"` – as a character string, returned invisibly. Capture it to hand off to `submitr::htc_gen_submit(container_image = )` without reassembling or retyping the URI by hand (C-G1).

Prerequisites

Before calling `push_image()` against the default `"registry.doit.wisc.edu"` registry, ensure the following are in place. (For other registries – `ghcr.io`, `quay.io`, or any other hostname supplied via `registry` – consult that registry's own documentation for the equivalent setup; the steps below are DoIT GitLab-specific.)

1. The image has been built locally with `build_image()`. Run `podman image ls` to find the image ID.
2. You have a GitLab account at `git.doit.wisc.edu` and a project with the container registry enabled.
3. You have a Personal Access Token (PAT) with `read_registry` and `write_registry` scopes. Create one at: `https://git.doit.wisc.edu/-/user_settings/personal_access_tokens`
4. You are logged in to the registry. Authenticate once in a terminal: `podman login registry.doit.wisc.edu` Enter your NetID as the username and your PAT as the password.

Authentication

The DoIT GitLab Container Registry requires authentication before pushing. Use a Personal Access Token (PAT) rather than your NetID password – PATs can be scoped to registry access only and revoked independently. Authentication is cached by `podman` or `docker` after the first login, so you only need to run `podman login` once per machine per session. For other registries, use whatever credential type that registry expects – e.g. a GitHub PAT with the `write:packages` scope for `ghcr.io`.

Note that GitLab Self-Managed authentication tokens expire after five minutes by default. If you see an `unauthorized: authentication required` error mid-push on a large image, re-authenticate and push again.

Examples

```
## Not run:
# Tag and push an image to the DoIT GitLab Container Registry --
# project is the image's own name, not a project that merely hosts
# the registry (C18)
push_image(
  image_id  = "974123909a36",
  namespace = "erwin.lares",
  project   = "my-analysis"
)

# Push with an explicit version tag
push_image(
  image_id  = "974123909a36",
  namespace = "erwin.lares",
  project   = "my-analysis",
  tag       = "1.0.0"
)

# Preview the commands without running them
push_image(
  image_id  = "974123909a36",
  namespace = "erwin.lares",
  project   = "my-analysis",
  dry_run   = TRUE
)

# Guided push for first-time users
push_image(
  image_id  = "974123909a36",
  namespace = "erwin.lares",
  project   = "my-analysis",
  verbose   = TRUE,
  comments  = TRUE
)

# Push to GitHub Container Registry instead of the default
push_image(
  image_id  = "974123909a36",
```

```
  namespace = "your-github-username",
  project   = "my-analysis",
  registry  = "ghcr.io"
)

# Capture the full registry URI to hand off to submitr -- e.g.
# submitr::htc_gen_submit(container_image = uri) -- instead of
# reassembling it by hand from registry/namespace/project/tag (C-G1)
uri <- push_image(
  image_id  = "974123909a36",
  namespace = "erwin.lares",
  project   = "my-analysis",
  tag       = "1.0.0"
)
uri
#> [1] "registry.doit.wisc.edu/erwin.lares/my-analysis:1.0.0"

## End(Not run)
```

Index

build_image, 2
build_image(), *7*, *11–13*

generate_dockerfile, 5
generate_dockerfile(), *3*, *13*

list_images, 10
list_images(), *3*

push_image, 12
push_image(), *7*, *11*

toolero::create_qmd(), *9*
toolero::init_project(), *6*, *8*