Package ‘googleCloudRunner’

October 13, 2022

Type Package

Title R Scripts in the Google Cloud via Cloud Run, Cloud Build and Cloud Scheduler

Version 0.5.0

Description Tools to easily enable R scripts in the Google Cloud Platform. Utilise cloud services such as Cloud Run <https://cloud.google.com/run/> for R over HTTP, Cloud Build <https://cloud.google.com/build> for Continuous Delivery and Integration services and Cloud Scheduler <https://cloud.google.com/scheduler/> for scheduled scripts.

URL https://code.markedmondson.me/googleCloudRunner/

BugReports https://github.com/MarkEdmondson1234/googleCloudRunner/issues

Depends R (>= 3.3.0)

Imports assertthat (>= 0.2.0), cli (>= 2.0.2), curl (>= 4.3), googleAuthR (>= 2.0.0), googleCloudStorageR (>= 0.7.0), googlePubsubR (>= 0.0.2), httr (>= 1.4.1), jose (>= 1.0), jsonlite (>= 1.5), openssl (>= 1.4.1), plumber (>= 1.0.0), usethis (>= 1.6.0), utils, withr, yaml (>= 2.2.0)

Suggests knitr, markdown, miniUI, rmarkdown, rstudioapi, shiny, targets, testthat (>= 2.1.0)

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

Config/testthat/edition 3

Config/testthat/parallel false

NeedsCompilation no

Author Mark Edmondson [aut, cre] (<https://orcid.org/0000-0002-8434-3881>), Sunholo Ltd [cph]

Maintainer Mark Edmondson <r@sunholo.com>

Repository CRAN

Date/Publication 2022-02-28 09:40:01 UTC

R topics documented:

`Build` *Build Object*

Description

Build Object

Usage

```
Build(
  Build.substitutions = NULL,
  Build.timing = NULL,
  results = NULL,
  logsBucket = NULL,
  steps = NULL,
  buildTriggerId = NULL,
  id = NULL,
  tags = NULL,
  startTime = NULL,
  substitutions = NULL,
  timing = NULL,
  sourceProvenance = NULL,
  createTime = NULL,
  images = NULL,
  projectId = NULL,
  logUrl = NULL,
  finishTime = NULL,
  source = NULL,
  options = NULL,
  timeout = NULL,
  status = NULL,
  statusDetail = NULL,
  artifacts = NULL,
  secrets = NULL,
  availableSecrets = NULL,
  serviceAccount = NULL
)
```

Arguments

`Build.substitutions`	The Build.substitutions object or list of objects
`Build.timing`	The Build.timing object or list of objects
`results`	Output only
`logsBucket`	Google Cloud Storage bucket where logs should be written (see
`steps`	Required

`buildTriggerId`	Output only
`id`	Output only
`tags`	Tags for annotation of a ‘Build‘
`startTime`	Output only
`substitutions`	Substitutions data for ‘Build‘ resource
`timing`	Output only
`sourceProvenance`	Output only
`createTime`	Output only
`images`	A list of images to be pushed upon the successful completion of all build
`projectId`	Output only
`logUrl`	Output only
`finishTime`	Output only
`source`	A Source object specifying the location of the source files to build, usually created by cr_build_source
`options`	Special options for this build
`timeout`	Amount of time that this build should be allowed to run, to second
`status`	Output only
`statusDetail`	Output only
`artifacts`	Artifacts produced by the build that should be uploaded upon
`secrets`	Secrets to decrypt using Cloud Key Management Service [deprecated]
`availableSecrets`	preferred way to use Secrets, via Secret Manager
`serviceAccount`	service account email to be used for the build

Details

A build resource in the Cloud Build API.

At a high level, a ‘Build‘ describes where to find source code, how to build it (for example, the builder image to run on the source), and where to store the built artifacts.

Value

Build object

Build Macros

Fields can include the following variables, which will be expanded when the build is created:-

- $PROJECT_ID: the project ID of the build.
- $BUILD_ID: the autogenerated ID of the build.
- $REPO_NAME: the source repository name specified by RepoSource.

- $BRANCH_NAME: the branch name specified by RepoSource.
- $TAG_NAME: the tag name specified by RepoSource.
- $REVISION_ID or $COMMIT_SHA: the commit SHA specified by RepoSource or resolved from the specified branch or tag.
- $SHORT_SHA: first 7 characters of $REVISION_ID or $COMMIT_SHA.

See Also

Other Cloud Build functions: `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

BuildTrigger *BuildTrigger Object*

Description

Configuration for an automated build in response to source repositorychanges.

Usage

```
BuildTrigger(
  filename = NULL,
  name = NULL,
  tags = NULL,
  build = NULL,
  ignoredFiles = NULL,
  github = NULL,
  substitutions = NULL,
  includedFiles = NULL,
  disabled = NULL,
  sourceToBuild = NULL,
  triggerTemplate = NULL,
  webhookConfig = NULL,
  description = NULL,
  pubsubConfig = NULL
)
```

Arguments

`filename`	Path, from the source root, to a file whose contents is used for the build
`name`	User assigned name of the trigger
`tags`	Tags for annotation of a ‘BuildTrigger‘
`build`	Contents of the build template

`ignoredFiles`	ignored_files and included_files are file glob matches extended with support for `"**"`.
`github`	a GitHubEventsConfig object - mutually exclusive with `triggerTemplate`
`substitutions`	A named list of Build macro variables
`includedFiles`	If any of the files altered in the commit pass the ignored_files
`disabled`	If true, the trigger will never result in a build
`sourceToBuild`	A cr_buildtrigger_repo object (but no regex allowed for branch or tag) This field is currently only used by Webhook, Pub/Sub, Manual, and Cron triggers and is the source of the build will execute upon.
`triggerTemplate`	a RepoSource object - mutually exclusive with `github`
`webhookConfig`	WebhookConfig describes the configuration of a trigger that creates a build whenever a webhook is sent to a trigger's webhook URL.
`description`	Human-readable description of this trigger
`pubsubConfig`	PubsubConfig describes the configuration of a trigger that creates a build whenever a Pub/Sub message is published.

Value

BuildTrigger object

See Also

`https://cloud.google.com/build/docs/api/reference/rest/v1/projects.triggers`

Other BuildTrigger functions: `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

`cr_bucket_set` *Get/Set the Cloud Storage bucket for your Cloud Build Service*

Description

Can also use environment arg GCS_DEFAULT_BUCKET

Usage

```
cr_bucket_set(bucket)

cr_bucket_get()
```

Arguments

`bucket`	The GCS bucket

Examples

```
cr_bucket_get()
```

---

cr_build *Starts a build with the specified configuration.*

Description

This method returns a long-running ‘Operation‘, which includes the buildID. Pass the build ID to cr_build_status to determine the build status (such as ‘SUCCESS‘ or ‘FAILURE‘).

Usage

```
cr_build(
  x,
  source = NULL,
  timeout = NULL,
  images = NULL,
  substitutions = NULL,
  serviceAccount = NULL,
  artifacts = NULL,
  options = NULL,
  projectId = cr_project_get(),
  launch_browser = interactive()
)
```

Arguments

`x`	A cloudbuild.yaml file location or an R object that will be turned into yaml via as.yaml or a Build object created by cr_build_make or from a previous build you want to rerun.
`source`	A Source object specifying the location of the source files to build, usually created by cr_build_source
`timeout`	Amount of time that this build should be allowed to run, to second
`images`	A list of images to be pushed upon the successful completion of all build
`substitutions`	Substitutions data for ‘Build‘ resource
`serviceAccount`	service account email to be used for the build
`artifacts`	Artifacts produced by the build that should be uploaded upon
`options`	Special options for this build
`projectId`	ID of the project
`launch_browser`	Whether to launch the logs URL in a browser once deployed

See Also

Google Documentation for Cloud Build

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`

Examples

```
cr_project_set("my-project")
my_gcs_source <- cr_build_source(StorageSource("my_code.tar.gz",
  bucket = "gs://my-bucket"
))
my_gcs_source

my_repo_source <- cr_build_source(RepoSource("github_username_my-repo.com",
  branchName = "master"
))
my_repo_source
## Not run:

# build from a cloudbuild.yaml file
cloudbuild_file <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)

# asynchronous, will launch log browser by default
b1 <- cr_build(cloudbuild_file)

# synchronous waiting for build to finish
b2 <- cr_build_wait(b1)

# the same results
cr_build_status(b1)
cr_build_status(b2)

# build from a cloud storage source
build1 <- cr_build(cloudbuild_file,
  source = my_gcs_source
)
# build from a git repository source
build2 <- cr_build(cloudbuild_file,
  source = my_repo_source
)

# you can send in results for previous builds to trigger
# the same build under a new Id
# will trigger build2 again
cr_build(build2)

# a build with substitutions (Cloud Build macros)
```

```
cr_build(build2, substitutions = list(`_SUB` = "yo"))

## End(Not run)
```

cr_buildstep *Create a yaml build step*

Description

Helper for creating build steps for upload to Cloud Build

Usage

```
cr_buildstep(
  name,
  args = NULL,
  id = NULL,
  prefix = "gcr.io/cloud-builders/",
  entrypoint = NULL,
  dir = "",
  env = NULL,
  waitFor = NULL,
  volumes = NULL,
  secretEnv = NULL
)
```

Arguments

`name`	name of docker image to call appended to `prefix`
`args`	character vector of arguments
`id`	Optional id for the step
`prefix`	prefixed to name - set to "" to suppress. Will be suppressed if `name` starts with gcr.io or `*-docker.pkg.dev`
`entrypoint`	change the entrypoint for the docker container
`dir`	The directory to use, relative to /workspace e.g. /workspace/deploy/
`env`	Environment variables for this step. A character vector for each assignment
`waitFor`	Whether to wait for previous buildsteps to complete before running. Default it will wait for previous step.
`volumes`	volumes to connect and write to
`secretEnv`	A list of secrets stored in Secret Manager referred to in args via a `$$var`

Details

This uses R to make building steps for cloudbuild.yml files harder to make mistakes with, and also means you can program creation of cloud build steps for use in R or other languages. Various templates with common use cases of buildsteps are also available that wrap this function, refer to the "See Also" section.

WaitFor

By default each buildstep waits for the previous, but if you pass `"-"` then it will start immediately, or if you pass in a list of ids it will wait for previous buildsteps to finish who have that id. See Configuring Build Step Order for details.

Build Macros

Fields can include the following variables, which will be expanded when the build is created:-

- $PROJECT_ID: the project ID of the build.
- $BUILD_ID: the autogenerated ID of the build.
- $REPO_NAME: the source repository name specified by RepoSource.
- $BRANCH_NAME: the branch name specified by RepoSource.
- $TAG_NAME: the tag name specified by RepoSource.
- $REVISION_ID or $COMMIT_SHA: the commit SHA specified by RepoSource or resolved from the specified branch or tag.
- $SHORT_SHA: first 7 characters of $REVISION_ID or $COMMIT_SHA.

Or you can add your own custom variables, set in the Build Trigger. Custom variables always start with $_ e.g. $_MY_VAR

secretEnv

You can pass secrets that are stored in Secret Manager directly instead of using a dedicated buildstep via cr_buildstep_secret

Within the code passed to `args` those secrets are referred to via `$$SECRET_NAME`. If used then cr_build_yaml must also include the `availableSecrets` argument.

See Also

Creating custom build steps how-to guide

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
# creating yaml for use in deploying cloud run
image <- "gcr.io/my-project/my-image:$BUILD_ID"
cr_build_yaml(
  steps = c(
    cr_buildstep("docker", c("build", "-t", image, ".")),
    cr_buildstep("docker", c("push", image)),
    cr_buildstep("gcloud", c(
      "beta", "run", "deploy", "test1",
      "--image", image
    ))
  ),
  images = image
)

# use premade docker buildstep - combine using c()
image <- "gcr.io/my-project/my-image"
cr_build_yaml(
  steps = c(
    cr_buildstep_docker(image),
    cr_buildstep("gcloud",
      args = c(
        "beta", "run", "deploy",
        "test1", "--image", image
      )
    )
  ),
  images = image
)

# list files with a new entrypoint for gcloud
cr_build_yaml(steps = cr_buildstep("gcloud", c("-c", "ls -la"),
  entrypoint = "bash"
))

# to call from images not using gcr.io/cloud-builders stem
cr_buildstep("alpine", c("-c", "ls -la"), entrypoint = "bash", prefix = "")

# to add environment arguments to the step
cr_buildstep("docker", "version", env = c("ENV1=env1", "ENV2=$PROJECT_ID"))

# to add volumes wrap in list()
cr_buildstep("test", "ls", volumes = list(list(name = "ssh", path = "/root/.ssh")))
```

cr_buildstep_bash *Run a bash script in a Cloud Build step*

Description

Helper to run a supplied bash script, that will be copied in-line

Usage

```
cr_buildstep_bash(
  bash_script,
  name = "ubuntu",
  bash_source = c("local", "runtime"),
  escape_dollar = TRUE,
  ...
)
```

Arguments

`bash_script`	bash code to run or a filepath to a file containing bash code that ends with .bash or .sh
`name`	The image that will run the R code
`bash_source`	Whether the code will be from a runtime file within the source or at build time copying over from a local file in your session
`escape_dollar`	Default TRUE. This will turn `$` into `$$` within the script to avoid them being recognised as Cloud Build variables. Turn this off if you want that behaviour (e.g. `my_project="$PROJECT_ID"`)
`...`	Other arguments passed to cr_buildstep

Details

If you need to escape build parameters in bash scripts, you need to escape CloudBuild's substitution via `$$` and bash's substitution via `\$` e.g. `\$$PARAM`

See Also

Other Cloud Buildsteps: `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
bs <- cr_build_yaml(
  steps = cr_buildstep_bash("echo 'Hello'")
)
## Not run:
cr_build(bs)

## End(Not run)
```

---

cr_buildstep_compute_container

Buildstep to deploy to Google Compute Engine

Description

This build step adds some helpers to cr_buildstep_gcloud for deploying to VMs to GCE that will auto create a container within them and atytach it to the disk

Usage

```
cr_buildstep_compute_container(
  vm_name,
  container_image = "gcr.io/gcer-public/persistent-rstudio:latest",
  disk_name = paste0(vm_name, "-disk"),
  disk_mount_path = "/home",
  zone = "europe-west1-b",
  disk_size = "10GB",
  machine_type = "n1-standard-1",
  container_env = "",
  scopes = "cloud-platform",
  network = "default",
  gcloud_args = NULL
)

cr_buildstep_compute_rstudio(
  rstudio_user,
  rstudio_pw,
  vm_name = "rstudio",
  disk_name = "rstudio-disk",
  zone = "europe-west1-b",
  disk_size = "10GB",
  machine_type = "n1-standard-1",
  container_image = "gcr.io/gcer-public/persistent-rstudio:latest",
  network = "default"
)
```

Arguments

`vm_name`	Name of the VM you will create
`container_image`	The Docker image that will be launched in the VM
`disk_name`	Name of the disk that will be attached to the VM's container image
`disk_mount_path`	Where the disk will be attached to the container in the VM
`zone`	Which zone the VM will launch within
`disk_size`	The size of the disk
`machine_type`	The type of VM that will be launched
`container_env`	Environment variables set within the VM's container image
`scopes`	The GCE scopes that the VM will be launched with permission to use
`network`	The network the VM will use. The container will bridge into the same network
`gcloud_args`	Other gcloud arguments you send in e.g. `c("--boot-disk-device-name=boot-disk1","--boot-disk`
`rstudio_user`	The usename for the RStudio image the VM will launch
`rstudio_pw`	The password for the RStudio image the VM will launch

Examples

```
bs <- cr_buildstep_compute_rstudio("mark", "securepassword1234")
build <- cr_build_yaml(bs)
build
## Not run:

cr_build(build)

## End(Not run)
```

`cr_buildstep_decrypt` *Create a build step for decrypting files via KMS*

Description

Create a build step to decrypt files using CryptoKey from Cloud Key Management Service. Usually you will prefer to use cr_buildstep_secret

Usage

```
cr_buildstep_decrypt(cipher, plain, keyring, key, location = "global", ...)
```

Arguments

`cipher`	The file that has been encrypted
`plain`	The file location to decrypt to
`keyring`	The KMS keyring to use
`key`	The KMS key to use
`location`	The KMS location
`...`	Further arguments passed in to cr_buildstep

Details

Key Management Store can encrypt secret files for use within your later buildsteps.

Setup

You will need to set up the encrypted key using gcloud following the link from Google

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
cr_buildstep_decrypt("secret.json.enc",
  plain = "secret.json",
  keyring = "my_keyring",
  key = "my_key"
)
```

cr_buildstep_df *Convert a data.frame into cr_buildstep*

Description

Helper to turn a data.frame of buildsteps info into format accepted by cr_build

Usage

```
cr_buildstep_df(x)
```

Arguments

x	A data.frame of steps to turn into buildsteps, with at least name and args columns

Details

This helps convert the output of cr_build into valid cr_buildstep so it can be sent back into the API

If constructing arg list columns then I suppresses conversion of the list to columns that would otherwise break the yaml format

See Also

Other Cloud Buildsteps: cr_buildstep_bash(), cr_buildstep_decrypt(), cr_buildstep_docker(), cr_buildstep_edit(), cr_buildstep_extract(), cr_buildstep_gcloud(), cr_buildstep_gitsetup(), cr_buildstep_mailgun(), cr_buildstep_nginx_setup(), cr_buildstep_packagetests(), cr_buildstep_pkgdown(
cr_buildstep_run(), cr_buildstep_r(), cr_buildstep_secret(), cr_buildstep_slack(), cr_buildstep_targets(), cr_buildstep()

Examples

```
y <- data.frame(
  name = c("docker", "alpine"),
  args = I(list(c("version"), c("echo", "Hello Cloud Build"))),
  id = c("Docker Version", "Hello Cloud Build"),
  prefix = c(NA, ""),
  stringsAsFactors = FALSE
)
cr_buildstep_df(y)
```

`cr_buildstep_docker` *Create a build step to build and push a docker image*

Description

Create a build step to build and push a docker image

Usage

```
cr_buildstep_docker(
  image,
  tag = c("latest", "$BUILD_ID"),
  location = ".",
  projectId = cr_project_get(),
  dockerfile = "Dockerfile",
  kaniko_cache = FALSE,
  build_args = NULL,
  push_image = TRUE,
  ...
)
```

Arguments

`image`	The image tag that will be pushed, starting with gcr.io or created by combining with `projectId` if not starting with gcr.io
`tag`	The tag or tags to be attached to the pushed image - can use `Build` macros
`location`	Where the Dockerfile to build is in relation to `dir`
`projectId`	The projectId
`dockerfile`	Specify the name of the Dockerfile found at `location`
`kaniko_cache`	If TRUE will use kaniko cache for Docker builds.
`build_args`	additional arguments to pass to `docker build`, should be a character vector.
`push_image`	if `kaniko_cache = FALSE` and `push_image = FALSE`, then the docker image is simply built and not pushed
`...`	Further arguments passed in to cr_buildstep

Details

Setting `kaniko_cache = TRUE` will enable caching of the layers of the Dockerfile, which will speed up subsequent builds of that Dockerfile. See Using Kaniko cache

If building multiple tags they don't have to run sequentially - set `waitFor = "-"` to build concurrently

See Also

Other Cloud Buildsteps: cr_buildstep_bash(), cr_buildstep_decrypt(), cr_buildstep_df(), cr_buildstep_edit(), cr_buildstep_extract(), cr_buildstep_gcloud(), cr_buildstep_gitsetup(), cr_buildstep_mailgun(), cr_buildstep_nginx_setup(), cr_buildstep_packagetests(), cr_buildstep_pkgdown(
cr_buildstep_run(), cr_buildstep_r(), cr_buildstep_secret(), cr_buildstep_slack(), cr_buildstep_targets(), cr_buildstep()

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")

cr_buildstep_docker("gcr.io/my-project/my-image")
cr_buildstep_docker("my-image")
cr_buildstep_docker("my-image", tag = "$BRANCH_NAME")

# setting up a build to trigger off a Git source:
my_image <- "gcr.io/my-project/my-image"
my_repo <- RepoSource("github_markedmondson1234_googlecloudrunner",
  branchName = "master"
)
## Not run:
docker_yaml <- cr_build_yaml(steps = cr_buildstep_docker(my_image))
built_docker <- cr_build(docker_yaml, source = my_repo)

# make a build trigger so it builds on each push to master
cr_buildtrigger("build-docker", trigger = my_repo, build = built_docker)


# add a cache to your docker build to speed up repeat builds
cr_buildstep_docker("my-image", kaniko_cache = TRUE)

# building using manual buildsteps to clone from git
bs <- c(
  cr_buildstep_gitsetup("github-ssh"),
 cr_buildstep_git(c("clone", "git@github.com:MarkEdmondson1234/googleCloudRunner", ".")),
  cr_buildstep_docker("gcr.io/gcer-public/packagetools",
    dir = "inst/docker/packages/"
  )
)

built <- cr_build(cr_build_yaml(bs))

## End(Not run)
```

cr_buildstep_edit *Modify an existing buildstep with new parameters*

Description

Useful for editing existing buildsteps

Usage

```
cr_buildstep_edit(x, ...)
```

Arguments

`x`	A buildstep created previously

`...`	Arguments passed on to `cr_buildstep`
	`name` name of docker image to call appended to `prefix`
	`args` character vector of arguments
	`prefix` prefixed to name - set to "" to suppress. Will be suppressed if `name` starts with gcr.io or `*-docker.pkg.dev`
	`entrypoint` change the entrypoint for the docker container
	`dir` The directory to use, relative to /workspace e.g. /workspace/deploy/
	`id` Optional id for the step
	`env` Environment variables for this step. A character vector for each assignment
	`volumes` volumes to connect and write to
	`waitFor` Whether to wait for previous buildsteps to complete before running. Default it will wait for previous step.
	`secretEnv` A list of secrets stored in Secret Manager referred to in args via a `$$var`

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
package_build <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)
build <- cr_build_make(package_build)
build
cr_buildstep_extract(build, step = 1)
cr_buildstep_extract(build, step = 2)

edit_me <- cr_buildstep_extract(build, step = 2)
cr_buildstep_edit(edit_me, name = "blah")
cr_buildstep_edit(edit_me, name = "gcr.io/blah")
cr_buildstep_edit(edit_me, args = c("blah1", "blah2"), dir = "meh")
```

```
# to edit multiple buildsteps at once
bs <- c(cr_buildstep_extract(build, 1), cr_buildstep_extract(build, 2))
lapply(bs, function(x) cr_buildstep_edit(list(x), dir = "blah")[[1]])
```

cr_buildstep_extract *Extract a buildstep from a Build object*

Description

Useful if you have a step from an existing cloudbuild.yaml you want in another

Usage

```
cr_buildstep_extract(x, step = NULL)
```

Arguments

x	A Build object
step	The numeric step number to extract

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
package_build <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)
build <- cr_build_make(package_build)
build
cr_buildstep_extract(build, step = 1)
cr_buildstep_extract(build, step = 2)
```

`cr_buildstep_gcloud` *A buildstep template for gcloud*

Description

This enables an optimised version of gcloud docker for your buildstep such as `gcr.io/google.com/cloudsdktool/cloud-s`

Usage

```
cr_buildstep_gcloud(component = c("gcloud", "bq", "gsutil", "kubectl"), ...)
```

Arguments

`component`	What gcloud service you need, such as "gcloud", "bq" or "gsutil"
`...`	Arguments passed on to `cr_buildstep`

- `name` name of docker image to call appended to `prefix`
- `args` character vector of arguments
- `prefix` prefixed to name - set to "" to suppress. Will be suppressed if `name` starts with gcr.io or `*-docker.pkg.dev`
- `entrypoint` change the entrypoint for the docker container
- `dir` The directory to use, relative to /workspace e.g. /workspace/deploy/
- `id` Optional id for the step
- `env` Environment variables for this step. A character vector for each assignment
- `volumes` volumes to connect and write to
- `waitFor` Whether to wait for previous buildsteps to complete before running. Default it will wait for previous step.
- `secretEnv` A list of secrets stored in Secret Manager referred to in args via a `$$var`

See Also

https://github.com/GoogleCloudPlatform/cloud-builders/tree/master/gcloud

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

`cr_buildstep_gitsetup` *Create a build step for authenticating with Git*

Description

This creates steps to configure git to use an ssh created key.

This creates steps to use git with an ssh created key.

Usage

```
cr_buildstep_gitsetup(secret, post_setup = NULL)

cr_buildstep_git(
  git_args = c("clone", "git@github.com:[GIT-USERNAME]/[REPOSITORY]", "."),
  ...
)

git_volume()
```

Arguments

`secret`	The name of the secret on Google Secret Manager for the git ssh private key
`post_setup`	Steps that occur after git setup
`git_args`	The arguments to send to git
`...`	Further arguments passed in to cr_buildstep

Details

The ssh private key should be uploaded to Google Secret Manager first

`cr_buildstep` must come after `cr_buildstep_gitsetup`

Use `git_volume` to add the git credentials folder to other buildsteps

See Also

Accessing private GitHub repositories using Cloud Build (google article)

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown(`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")

# assumes you have previously saved git ssh key called "github-ssh"
cr_build_yaml(
  steps = c(
    cr_buildstep_gitsetup("github-ssh"),
    cr_buildstep_git(c(
      "clone",
      "git@github.com:github_name/repo_name"
    ))
  )
)
```

`cr_buildstep_mailgun` *Send an email in a Cloud Build step via MailGun.org*

Description

This uses Mailgun to send emails. It calls an R script that posts the message to MailGuns API.

Usage

```
cr_buildstep_mailgun(
  message,
  to,
  subject,
  from,
  mailgun_url = "$_MAILGUN_URL",
  mailgun_key = "$_MAILGUN_KEY",
  ...
)
```

Arguments

`message`	The message markdown
`to`	to email
`subject`	subject email
`from`	from email
`mailgun_url`	The Mailgun API base URL. Default assumes you set this in Build substitution macros
`mailgun_key`	The Mailgun API key. Default assumes you set this in Build substitution macros
`...`	Other arguments passed to cr_buildstep_r

Details

Requires an account at Mailgun: https://mailgun.com Pre-verification you can only send to a whitelist of emails you configure - see Mailgun website for details.

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
mailgun_url <- "https://api.mailgun.net/v3/sandboxXXX.mailgun.org"
mailgun_key <- "key-XXXX"
## Not run:
# assumes you have verified the email
cr_build(
  cr_build_yaml(
    steps = cr_buildstep_mailgun(
      "Hello from Cloud Build",
      to = "me@verfied_email.com",
      subject = "Hello",
      from = "googleCloudRunner@example.com"
    ),
    substitutions = list(
      `_MAILGUN_URL` = mailgun_url,
      `_MAILGUN_KEY` = mailgun_key
    )
  )
)

## End(Not run)
```

---

cr_buildstep_nginx_setup

Setup nginx for Cloud Run in a buildstep

Description

Setup nginx for Cloud Run in a buildstep

Usage

```
cr_buildstep_nginx_setup(html_folder, ...)
```

Arguments

`html_folder`	The folder that will hold the HTML for Cloud Run This uses a premade bash script that sets up a Docker container ready for Cloud Run running nginx
`...`	Other arguments passed to cr_buildstep_bash

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_packagetests()`, `cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
cr_region_set("europe-west1")

html_folder <- "my_html"
run_image <- "gcr.io/my-project/my-image-for-cloudrun"
cr_build_yaml(
  steps = c(
    cr_buildstep_nginx_setup(html_folder),
    cr_buildstep_docker(run_image, dir = html_folder),
    cr_buildstep_run(
      name = "running-nginx",
      image = run_image,
      concurrency = 80
    )
  )
)
```

`cr_buildstep_packagetests` *Do R package tests and upload to Codecov*

Description

This lets you run R package tests and is intended to be used in a trigger when you push to a repository so you can monitor code quality.

Usage

```
cr_buildstep_packagetests(
  test_script = NULL,
  codecov_script = NULL,
  codecov_token = "$_CODECOV_TOKEN",
  build_image = "gcr.io/gcer-public/packagetools:latest",
  env = c("NOT_CRAN=true")
)
```

Arguments

`test_script`	The script that will call rcmdcheck to perform tests. If `NULL` a default script is used in `system.file("r_buildsteps", "devtools_tests.R", package="googlecloudRunner")`
`codecov_script`	The script that will call codecov to perform coverage. If `NULL` a default script is used in `system.file("r_buildsteps", "codecov_tests.R", package="googleCloudRunner")`
`codecov_token`	If using codecov, supply your codecov token here.
`build_image`	The docker image that will be used to run the R code for the test scripts
`env`	Environment arguments to be set during the test script runs

Details

If using codecov, these environment arguments are added to aid with the codecoverage:

`* "CODECOV_TOKEN=$_CODECOV_TOKEN" * "GCB_PROJECT_ID=$PROJECT_ID" * "GCB_BUILD_ID=$BUILD_ID" * "GCB_COMMIT_SHA=$COMMIT_SHA" * "GCB_REPO_NAME=$REPO_NAME" * "GCB_BRANCH_NAME=$BRANCH_NAME" * "GCB_TAG_NAME=$TAG_NAME" * "GCB_HEAD_BRANCH=$_HEAD_BRANCH" * "GCB_BASE_BRANCH=$_BASE_BRANCH" * "GCB_HEAD_REPO_URL=$_HEAD_REPO_URL" * "GCB_PR_NUMBER=$_PR_NUMBER"`

See Also

`https://docs.codecov.com/reference`

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_buildstep_packagetests()
```

cr_buildstep_pkgdown *Create buildsteps for deploying an R pkgdown website to GitHub*

Description

Create buildsteps for deploying an R pkgdown website to GitHub

Usage

```
cr_buildstep_pkgdown(
  github_repo,
  git_email,
  secret,
  env = NULL,
  build_image = "gcr.io/gcer-public/packagetools:latest",
  post_setup = NULL,
  post_clone = NULL
)
```

Arguments

`github_repo`	The GitHub repo to deploy pkgdown website from and to.
`git_email`	The email the git commands will be identifying as
`secret`	The name of the secret on Google Secret Manager for the git ssh private key
`env`	A character vector of env arguments to set for all steps
`build_image`	A docker image with pkgdown installed
`post_setup`	Steps that occur after git setup
`post_clone`	A cr_buildstep that occurs after the repo is cloned

Details

Its convenient to set some of the above via Build macros, such as `github_repo=$_GITHUB_REPO` and `git_email=$_BUILD_EMAIL` in the Build Trigger web UI

To commit the website to git, cr_buildstep_gitsetup is used for which you will need to add your git ssh private key to Google Secret Manager

The R package is installed via install before running build_site

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")

# set github repo directly to write it out via cr_build_write()
cr_buildstep_pkgdown("MarkEdmondson1234/googleCloudRunner",
  git_email = "cloudbuild@google.com",
  secret = "github-ssh"
)

# github repo set via build trigger macro _GITHUB_REPO
cr_buildstep_pkgdown("$_GITHUB_REPO",
  git_email = "cloudbuild@google.com",
  secret = "github-ssh"
)

# example including environment arguments for pkgdown build step
cr_buildstep_pkgdown("$_GITHUB_REPO",
  git_email = "cloudbuild@google.com",
  secret = "github-ssh",
  env = c("MYVAR=$_MY_VAR", "PROJECT=$PROJECT_ID")
)
```

`cr_buildstep_r` *Run an R script in a Cloud Build R step*

Description

Helper to run R code within build steps, from either an existing local R file or within the source of the build.

Usage

```
cr_buildstep_r(
  r,
  name = "r-base",
  r_source = c("local", "runtime"),
  prefix = "rocker/",
  escape_dollar = TRUE,
  rscript_args = NULL,
  r_cmd = c("Rscript", "R"),
  ...
)
```

Arguments

`r`	R code to run or a file containing R code ending with .R, or the gs:// location on Cloud Storage of the R file you want to run

`name`	The docker image that will run the R code, usually from rocker-project.org
`r_source`	Whether the R code will be from a runtime file within the source or at build time copying over from a local R file in your session
`prefix`	prefixed to name - set to "" to suppress. Will be suppressed if `name` starts with gcr.io or `*-docker.pkg.dev`
`escape_dollar`	Default TRUE. This will turn `$` into `$$` within the script to avoid them being recognised as Cloud Build variables. Turn this off if you want that behaviour (e.g. `my_project="$PROJECT_ID"`)
`rscript_args`	Optional arguments for the R script run by `Rscript`.
`r_cmd`	should ‘Rscript‘ be run or ‘R‘?
`...`	Other arguments passed to cr_buildstep

Details

If `r_source="runtime"` then `r` should be the location of that file within the source or `image` that will be run by the R code from `image`

If `r_source="local"` then it will copy over from a character string or local file into the build step directly.

If the R code location starts with `gs://` then an extra buildstep will be added that will download the R script from that location then run it as per `r_source="runtime"`. This will consequently override your setting of `r_source`

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")

# create an R buildstep inline
cr_buildstep_r(c("paste('1+1=', 1+1)", "sessionInfo()"))
## Not run:

# create an R buildstep from a local file
cr_buildstep_r("my-r-file.R")

# create an R buildstep from a file within the source of the Build
cr_buildstep_r("inst/schedule/schedule.R", r_source = "runtime")

# create an R buildstep with Rscript arguments and use a large
# machine with 32 cores
```

```
## create storage source
storage_source <- cr_build_upload_gcs(
  "my-r-script.R"
)
## create the buildstep with the R script
step1 <- cr_buildstep_r("deploy/my-r-script.R",
  r_source = "runtime",
  rscript_args = c("args_1=<args1>", "args_2=<args_2>")
)

## run the script on Cloud Build
cr_build(
  cr_build_yaml(
    steps = step1
  ),
  source = storage_source,
  options = list(machineType = "E2_HIGHCPU_32")
)

## End(Not run)
# use a different Rocker image e.g. rocker/verse
cr_buildstep_r(c(
  "library(dplyr)",
  "mtcars %>% select(mpg)",
  "sessionInfo()"
),
name = "verse"
)

# use your own R image with custom R
my_r <- c("devtools::install()", "pkgdown::build_site()")
br <- cr_buildstep_r(my_r, name = "gcr.io/gcer-public/packagetools:latest")
```

cr_buildstep_run *Create buildsteps to deploy to Cloud Run*

Description

Create buildsteps to deploy to Cloud Run

Usage

```
cr_buildstep_run(
  name,
  image,
  allowUnauthenticated = TRUE,
  region = cr_region_get(),
  concurrency = 80,
  port = NULL,
  max_instances = "default",
  memory = "256Mi",
  cpu = 1,
  env_vars = NULL,
  gcloud_args = NULL,
  ...
)
```

Arguments

`name`	Name for deployment on Cloud Run
`image`	The name of the image to create or use in deployment - `gcr.io`
`allowUnauthenticated`	TRUE if can be reached from public HTTP address. If FALSE will configure a service-email called `(name)-cloudrun-invoker@(project-id).iam.gserviceaccount.com`
`region`	The endpoint region for deployment
`concurrency`	How many connections each container instance can serve. Can be up to 80.
`port`	Container port to receive requests at. Also sets the $PORT environment variable. Must be a number between 1 and 65535, inclusive. To unset this field, pass the special value "default".
`max_instances`	the desired maximum nuimber of container instances. "default" is 1000, you can get more if you requested a quota instance. For Shiny instances on Cloud Run, this needs to be 1.
`memory`	The format for size is a fixed or floating point number followed by a unit: G, M, or K corresponding to gigabyte, megabyte, or kilobyte, respectively, or use the power-of-two equivalents: Gi, Mi, Ki corresponding to gibibyte, mebibyte or kibibyte respectively. The default is 256Mi
`cpu`	1 or 2 CPUs for your instance
`env_vars`	Environment arguments passed to the Cloud Run container at runtime. Distinct from `env` that run at build time.
`gcloud_args`	a character string of arguments that can be sent to the gcloud command not covered by other parameters of this function
`...`	passed on to cr_buildstep

See Also

Docs for gcloud run deploy this buildstep invokes

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_pkgdown()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

cr_buildstep_secret *Create a buildstep for using Secret Manager*

Description

This is the preferred way to manage secrets for files, rather than cr_buildstep_decrypt, as it stores the encrypted file in the cloud rather than in your project workspace. For single environment values, cr_build_yaml_secrets may be more suitable.

Usage

```
cr_buildstep_secret(
  secret,
  decrypted,
  version = "latest",
  binary_mode = FALSE,
  ...
)
```

Arguments

`secret`	The secret data name in Secret Manager
`decrypted`	The name of the file the secret will be decrypted into
`version`	The version of the secret
`binary_mode`	Should the file be treated in binary/raw format?
`...`	Other arguments sent to cr_buildstep_bash

Details

This is for downloading encrypted files from Google Secret Manager. You will need to add the Secret Accessor Cloud IAM role to the Cloud Build service account to use it. Once you have uploaded your secret file and named it, it is available for Cloud Build to use.

See Also

How to set up secrets using Secret Manager

cr_build_yaml_secrets let you directly support environment arguments in the buildsteps

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_slack()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
cr_buildstep_secret("my_secret", decrypted = "/workspace/secret.json")
```

cr_buildstep_slack — *Send a Slack message to a channel from a Cloud Build step*

Description

This uses https://github.com/technosophos/slack-notify to send Slack messages

Usage

```
cr_buildstep_slack(
  message,
  title = "CloudBuild - $BUILD_ID",
  channel = NULL,
  username = "googleCloudRunnerBot",
  webhook = "$_SLACK_WEBHOOK",
  icon = NULL,
  colour = "#efefef"
)
```

Arguments

`message`	The body of the message
`title`	The title of the message
`channel`	The channel to send the message to (if omitted, use Slack-configured default)
`username`	The name of the sender of the message. Does not need to be a "real" username
`webhook`	The Slack webhook to send to
`icon`	A URL to an icon (squares between 512px and 2000px)
`colour`	The RGB colour for message formatting

Details

You will need to set up a Slack webhook first, via this Slack guide on using incoming webhooks.

Once set, the default is to set this webhook to a Build macro called `_SLACK_WEBHOOK`, or supply it to the webhook argument.

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests`
`cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_targets()`, `cr_buildstep()`

Examples

```
# send a message to googleAuthRverse Slack
webhook <-
  "https://hooks.slack.com/services/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"
cr_buildstep_slack("Hello Slack", webhook = webhook)
## Not run:

bs <- cr_build_yaml(steps = cr_buildstep_slack("Hello Slack"))

cr_build(bs, substitutions = list(`_SLACK_WEBHOOK` = webhook))

## End(Not run)
```

`cr_buildstep_targets` *Buildstep to run a targets pipeline on Cloud Build*

Description

This is a buildstep to help upload a targets pipeline, see cr_build_targets for examples and suggested workflow

Usage

```
cr_buildstep_targets(
  task_args = NULL,
  tar_make = "targets::tar_make()",
  task_image = "gcr.io/gcer-public/targets",
  id = "target pipeline"
)

cr_buildstep_targets_setup(bucket_folder)

cr_buildstep_targets_teardown(bucket_folder, last_id = NULL)
```

Arguments

`task_args`	If not NULL, a named list of additional arguments to send to cr_buildstep_r when its executing the tar_make command (such as environment arguments or waitFor ids)
`tar_make`	The R script that will run in the `tar_make()` step. Modify to include custom settings
`task_image`	An existing Docker image that will be used to run your targets workflow after the targets meta has been downloaded from Google Cloud Storage
`id`	The id of the buildstep. In linkcr_buildstep_targets_multi this is used along with `waitFor` to determine the order of execution

`bucket_folder`	The Google Cloud Storage bucket and folder the target metadata will be saved to, e.g. `gs://my-bucket/my_target_project` You can also pass in build substitution variables such as `"${_MY_BUCKET}"`.
`last_id`	The final buildstep that needs to complete before the upload. If left NULL then will default to the last tar_target step.

See Also

Other Cloud Buildsteps: `cr_buildstep_bash()`, `cr_buildstep_decrypt()`, `cr_buildstep_df()`, `cr_buildstep_docker()`, `cr_buildstep_edit()`, `cr_buildstep_extract()`, `cr_buildstep_gcloud()`, `cr_buildstep_gitsetup()`, `cr_buildstep_mailgun()`, `cr_buildstep_nginx_setup()`, `cr_buildstep_packagetests` `cr_buildstep_pkgdown()`, `cr_buildstep_run()`, `cr_buildstep_r()`, `cr_buildstep_secret()`, `cr_buildstep_slack()`, `cr_buildstep()`

`cr_buildtrigger` *Create a new BuildTrigger*

Description

Build Triggers are a way to have your builds respond to various events, most commonly a git commit or a pubsub event.

Usage

```
cr_buildtrigger(
  build,
  name,
  trigger,
  description = paste("cr_buildtrigger: ", Sys.time()),
  disabled = FALSE,
  substitutions = NULL,
  ignoredFiles = NULL,
  includedFiles = NULL,
  trigger_tags = NULL,
  projectId = cr_project_get(),
  sourceToBuild = NULL,
  overwrite = FALSE
)
```

Arguments

`build`	The build to trigger created via cr_build_make, or the file location of the cloudbuild.yaml within the trigger source
`name`	User assigned name of the trigger
`trigger`	The trigger source created via cr_buildtrigger_repo or a pubsub trigger made with cr_buildtrigger_pubsub or a webhook trigger made with cr_buildtrigger_webhook

`description`	Human-readable description of this trigger
`disabled`	If true, the trigger will never result in a build
`substitutions`	A named list of Build macro variables
`ignoredFiles`	ignored_files and included_files are file glob matches extended with support for "**".
`includedFiles`	If any of the files altered in the commit pass the ignored_files
`trigger_tags`	Tags for the buildtrigger listing
`projectId`	ID of the project for which to configure automatic builds
`sourceToBuild`	A cr_buildtrigger_repo object (but no regex allowed for branch or tag) This field is currently only used by Webhook, Pub/Sub, Manual, and Cron triggers and is the source of the build will execute upon.
`overwrite`	If TRUE will overwrite an existing trigger with the same name

Details

Any source specified in the build will be overwritten to use the trigger as a source (GitHub or Cloud Source Repositories)

If you want multiple triggers for a build, then duplicate the build and create another build under a different name but with a different trigger. Its easier to keep track of.

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
cloudbuild <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)
bb <- cr_build_make(cloudbuild)

# repo hosted on GitHub
gh_trigger <- cr_buildtrigger_repo("MarkEdmondson1234/googleCloudRunner")

# repo mirrored to Cloud Source Repositories
cs_trigger <- cr_buildtrigger_repo("github_markedmondson1234_googlecloudrunner",
  type = "cloud_source"
)
## Not run:
# build with in-line build code
cr_buildtrigger(bb, name = "bt-github-inline", trigger = gh_trigger)

# build with in-line build code using Cloud Source Repository
cr_buildtrigger(bb, name = "bt-github-inline", trigger = cs_trigger)
```

```
# build pointing to cloudbuild.yaml within the GitHub repo
cr_buildtrigger("inst/cloudbuild/cloudbuild.yaml",
  name = "bt-github-file", trigger = gh_trigger
)

# build with repo mirror from file
cr_buildtrigger("inst/cloudbuild/cloudbuild.yaml",
  name = "bt-cs-file", trigger = cs_trigger
)

## End(Not run)

# creating build triggers that respond to pubsub events
## Not run:
# create a pubsub topic either in webUI or via library(googlePubSubR)
library(googlePubsubR)
pubsub_auth()
topics_create("test-topic")

## End(Not run)

# create build trigger that will work from pub/subscription
pubsub_trigger <- cr_buildtrigger_pubsub("test-topic")
pubsub_trigger
## Not run:
# create the build trigger with in-line build
cr_buildtrigger(bb, name = "pubsub-triggered", trigger = pubsub_trigger)
# create scheduler that calls the pub/sub topic

cr_schedule("cloud-build-pubsub",
  "15 5 * * *",
  pubsubTarget = cr_schedule_pubsub("test-topic")
)

## End(Not run)

# create a pubsub trigger that uses github as a source of code to build upon
gh <- cr_buildtrigger_repo("MarkEdmondson1234/googleCloudRunner")
blist <- cr_build_make(cr_build_yaml(cr_buildstep_r('list.files()')))

## Not run:
cr_buildtrigger(blist,
                name = "pubsub-triggered-github-source",
                trigger = pubsub_trigger,
                sourceToBuild = gh)

## End(Not run)
```

cr_buildtrigger_copy *Copy a buildtrigger*

Description

This lets you use the response from cr_buildtrigger_get for an existing buildtrigger to copy over settings to a new buildtrigger.

Usage

```
cr_buildtrigger_copy(
  buildTrigger,
  filename = NULL,
  name = NULL,
  tags = NULL,
  build = NULL,
  ignoredFiles = NULL,
  github = NULL,
  sourceToBuild = NULL,
  substitutions = NULL,
  includedFiles = NULL,
  disabled = NULL,
  triggerTemplate = NULL,
  projectId = cr_project_get()
)
```

Arguments

`buildTrigger`	A `CloudBuildTriggerResponse` object from cr_buildtrigger_get
`filename`	Path, from the source root, to a file whose contents is used for the build
`name`	User assigned name of the trigger
`tags`	Tags for annotation of a ‘BuildTrigger‘
`build`	Contents of the build template
`ignoredFiles`	ignored_files and included_files are file glob matches extended with support for `"**"`.
`github`	a GitHubEventsConfig object - mutually exclusive with `triggerTemplate`
`sourceToBuild`	A cr_buildtrigger_repo object (but no regex allowed for branch or tag) This field is currently only used by Webhook, Pub/Sub, Manual, and Cron triggers and is the source of the build will execute upon.
`substitutions`	A named list of Build macro variables
`includedFiles`	If any of the files altered in the commit pass the ignored_files
`disabled`	If true, the trigger will never result in a build
`triggerTemplate`	a RepoSource object - mutually exclusive with `github`
`projectId`	The projectId you are copying to

Details

Overwrite settings for the build trigger you are copying by supplying it as one of the other arguments from BuildTrigger.

See Also

Other BuildTrigger functions: BuildTrigger(), GitHubEventsConfig(), cr_buildtrigger_delete(), cr_buildtrigger_edit(), cr_buildtrigger_get(), cr_buildtrigger_list(), cr_buildtrigger_pubsub(), cr_buildtrigger_repo(), cr_buildtrigger_run(), cr_buildtrigger_webhook(), cr_buildtrigger()

Examples

```
## Not run:
# copying a GitHub buildtrigger across projects and git repos
bt <- cr_buildtrigger_get("my-trigger", projectId = "my-project-1")

# a new GitHub project
gh <- GitHubEventsConfig("username/new-repo",
  event = "push",
  branch = "^master$"
)

# give 'Cloud Build Editor' role to your service auth key in new project
# then copy configuration across
cr_buildtrigger_copy(bt, github = gh, projectId = "my-new-project")

## End(Not run)
```

cr_buildtrigger_delete

Deletes a ‘BuildTrigger‘ by its project ID and trigger ID.This API is experimental.

Description

Deletes a ‘BuildTrigger‘ by its project ID and trigger ID.This API is experimental.

Usage

```
cr_buildtrigger_delete(triggerId, projectId = cr_project_get())
```

Arguments

triggerId	ID of the ‘BuildTrigger‘ to get or a BuildTriggerResponse object
projectId	ID of the project that owns the trigger

See Also

Other BuildTrigger functions: BuildTrigger(), GitHubEventsConfig(), cr_buildtrigger_copy(), cr_buildtrigger_edit(), cr_buildtrigger_get(), cr_buildtrigger_list(), cr_buildtrigger_pubsub(), cr_buildtrigger_repo(), cr_buildtrigger_run(), cr_buildtrigger_webhook(), cr_buildtrigger()

cr_buildtrigger_edit — *Updates a ‘BuildTrigger‘ by its project ID and trigger ID.This API is experimental.*

Description

Seems not to work at the moment (issue #16)

Usage

```
cr_buildtrigger_edit(BuildTrigger, triggerId, projectId = cr_project_get())
```

Arguments

`BuildTrigger`	The BuildTrigger object to update to
`triggerId`	ID of the ‘BuildTrigger‘ to edit or a previous `BuildTriggerResponse` object that will be edited
`projectId`	ID of the project that owns the trigger

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

Examples

```
## Not run:

github <- GitHubEventsConfig("MarkEdmondson1234/googleCloudRunner",
  branch = "master"
)
bt2 <- cr_buildtrigger("trig2",
  trigger = github,
  build = "inst/cloudbuild/cloudbuild.yaml"
)
bt3 <- BuildTrigger(
  filename = "inst/cloudbuild/cloudbuild.yaml",
  name = "edited1",
  tags = "edit",
  github = github,
  disabled = TRUE,
  description = "edited trigger"
)

edited <- cr_buildtrigger_edit(bt3, triggerId = bt2)

## End(Not run)
```

`cr_buildtrigger_get` *Returns information about a ‘BuildTrigger‘.This API is experimental.*

Description

Returns information about a ‘BuildTrigger‘.This API is experimental.

Usage

```
cr_buildtrigger_get(triggerId, projectId = cr_project_get())
```

Arguments

`triggerId`	ID of the ‘BuildTrigger‘ to get or a `BuildTriggerResponse` object
`projectId`	ID of the project that owns the trigger

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

`cr_buildtrigger_list` *Lists existing ‘BuildTrigger‘s.This API is experimental.*

Description

Lists existing ‘BuildTrigger‘s.This API is experimental.

Usage

```
cr_buildtrigger_list(projectId = cr_project_get())
```

Arguments

`projectId`	ID of the project for which to list BuildTriggers

See Also

cr_build_list which merges with this list

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

Examples

```
## Not run:

cr_buildtrigger_list()

## End(Not run)
```

cr_buildtrigger_pubsub

Create a buildtrigger pub/sub object

Description

Create a trigger from a Pub/Sub topic

Usage

```
cr_buildtrigger_pubsub(
  topic,
  serviceAccountEmail = NULL,
  projectId = cr_project_get()
)
```

Arguments

`topic`	The name of the Cloud Pub/Sub topic or a Topic object from topics_get
`serviceAccountEmail`	Service account that will make the push request.
`projectId`	The GCP project the topic is created within

Details

When using a PubSub trigger, you can use data within your PubSub message in substitution variables within the build. The data from pubsub is available in the variable value: `$(body.message.data.x)` when x is a field in the pubsub message.

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

Examples

```
# create build object
cloudbuild <- system.file("cloudbuild/cloudbuild_substitutions.yml",
  package = "googleCloudRunner"
)
the_build <- cr_build_make(cloudbuild)

# this build includes substitution variables that read from pubsub message var1
the_build

# using googlePubSubR to create pub/sub topic if needed
## Not run:
library(googlePubsubR)
pubsub_auth()
topics_create("test-topic")

## End(Not run)

# create build trigger that will work from pub/subscription
pubsub_trigger <- cr_buildtrigger_pubsub("test-topic")
pubsub_trigger
## Not run:
cr_buildtrigger(the_build, name = "pubsub-triggered-subs", trigger = pubsub_trigger)

## End(Not run)

# make base64 encoded json for pubsub
library(jsonlite)
library(googlePubsubR)

# the message with the var1 that will be passed into the Cloud Build via substitution
message <- toJSON(list(var1 = "hello mum"))

# turning into JSON and encoding
send_me <- msg_encode(message)
## Not run:
# send a PubSub message with the encoded data message
topics_publish(PubsubMessage(send_me), "test-topic")

# did it work? After a while should see logs if it did
cr_buildtrigger_logs("pubsub-triggered-subs")

## End(Not run)
```

cr_buildtrigger_repo *Create a buildtrigger repo object*

Description

Create a repository trigger object for use in build triggers

Usage

```
cr_buildtrigger_repo(
  repo_name,
  branch = ".*",
  tag = NULL,
  type = c("github", "cloud_source"),
  github_secret = NULL,
  ...
)
```

Arguments

`repo_name`	Either the GitHub username/repo_name or the Cloud Source repo_name
`branch`	Regex of the branches that will trigger a build. Ignore if tag is not NULL
`tag`	Regex of tags that will trigger a build
`type`	Whether trigger is GitHub or Cloud Source repoistory
`github_secret`	If you need to pull from a private GitHub repo, add the github secret from Google Secret Manager which will be used via cr_buildstep_secret
`...`	Other arguments passed to either GitHubEventsConfig or RepoSource

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

`cr_buildtrigger_run` *Runs a ‘BuildTrigger‘ at a particular source revision.*

Description

Runs a ‘BuildTrigger‘ at a particular source revision.

Usage

```
cr_buildtrigger_run(triggerId, RepoSource, projectId = cr_project_get())
```

Arguments

`triggerId`	ID of the ‘BuildTrigger‘ to get or a `BuildTriggerResponse` object
`RepoSource`	The RepoSource object to pass to this method
`projectId`	ID of the project

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

`cr_buildtrigger_webhook` *Create a buildtrigger webhook object*

Description

Create a trigger from a webhook

Usage

```
cr_buildtrigger_webhook(secret)
```

Arguments

`secret`	Resource name for the secret required as a URL parameter.

See Also

Other BuildTrigger functions: `BuildTrigger()`, `GitHubEventsConfig()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger()`

`cr_build_artifacts` *Download artifacts from a build*

Description

If a completed build includes artifact files this downloads them to local files

Usage

```
cr_build_artifacts(
  build,
  download_folder = getwd(),
  overwrite = FALSE,
  path_regex = NULL
)
```

Arguments

`build`	A Build object that includes the artifact location
`download_folder`	Where to download the artifact files
`overwrite`	Whether to overwrite existing local data
`path_regex`	A regex of files to fetch from the artifact bucket location. This is due to not being able to support the path globs

Details

If your artifacts are using file glob (e.g. `myfolder/**`) to decide which workspace files are uploaded to Cloud Storage, you will need to create a path_regex of similar functionality (`"^myfolder/"`). This is not needed if you use absolute path names such as `"myfile.csv"`

See Also

Storing images and artifacts

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
## Not run:
#' r <- "write.csv(mtcars,file = 'artifact.csv')"
ba <- cr_build_yaml(
  steps = cr_buildstep_r(r),
  artifacts = cr_build_yaml_artifact("artifact.csv", bucket = "my-bucket")
)
ba

build <- cr_build(ba)
built <- cr_build_wait(build)

cr_build_artifacts(built)

## End(Not run)
```

`cr_build_list` *Lists the build*

Description

Get a list of builds within your project

cr_build_list_filter outputs valid filters for `cr_build_list`'s filter argument

Usage

```
cr_build_list(
  filter = NULL,
  projectId = cr_project_get(),
  pageSize = 1000,
  data_frame_output = TRUE
)

cr_build_list_filter(
  field,
  operator = c("=", "!=", ">", ">=", "<", "<="),
  value
)
```

Arguments

`filter`	Text filter for the list - use `cr_build_list_filter()` or a raw string
`projectId`	ID of the project
`pageSize`	How many builds to fetch per page
`data_frame_output`	If TRUE will output a data.frame of a subset of info from the builds, merged with the list of triggers from cr_buildtrigger_list. Set to FALSE to return a list of Build objects similar to output from cr_build_status
`field`	The field you want to filter on. Will validate.
`operator`	The type of comparision for the filter
`value`	The value for the filter's field. Auto-formats `POSIXct` and `Date` objects

Details

When `data_frame_output=TRUE` results are sorted with the latest buildStartTime in the first row

If filter is `NULL` then this will return all historic builds. To use filters, ensure you use `""` and not `''` to quote the fields e.g. `'status!="SUCCESS"'` and `'status="SUCCESS"'` - see Filtering build results docs. `cr_build_list_filter` helps you construct valid filters. More complex filters can be done using a combination of paste and `cr_build_list_filter()` - see examples

Use POSIXct via functions like Sys.time to have them formatted into valid timestamps for time related fields, or Date objects via functions like Sys.Date

See Also

`https://cloud.google.com/build/docs/api/reference/rest/v1/projects.builds/list`

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
## Not run:

# merge with buildtrigger list
cr_build_list()

# output a list of build objects
cr_build_list(data_frame_output = FALSE)

# output a list of builds that failed using raw string
cr_build_list('status!="SUCCESS"')

# output builds for a specific trigger using raw string
cr_build_list('trigger_id="af2c7ddc-e4eb-4170-b938-a4babb53bac6"')

# use cr_build_list_filter to help validate filters
failed_builds <- cr_build_list_filter("status", "!=", "SUCCESS")
cr_build_list(failed_builds)

f1 <- cr_build_list_filter(
  "trigger_id", "=", "af2c7ddc-e4eb-4170-b938-a4babb53bac6"
)
cr_build_list(f1)

# do AND (and other) filters via paste() and cr_build_list_filter()
cr_build_list(paste(f1, "AND", failed_builds))

# builds in last 5 days
last_five <- cr_build_list_filter("create_time", ">", Sys.Date() - 5)
cr_build_list(last_five)

# builds in last 60 mins
last_hour <- cr_build_list_filter("create_time", ">", Sys.time() - 3600)
cr_build_list(last_hour)

# builds for this package's buildtrigger
gcr_trigger_id <- "0a3cade0-425f-4adc-b86b-14cde51af674"
gcr_bt <- cr_build_list_filter(
  "trigger_id",
  value = gcr_trigger_id
)
gcr_builds <- cr_build_list(gcr_bt)

# get logs for last build
last_build <- gcr_builds[1, ]
last_build_logs <- cr_build_logs(log_url = last_build$bucketLogUrl)
tail(last_build_logs, 10)

## End(Not run)
```

cr_build_logs *Download logs from a Cloud Build*

Description

This lets you download the logs to your local R session, rather than viewing them in the Cloud Console.

Usage

```
cr_build_logs(built = NULL, log_url = NULL)

cr_buildtrigger_logs(
  trigger_name = NULL,
  trigger_id = NULL,
  projectId = cr_project_get()
)
```

Arguments

`built`	The built object from cr_build_status or cr_build_wait
`log_url`	You can optionally instead of `built` provide the direct gs:// URI to the log here. It is in the format `gs://{{bucket}}/log-{{buildId}}.txt`
`trigger_name`	The trigger name to check, will be used to look up trigger_id
`trigger_id`	If supplied, trigger_name will be ignored
`projectId`	The project containing the trigger_id

Details

By default, Cloud Build stores your build logs in a Google-created Cloud Storage bucket. You can view build logs store in the Google-created Cloud Storage bucket, but you cannot make any other changes to it. If you require full control over your logs bucket, store the logs in a user-created Cloud Storage bucket.

See Also

`https://cloud.google.com/build/docs/securing-builds/store-manage-build-logs`

cr_build_logs_badger to see logs for a badger created build

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
## Not run:
s_yaml <- cr_build_yaml(steps = cr_buildstep("gcloud", "version"))
build <- cr_build_make(s_yaml)
built <- cr_build(build)
the_build <- cr_build_wait(built)
cr_build_logs(the_build)
# [1] "starting build \"6ce86e05-b0b1-4070-a849-05ec9020fd3b\""
# [2] ""
# [3] "FETCHSOURCE"
# [4] "BUILD"
# [5] "Already have image (with digest): gcr.io/cloud-builders/gcloud"
# [6] "Google Cloud SDK 325.0.0"
# [7] "alpha 2021.01.22"
# [8] "app-engine-go 1.9.71"
# ...

## End(Not run)
## Not run:

# get your trigger name
ts <- cr_buildtrigger_list()
ts$buildTriggerName

my_trigger <- "package-checks"
last_logs <- cr_buildtrigger_logs(my_trigger)

my_trigger_id <- "0a3cade0-425f-4adc-b86b-14cde51af674"
last_logs <- cr_buildtrigger_logs(trigger_id = my_trigger_id)

## End(Not run)
```

cr_build_make *Make a Cloud Build object out of a cloudbuild.yml file*

Description

This creates a Build object via the standard cloudbuild.yaml format

Usage

```
cr_build_make(
  yaml,
  source = NULL,
  timeout = NULL,
  images = NULL,
  artifacts = NULL,
  options = NULL,
  substitutions = NULL,
  availableSecrets = NULL,
  serviceAccount = NULL,
  logsBucket = NULL
)
```

Arguments

`yaml`	A `Yaml` object created from cr_build_yaml or a file location of a .yaml/.yml cloud build file
`source`	A Source object specifying the location of the source files to build, usually created by cr_build_source
`timeout`	Amount of time that this build should be allowed to run, to second
`images`	A list of images to be pushed upon the successful completion of all build
`artifacts`	Artifacts that may be built via cr_build_yaml_artifact
`options`	Options to pass to a Cloud Build
`substitutions`	Substitutions data for ‘Build‘ resource
`availableSecrets`	Secret Manager objects built by cr_build_yaml_secrets
`serviceAccount`	service account email to be used for the build
`logsBucket`	The gs:// location of a bucket to put logs in

See Also

`https://cloud.google.com/build/docs/build-config-file-schema`

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
cloudbuild <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)
cr_build_make(cloudbuild)
```

`cr_build_schedule_http` *Create a Cloud Scheduler HTTP target from a Cloud Build object*

Description

This enables Cloud Scheduler to trigger Cloud Builds

Usage

```
cr_build_schedule_http(
  build,
  email = cr_email_get(),
  projectId = cr_project_get()
)

cr_schedule_http(build, email = cr_email_get(), projectId = cr_project_get())

cr_schedule_pubsub(
  topicName,
  PubsubMessage = NULL,
  data = NULL,
  attributes = NULL,
  projectId = cr_project_get()
)

cr_schedule(
  name,
  schedule = NULL,
  httpTarget = NULL,
  pubsubTarget = NULL,
  description = NULL,
  overwrite = FALSE,
  timeZone = Sys.timezone(),
  region = cr_region_get(),
  projectId = cr_project_get()
)
```

Arguments

`build`	A Build object created via cr_build_make or cr_build
`email`	The email that will authenticate the job set via cr_email_set
`projectId`	The GCP project to run within usually set with cr_project_set
`topicName`	The name of the Cloud Pub/Sub topic or a Topic object from topics_get
`PubsubMessage`	A `PubsubMessage` object generated via PubsubMessage. If used, then do not send in ‘data‘ or ‘attributes‘ arguments as will be redundant since this variable will hold the information.
`data`	The message payload for PubsubMessage. An R object that will be turned into JSON via [jsonlite] and then base64 encoded into the PubSub format.
`attributes`	Attributes for PubsubMessage.
`name`	Name to call your scheduled job
`schedule`	A cron schedule e.g. `"15 5 * * *"`
`httpTarget`	A HTTP target object HttpTarget
`pubsubTarget`	A Pub/Sub target object PubsubTarget such as created via cr_schedule_pubsub

`description`	Optionally caller-specified in CreateJob or
`overwrite`	If TRUE and an existing job with the same name exists, will overwrite it with the new parameters
`timeZone`	Specifies the time zone to be used in interpreting schedule. If set to `NULL` will be "UTC". Note that some time zones include a provision for daylight savings time.
`region`	The region usually set with cr_region_set

Details

Ensure you have a service email with cr_email_set of format `service-{project-number}@gcp-sa-cloudscheduler.iam.` with Cloud Scheduler Service Agent role as per https://cloud.google.com/scheduler/docs/http-target-auth#add

You can parametrise builds by sending in values within PubSub. To read the data in the message set a substitution variable that picks up the data. For example `_VAR1=$(body.message.data.var1)`

If your schedule to PubSub fails with a permission error, try turning the Cloud Scheduler API off and on again the Cloud Console, which will refresh the Google permissions.

Value

`cr_schedule_http` returns a HttpTarget object for use in cr_schedule

`cr_schedule_pubsub` returns a PubsubTarget object for use within cr_schedule or cr_schedule_build

A `gar_scheduleJob` class object

See Also

https://cloud.google.com/build/docs/api/reference/rest/v1/projects.builds/create

Google Documentation for Cloud Scheduler

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

Examples

```
cloudbuild <- system.file("cloudbuild/cloudbuild.yaml", package = "googleCloudRunner")
build1 <- cr_build_make(cloudbuild)
build1
## Not run:
cr_schedule("cloud-build-test1",
  schedule = "15 5 * * *",
  httpTarget = cr_schedule_http(build1)
)

# a cloud build you would like to schedule
itworks <- cr_build("cloudbuild.yaml", launch_browser = FALSE)
```

```
# once working, pass in the build to the scheduler
cr_schedule("itworks-schedule",
  schedule = "15 5 * * *",
  httpTarget = cr_schedule_http(itworks)
)

## End(Not run)
cr_project_set("my-project")
cr_bucket_set("my-bucket")
cloudbuild <- system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
)
bb <- cr_build_make(cloudbuild)
## Not run:
# create a pubsub topic either in Google Console webUI or library(googlePubSubR)
library(googlePubsubR)
pubsub_auth()
topics_create("test-topic")

## End(Not run)

# create build trigger that will watch for messages to your created topic
pubsub_trigger <- cr_buildtrigger_pubsub("test-topic")
pubsub_trigger
## Not run:
# create the build trigger with in-line build
cr_buildtrigger(bb, name = "pubsub-triggered", trigger = pubsub_trigger)


# create scheduler that calls the pub/sub topic
cr_schedule("cloud-build-pubsub",
  "15 5 * * *",
  pubsubTarget = cr_schedule_pubsub("test-topic")
)

## End(Not run)

# builds can be also parametrised to respond to parameters within your pubsub topic
# this cloudbuild echos back the value sent in 'var1'
cloudbuild <- system.file("cloudbuild/cloudbuild_substitutions.yml",
  package = "googleCloudRunner"
)
the_build <- cr_build_make(cloudbuild)

# var1 is sent via Pubsub to the buildtrigger
message <- list(var1 = "hello mum")
send_me <- jsonlite::base64_enc(jsonlite::toJSON(message))

# create build trigger that will work from pub/subscription
pubsub_trigger <- cr_buildtrigger_pubsub("test-topic")
## Not run:
cr_buildtrigger(the_build, name = "pubsub-triggered-subs", trigger = pubsub_trigger)
```

```
# create scheduler that calls the pub/sub topic with a parameter
cr_schedule("cloud-build-pubsub",
  "15 5 * * *",
  pubsubTarget = cr_schedule_pubsub("test-topic",
    data = send_me
  )
)

## End(Not run)

## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule("test",
      "* * * * *",
      httpTarget = HttpTarget(uri="https://code.markedmondson.me"))

# schedule a cloud build (no source)
build1 <- cr_build_make("cloudbuild.yaml")
cr_schedule("cloud-build-test", "15 5 * * *",
             httpTarget = cr_schedule_http(build1))

# schedule a cloud build with code source from GCS bucket
my_gcs_source <- cr_build_upload_gcs("my_folder", bucket = cr_get_bucket())
build <- cr_build_make("cloudbuild.yaml", source = my_gcs_source)
cr_schedule("cloud-build-test2", "15 5 * * *",
            httpTarget = cr_schedule_http(build))

# update a schedule with the same name - only supply what you want to change
cr_schedule("cloud-build-test2", "12 6 * * *", overwrite=TRUE)

# By default will use the timezone as specified by Sys.timezone() - change
# this by supplying it directly
cr_schedule("timzone-utc", "12 2 * * *", timeZone = "UTC")

# schedule private Cloud Run app
# for authenticated Cloud Run apps - create with allowUnauthenticated=FALSE
cr_deploy_run("my-app", allowUnauthenticated = TRUE)

# deploying via R will help create a service email called my-app-invoker
cr_run_email("my-app")
#> "my-app-invoker@your-project.iam.gserviceaccount.com"

# schedule the endpoint
my_app <- cr_run_get("my-app")

endpoint <- paste0(my_app$status$url, "/fetch_stuff")

app_sched <- cr_run_schedule_http(endpoint, http_method = "GET",
                                  email = cr_run_email("my-app"))

cr_schedule("my-app-scheduled-1", schedule = "16 4 * * *",
            httpTarget = app_sched)
```

```
# creating build triggers that respond to pubsub events

\dontrun{
# create a pubsub topic either in webUI or via library(googlePubSubR)
library(googlePubsubR)
pubsub_auth()
topics_create("test-topic")
}

# create build trigger that will work from pub/subscription
pubsub_trigger <- cr_buildtrigger_pubsub("test-topic")
pubsub_trigger

\dontrun{
# create the build trigger with in-line build
cr_buildtrigger(bb, name = "pubsub-triggered", trigger = pubsub_trigger)
# create scheduler that calls the pub/sub topic

cr_schedule("cloud-build-pubsub",
            "15 5 * * *",
            pubsubTarget = cr_schedule_pubsub("test-topic"))

}


## End(Not run)
```

cr_build_source *Build a source object*

Description

This creates a source object for a build. Note you may instead want to use sources connected to a Build Trigger in which case see cr_buildtrigger_repo

Usage

```
cr_build_source(x)

## S3 method for class 'gar_RepoSource'
cr_build_source(x)

## S3 method for class 'gar_StorageSource'
cr_build_source(x)
```

Arguments

x	A RepoSource or a StorageSource object

Examples

```
repo <- RepoSource("my_repo", branchName = "master")
gcs <- StorageSource("my_code.tar.gz", "gs://my-bucket")

cr_build_source(repo)
cr_build_source(gcs)

my_gcs_source <- cr_build_source(gcs)
my_repo_source <- cr_build_source(repo)
## Not run:

build1 <- cr_build("cloudbuild.yaml", source = my_gcs_source)
build2 <- cr_build("cloudbuild.yaml", source = my_repo_source)

## End(Not run)
```

`cr_build_status` *Returns information about a previously requested build.*

Description

The ‘Build‘ that is returned includes its status (such as ‘SUCCESS‘,‘FAILURE‘, or ‘WORKING‘), and timing information.

Usage

```
cr_build_status(id = .Last.value, projectId = cr_project_get())
```

Arguments

`id`	ID of the build or a `BuildOperationMetadata` object
`projectId`	ID of the project

Value

A gar_Build object Build or NULL if not found

See Also

`https://cloud.google.com/build/docs/api/reference/rest/v1/projects.builds#Build.Status`

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

cr_build_targets *Set up Google Cloud Build to run a targets pipeline*

Description

Creates a Google Cloud Build yaml file so as to execute tar_make pipelines

Historical runs accumulate in the configured Google Cloud Storage bucket, and the latest output is downloaded before tar_make executes so up-to-date steps do not rerun.

Usage

```
cr_build_targets(
  buildsteps = cr_buildstep_targets_multi(),
  execute = c("trigger", "now"),
  path = "cloudbuild_targets.yaml",
  local = ".",
  predefinedAcl = "bucketLevel",
  bucket = cr_bucket_get(),
  download_folder = getwd(),
  ...
)

cr_build_targets_artifacts(
  build,
  bucket = cr_bucket_get(),
  target_folder = NULL,
  download_folder = NULL,
  target_subfolder = c("all", "meta", "objects", "user"),
  overwrite = TRUE
)

cr_buildstep_targets_single(
  target_folder = NULL,
  bucket = cr_bucket_get(),
  tar_config = NULL,
  task_image = "gcr.io/gcer-public/targets",
  task_args = NULL,
  tar_make = "targets::tar_make()"
)

cr_buildstep_targets_multi(
  target_folder = NULL,
  bucket = cr_bucket_get(),
  tar_config = NULL,
  task_image = "gcr.io/gcer-public/targets",
  task_args = NULL,
  last_id = NULL
)
```

Arguments

`buildsteps`	Generated buildsteps that create the targets build
`execute`	Whether to run the Cloud Build now or to write to a file for use within triggers or otherwise
`path`	File path to write the Google Cloud Build yaml workflow file. Set to NULL to write no file and just return the `Yaml` object.
`local`	If executing now, the local folder that will be uploaded as the context for the target build
`predefinedAcl`	The ACL rules for the object uploaded. Set to "bucketLevel" for buckets with bucket level access enabled
`bucket`	The Google Cloud Storage bucket the target metadata will be saved to in folder ‘target_folder‘
`download_folder`	Set to NULL to overwrite local _target folder: `_targets/*` otherwise will write to `download_folder/_targets/*`
`...`	Arguments passed on to `cr_build_yaml`, `cr_build_yaml` `steps` A vector of cr_buildstep `timeout` How long the entire build will run. If not set will be 10mins `logsBucket` Where logs are written. If you don’t set this field, Cloud Build will use a default bucket to store your build logs. `options` A named list of options `substitutions` Build macros that will replace entries in other elements `tags` Tags for the build `secrets` A secrets object `images` What images will be build from this cloudbuild `artifacts` What artifacts may be built from this cloudbuild - create via cr_build_yaml_artifact `availableSecrets` What environment arguments from Secret Manager are available to the build - create via cr_build_yaml_secrets `serviceAccount` What service account should the build be run under?
`build`	A Build object that includes the artifact location
`target_folder`	Where target metadata will sit within the Google Cloud Storage bucket as a folder. If NULL defaults to RStudio project name or "targets_cloudbuild" if no RStudio project found.
`target_subfolder`	If you only want to download a specific folder from the _targets/ folder on Cloud Build then specify it here.
`overwrite`	Whether to overwrite existing local data
`tar_config`	An R script that will run before `targets::tar_make()` in the build e.g. `"targets::tar_config_set(sc = 'targets/_targets.R')"`

`task_image`	An existing Docker image that will be used to run your targets workflow after the targets meta has been downloaded from Google Cloud Storage
`task_args`	A named list of additional arguments to send to cr_buildstep_r when its executing the tar_make command (such as environment arguments)
`tar_make`	The R script that will run in the tar_make() step. Modify to include custom settings such as "script"
`last_id`	The final buildstep that needs to complete before the upload. If left NULL then will default to the last tar_target step.

Details

Steps to set up your target task in Cloud Build:

- Create your ‘targets‘ workflow.
- Create a Dockerfile that holds the R and system dependencies for your workflow. You can test the image using cr_deploy_docker. Include `library(targets)` dependencies - a Docker image with `targets` installed is available at `gcr.io/gcer-public/targets`.
- Run `cr_build_targets` to create the cloudbuild yaml file.
- Run the build via cr_build or similar. Each build should only recompute outdated targets.
- Optionally create a build trigger via cr_buildtrigger.
- Trigger a build. The first trigger will run the targets pipeline, subsequent runs will only recompute the outdated targets.

Use `cr_build_targets_artifacts` to download the return values of a target Cloud Build, then tar_read to read the results. You can set the downloaded files as the target store via `targets::tar_config_set(store="_tar` Set `download_folder = "_targets"` to overwrite your local targets store.

Value

A Yaml object as generated by cr_build_yaml if `execute="trigger"` or the built object if `execute="now"`

`cr_build_targets_artifacts` returns the file path to where the download occurred.

DAGs

If your target workflow has parallel processing steps then leaving this as default `cr_buildstep_targets_multi()` will create a build that uses waitFor and build ids to create a DAG. Setting this to `cr_buildstep_targets_single()` will be single thread but you can then customise the `targets::tar_make` script. Or add your own custom target buildsteps here using cr_buildstep_targets - for example you could create the docker environment targets runs within before the main pipeline.

See Also

cr_buildstep_targets if you want to customise the build

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
write.csv(mtcars, file = "mtcars.csv", row.names = FALSE)

targets::tar_script(
  list(
    targets::tar_target(file1,
      "mtcars.csv", format = "file"),
    targets::tar_target(input1,
      read.csv(file1)),
    targets::tar_target(result1,
      sum(input1$mpg)),
    targets::tar_target(result2,
      mean(input1$mpg)),
    targets::tar_target(result3,
      max(input1$mpg)),
    targets::tar_target(result4,
      min(input1$mpg)),
    targets::tar_target(merge1,
      paste(result1, result2, result3, result4))
    ),
 ask = FALSE)

bs <- cr_buildstep_targets_multi()

# only create the yaml
par_build <- cr_build_targets(bs, path = NULL)
par_build

# clean up example
unlink("mtcars.csv")
unlink("_targets.R")

## Not run:
# run it immediately in cloud
cr_build_targets(bs, execute="now")

# create a yaml file for use in build triggers
cr_build_targets(bs)

## End(Not run)
```

`cr_build_upload_gcs` *Create a StorageSource*

Description

This creates a StorageSource object after uploading to Google Cloud Storage

Usage

```
cr_build_upload_gcs(
  local,
  remote = paste0(local, format(Sys.time(), "%Y%m%d%H%M%S"), ".tar.gz"),
  bucket = cr_bucket_get(),
  predefinedAcl = "bucketOwnerFullControl",
  deploy_folder = "deploy"
)

cr_buildstep_source_move(deploy_folder)
```

Arguments

`local`	Local directory containing the Dockerfile etc. you wish to deploy
`remote`	The name of the folder in your bucket
`bucket`	The Google Cloud Storage bucket to upload to
`predefinedAcl`	The ACL rules for the object uploaded. Set to "bucketLevel" for buckets with bucket level access enabled
`deploy_folder`	Which folder to deploy from - this will mean the files uploaded will be by default in `/workspace/deploy/`

Details

`cr_build_upload_gcs` copies the files into the `deploy_folder` in your working directory, then tars it for upload. Files will be available on Cloud Build at `/workspace/deploy_folder/*`.

`cr_buildstep_source_move` is a way to move the StorageSource files in `/workspace/deploy_folder/*` into the root `/workspace/*` location, which is more consistent with RepoSource objects or GitHub build triggers created using cr_buildtrigger_repo. This means the same runtime code can run for both sources.

Value

A Source object

See Also

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_bucket_set("my-bucket")
my_gcs_source <- cr_build_upload_gcs("my_folder")
build1 <- cr_build("cloudbuild.yaml", source = my_gcs_source)
```

```
## End(Not run)
cr_buildstep_source_move("deploy")
```

cr_build_wait *Wait for a Build to run*

Description

This will repeatedly call cr_build_status whilst the status is STATUS_UNKNOWN, QUEUED or WORKING

Usage

```
cr_build_wait(op = .Last.value, projectId = cr_project_get())
```

Arguments

`op`	The operation build object to wait for
`projectId`	The projectId

Value

A gar_Build object Build

See Also

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

cr_build_write *Write out a Build object to cloudbuild.yaml*

Description

Write out a Build object to cloudbuild.yaml

Usage

```
cr_build_write(x, file = "cloudbuild.yaml")
```

Arguments

x	A Build object perhaps created with cr_build_make or cr_build_yaml
file	Where to write the yaml file

See Also

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
cr_project_set("my-project")
# write from creating a Yaml object
image <- "gcr.io/my-project/my-image$BUILD_ID"
run_yaml <- cr_build_yaml(
  steps = c(
    cr_buildstep("docker", c("build", "-t", image, ".")),
    cr_buildstep("docker", c("push", image)),
    cr_buildstep("gcloud", c("beta", "run", "deploy", "test1", "--image", image))
  ),
  images = image
)
## Not run:
cr_build_write(run_yaml)

## End(Not run)

# write from a Build object
build <- cr_build_make(system.file("cloudbuild/cloudbuild.yaml",
  package = "googleCloudRunner"
))
## Not run:
cr_build_write(build)

## End(Not run)
```

`cr_build_yaml` *Create a cloudbuild Yaml object in R*

Description

This can be written to disk or used directly with functions such as cr_build

Usage

```
cr_build_yaml(
  steps,
  timeout = NULL,
  logsBucket = NULL,
  options = NULL,
  substitutions = NULL,
  tags = NULL,
  secrets = NULL,
  availableSecrets = NULL,
  images = NULL,
  artifacts = NULL,
  serviceAccount = NULL
)
```

Arguments

`steps`	A vector of cr_buildstep
`timeout`	How long the entire build will run. If not set will be 10mins
`logsBucket`	Where logs are written. If you don't set this field, Cloud Build will use a default bucket to store your build logs.
`options`	A named list of options
`substitutions`	Build macros that will replace entries in other elements
`tags`	Tags for the build
`secrets`	A secrets object
`availableSecrets`	What environment arguments from Secret Manager are available to the build - create via cr_build_yaml_secrets
`images`	What images will be build from this cloudbuild
`artifacts`	What artifacts may be built from this cloudbuild - create via cr_build_yaml_artifact
`serviceAccount`	What service account should the build be run under?

See Also

Build configuration overview for cloudbuild.yaml

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build()`

Examples

```
cr_project_set("my-project")
image <- "gcr.io/my-project/my-image"
cr_build_yaml(
  steps = c(
    cr_buildstep("docker", c("build", "-t", image, ".")),
    cr_buildstep("docker", c("push", image)),
    cr_buildstep("gcloud", c("beta", "run", "deploy", "test1", "--image", image))
  ),
  images = image
)
```

`cr_build_yaml_artifact`

Add an artifact for cloudbuild.yaml

Description

Add artifact objects to a build

Usage

```
cr_build_yaml_artifact(paths, bucket_dir = NULL, bucket = cr_bucket_get())
```

Arguments

`paths`	Which files from the working directory to upload to cloud storage once the build is finished. Can use globs but see details of cr_build_artifacts on how that affects downloads
`bucket_dir`	The directory in the bucket the files will be uploaded to
`bucket`	the bucket to send to

See Also

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
## Not run:
cr_project_set("my-project")
r <- "write.csv(mtcars,file = 'artifact.csv')"
cr_build_yaml(
  steps = cr_buildstep_r(r),
  artifacts = cr_build_yaml_artifact("artifact.csv", bucket = "my-bucket")
)

## End(Not run)
```

cr_build_yaml_secrets *Create an availableSecrets entry for build yaml*

Description

This creates the availabelSecrets entry for Builds so they can use Secret Manager environment arguments in the builds.

Usage

```
cr_build_yaml_secrets(
  secretEnv,
  secret,
  version = "latest",
  projectId = cr_project_get()
)
```

Arguments

`secretEnv`	The name of the secretEnv that will be referred to in the build steps e.g. `'GH_TOKEN'`
`secret`	The secret data name in Secret Manager
`version`	The version of the secret
`projectId`	The project to get the Secret from

See Also

To download from Secret Manager to a file in a dedicated buildstep see cr_buildstep_secret.

Using secrets from Secret Manager

Other Cloud Build functions: `Build()`, `RepoSource()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml()`, `cr_build()`

Examples

```
cr_build_yaml_secrets("GH_TOKEN", "github_token")

s1 <- cr_build_yaml_secrets("USERNAME", "my_username")
s2 <- cr_build_yaml_secrets("PASSWORD", "my_password")

# use one $ in scripts to use the secretEnv (will be replaced by $$)
cr_build_yaml(
  steps = cr_buildstep(
    "docker",
    entrypoint = "bash",
    args = c(
      "-c",
      "docker login --username=$USERNAME --password=$PASSWORD"
    ),
    secretEnv = c("USERNAME", "PASSWORD")
  ),
  availableSecrets = list(s1, s2)
)
```

`cr_deploy_badger` *Deploy a Cloud Run app to display build badges*

Description

This uses https://github.com/kelseyhightower/badger to create badges you can display in README.md etc. showing the current status of a Cloud Build

Usage

```
cr_deploy_badger(
  badger_image = "gcr.io/hightowerlabs/badger:0.0.1",
  json = Sys.getenv("GAR_CLIENT_JSON"),
  region = cr_region_get()
)

cr_build_logs_badger(dir = getwd(), projectId = cr_project_get())
```

Arguments

`badger_image`	The docker image from the badger project to use
`json`	The clientId JSON file of the project to create within
`region`	The Cloud Run region
`dir`	The directory containing the README.md file
`projectId`	The projectId running the badger badge for a buildtrigger

Details

`cr_build_logs_badger` is intended to be run from the root directory of an R package that holds a README.md file containing a `![Cloudbuild]` badge as created by `cr_deploy_badger()`. The function will scan the README.md file for the correct triggerId to pass to cr_buildtrigger_logs

cr_deploy_docker *Deploy a local Dockerfile to be built on ContainerRegistry*

Description

Build a local Dockerfile in the cloud. See googleCloudRunner website for help how to generate Dockerfiles. If you want the docker to build on each commit, see also cr_deploy_docker_trigger

Usage

```
cr_deploy_docker(
  local,
  image_name = remote,
  dockerfile = NULL,
  remote = basename(local),
  tag = c("latest", "$BUILD_ID"),
  timeout = 600L,
  bucket = cr_bucket_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  kaniko_cache = TRUE,
  predefinedAcl = "bucketOwnerFullControl",
  pre_steps = NULL,
  post_steps = NULL,
  ...
)

cr_deploy_docker_construct(
  local,
  image_name = remote,
  dockerfile = NULL,
  remote = basename(local),
  tag = c("latest", "$BUILD_ID"),
  timeout = 600L,
  bucket = cr_bucket_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  kaniko_cache = TRUE,
  predefinedAcl = "bucketOwnerFullControl",
  pre_steps = NULL,
  post_steps = NULL,
  ...
)
```

Arguments

`local`	The folder containing the Dockerfile to build

`image_name`	The name of the docker image to be built either full name starting with gcr.io or constructed from the image_name and projectId via `gcr.io/{projectId}/{image_name}`
`dockerfile`	An optional Dockerfile built to support the script. Not needed if "Dockerfile" exists in folder. If supplied will be copied into deployment folder and called "Dockerfile"
`remote`	The folder on Google Cloud Storage
`tag`	The tag or tags to be attached to the pushed image - can use `Build` macros
`timeout`	Amount of time that this build should be allowed to run, to second
`bucket`	The GCS bucket that will be used to deploy code source
`projectId`	The projectId
`launch_browser`	Whether to launch the logs URL in a browser once deployed
`kaniko_cache`	If TRUE will use kaniko cache for Docker builds.
`predefinedAcl`	Access setting for the bucket used in deployed. Set to "bucketLevel" if using bucket level access
`pre_steps`	Other cr_buildstep to run before the docker build
`post_steps`	Other cr_buildstep to run after the docker build
`...`	Arguments passed on to `cr_buildstep_docker`

- `image` The image tag that will be pushed, starting with gcr.io or created by combining with `projectId` if not starting with gcr.io
- `location` Where the Dockerfile to build is in relation to `dir`
- `build_args` additional arguments to pass to `docker build`, should be a character vector.
- `push_image` if `kaniko_cache = FALSE` and `push_image = FALSE`, then the docker image is simply built and not pushed

Details

This lets you deploy local folders with Dockerfiles, automating saving the source on Google Cloud Storage.

To deploy builds on git triggers and sources such as GitHub, see the examples of cr_buildstep_docker or the use cases on the website

Note

'cr_deploy_docker_construct' is a helper function to construct the arguments needed to deploy the docker, which may be combined with `cr_deploy_r` to combine Docker and R

See Also

If you want the docker to build on each commit, see cr_deploy_docker_trigger

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_packagetests()`, `cr_deploy_pkgdown()`, `cr_deploy_run_website()`, `cr_deploy_run()`, `cr_deploy_r()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_email_set("123456@projectid.iam.gserviceaccount.com")
cr_bucket_set("my-bucket")

b <- cr_deploy_docker(system.file("example/", package = "googleCloudRunner"))

## End(Not run)
```

`cr_deploy_docker_trigger`

Deploy Docker build from a Git repo

Description

This helps the common use case of building a Dockerfile based on the contents of a GitHub repo, and sets up a build trigger so it will build on every commit.

Usage

```
cr_deploy_docker_trigger(
  repo,
  image,
  trigger_name = paste0("docker-", image),
  image_tag = c("latest", "$SHORT_SHA", "$BRANCH_NAME"),
  ...,
  substitutions = NULL,
  ignoredFiles = NULL,
  includedFiles = NULL,
  timeout = NULL,
  projectId_target = cr_project_get()
)
```

Arguments

`repo`	The git repo holding the Dockerfile from cr_buildtrigger_repo
`image`	The name of the image you want to build
`trigger_name`	The trigger name
`image_tag`	What to tag the build docker image
`...`	Arguments passed on to `cr_buildstep_docker` `tag` The tag or tags to be attached to the pushed image - can use `Build` macros `location` Where the Dockerfile to build is in relation to `dir` `projectId` The projectId

	`dockerfile` Specify the name of the Dockerfile found at `location`
	`kaniko_cache` If TRUE will use kaniko cache for Docker builds.
	`build_args` additional arguments to pass to `docker build`, should be a character vector.
	`push_image` if `kaniko_cache = FALSE` and `push_image = FALSE`, then the docker image is simply built and not pushed
`substitutions`	A named list of Build macro variables
`ignoredFiles`	ignored_files and included_files are file glob matches extended with support for `"**"`.
`includedFiles`	If any of the files altered in the commit pass the ignored_files
`timeout`	Timeout for build
`projectId_target`	The project to publish the Docker image to. The image will be built under the project configured via cr_project_get. You will need to give the build project's service email access to the target GCP project via IAM for it to push successfully.

Details

This creates a buildtrigger to do a kamiko cache enabled Docker build upon each commit, as defined by your repo settings via cr_buildtrigger_repo. It will build all tags concurrently.

See Also

cr_deploy_docker which lets you build Dockerfiles for more generic use cases

Other Deployment functions: `cr_deploy_docker()`, `cr_deploy_packagetests()`, `cr_deploy_pkgdown()`, `cr_deploy_run_website()`, `cr_deploy_run()`, `cr_deploy_r()`

Examples

```
## Not run:
repo <- cr_buildtrigger_repo("MarkEdmondson1234/googleCloudRunner")
# create trigger that will publish Docker image to gcr.io/your-project/test upon each GitHub commit
cr_deploy_docker_trigger(repo, "test", dir = "cloud_build")

# build in one project, publish the docker image to another project (gcr.io/another-project/test)
cr_deploy_docker_trigger(repo, "test", projectId_target = "another-project", dir = "cloud_build")

## End(Not run)
```

`cr_deploy_gadget` *Launch the googleCloudRunner deployment RStudio gadget*

Description

You can assign a hotkey to the addin via Tools > Addins > Browse Addins > Keyboard shortcuts. CTRL+SHIFT+D is a suggested hotkey.

Usage

```
cr_deploy_gadget()
```

`cr_deploy_packagetests` *Deploy a cloudbuild.yml for R package tests and upload to Codecov*

Description

This tests an R package each time you commit, and uploads the test coverage results to Codecov

Usage

```
cr_deploy_packagetests(
  steps = NULL,
  cloudbuild_file = "cloudbuild-tests.yml",
  env = c("NOT_CRAN=true"),
  test_script = NULL,
  codecov_script = NULL,
  codecov_token = "$_CODECOV_TOKEN",
  build_image = "gcr.io/gcer-public/packagetools:latest",
  create_trigger = c("file", "inline", "no"),
  trigger_repo = NULL,
  ...
)
```

Arguments

`steps`	extra steps to run before the cr_buildstep_packagetests steps run (such as decryption of auth files)
`cloudbuild_file`	The cloudbuild yaml file to write to. See create_trigger
`env`	Environment arguments to be set during the test script runs
`test_script`	The script that will call rcmdcheck to perform tests. If `NULL` a default script is used in `system.file("r_buildsteps", "devtools_tests.R", package="googlecloudRunner")`

`codecov_script`	The script that will call codecov to perform coverage. If `NULL` a default script is used in `system.file("r_buildsteps", "codecov_tests.R", package="googleCloudRunner")`
`codecov_token`	If using codecov, supply your codecov token here.
`build_image`	The docker image that will be used to run the R code for the test scripts
`create_trigger`	If creating a trigger, whether to create it from the cloudbuild_file or inline
`trigger_repo`	If not NULL, a cr_buildtrigger_repo where a buildtrigger will be created via cr_buildtrigger
`...`	Arguments passed on to `cr_build_make`

- `yaml` A `Yaml` object created from cr_build_yaml or a file location of a .yaml/.yml cloud build file
- `artifacts` Artifacts that may be built via cr_build_yaml_artifact
- `options` Options to pass to a Cloud Build
- `availableSecrets` Secret Manager objects built by cr_build_yaml_secrets
- `logsBucket` The gs:// location of a bucket to put logs in
- `source` A Source object specifying the location of the source files to build, usually created by cr_build_source
- `timeout` Amount of time that this build should be allowed to run, to second
- `images` A list of images to be pushed upon the successful completion of all build
- `substitutions` Substitutions data for 'Build' resource
- `serviceAccount` service account email to be used for the build

Details

The trigger repository needs to hold an R package configured to do tests upon.

For GitHub, the repository will need to be linked to the project you are building within, via `https://console.cloud.google.com/cloud-build/triggers/connect`

If your tests need authentication details, add these via cr_buildstep_secret to the `steps` argument, which will prepend decrypting the authentication file before running the tests.

If you want codecov to ignore some files then also deploy a .covrignore file to your repository - see covr website at `https://covr.r-lib.org/` for details.

See Also

Create your own custom deployment using cr_buildstep_packagetests which this function uses with some defaults

cr_buildstep_packagetests

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_docker()`, `cr_deploy_pkgdown()`, `cr_deploy_run_website()`, `cr_deploy_run()`, `cr_deploy_r()`

Examples

```
# create a local cloudbuild.yml file for packagetests
pd <- cr_deploy_packagetests(create_trigger = "no")
pd

# add a decryption step for an auth file
cr_deploy_packagetests(
  steps = cr_buildstep_secret("my_secret", "auth.json"),
  env = c("NOT_CRAN=true", "MY_AUTH_FILE=auth.json"),
  timeout = 1200,
  create_trigger = "no"
)


# creating a buildtrigger repo for trigger_repo
repo <- cr_buildtrigger_repo("MarkEdmondson1234/googleCloudRunner",
  branch = "master"
)
## Not run:

# will create the file in the repo, and point a buildtrigger at it
cr_deploy_packagetests(create_trigger = "file", trigger_repo = repo)


# will make an inline build within a buildtrigger
cr_deploy_packagetests(create_trigger = "inline", trigger_repo = repo)

## End(Not run)

unlink("cloudbuild-tests.yml")
```

`cr_deploy_pkgdown` *Deploy a cloudbuild.yml for a pkgdown website of an R package*

Description

This builds a pkgdown website each time the trigger fires and deploys it to git

Usage

```
cr_deploy_pkgdown(
  github_repo,
  secret,
  steps = NULL,
  create_trigger = c("file", "inline", "no"),
  cloudbuild_file = "cloudbuild-pkgdown.yml",
  git_email = "googlecloudrunner@r.com",
  env = NULL,
  build_image = "gcr.io/gcer-public/packagetools:latest",
  post_setup = NULL,
  post_clone = NULL
)
```

Arguments

`github_repo`	The GitHub repo to deploy pkgdown website from and to.
`secret`	The name of the secret on Google Secret Manager for the git ssh private key
`steps`	extra steps to run before the pkgdown website steps run
`create_trigger`	If not "no" then the buildtrigger will be setup for you via cr_buildtrigger, if "file" will create a buildtrigger pointing at `cloudbuild_file`, if "inline" will put the build inline within the trigger (no file created)
`cloudbuild_file`	The cloudbuild yaml file to write to
`git_email`	The email the git commands will be identifying as
`env`	A character vector of env arguments to set for all steps
`build_image`	A docker image with pkgdown installed
`post_setup`	Steps that occur after git setup
`post_clone`	A cr_buildstep that occurs after the repo is cloned

Details

The trigger repository needs to hold an R package configured to build a pkgdown website.

For GitHub, the repository will also need to be linked to the project you are building within, via `https://console.cloud.google.com/cloud-build/triggers/connect`

The git ssh keys need to be deployed to Google Secret Manager for the deployment of the website - see cr_buildstep_git - this only needs to be done once per Git account.

See Also

Create your own custom deployment using cr_buildstep_pkgdown which this function uses with some defaults.

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_docker()`, `cr_deploy_packagetests()`, `cr_deploy_run_website()`, `cr_deploy_run()`, `cr_deploy_r()`

Examples

```
pd <- cr_deploy_pkgdown("MarkEdmondson1234/googleCloudRunner",
  secret = "my_git_secret",
  create_trigger = "no"
)
pd
file.exists("cloudbuild-pkgdown.yml")
unlink("cloudbuild-pkgdown.yml")
```

```
## Not run:
cr_deploy_pkgdown("MarkEdmondson1234/googleCloudRunner",
  secret = "my_git_secret",
  create_trigger = "inline"
)

## End(Not run)
```

cr_deploy_r *Deploy an R script with an optional schedule*

Description

Will create a build to run an R script in Cloud Build with an optional schedule from Cloud Scheduler

Usage

```
cr_deploy_r(
  r,
  schedule = NULL,
  source = NULL,
  run_name = NULL,
  r_image = "rocker/verse",
  pre_steps = NULL,
  post_steps = NULL,
  timeout = 600L,
  ...,
  schedule_type = c("pubsub", "http"),
  schedule_pubsub = NULL,
  email = cr_email_get(),
  region = cr_region_get(),
  projectId = cr_project_get(),
  serviceAccount = NULL,
  launch_browser = interactive()
)
```

Arguments

`r`	R code to run or a file containing R code ending with .R, or the gs:// location on Cloud Storage of the R file you want to run
`schedule`	A cron schedule e.g. `"15 5 * * *"`
`source`	A Source object specifying the location of the source files to build, usually created by cr_build_source
`run_name`	What name the R code will identify itself as. If `NULL` one is autogenerated.
`r_image`	The R docker environment executing the R code

`pre_steps`	Other cr_buildstep to run before the R code executes
`post_steps`	Other cr_buildstep to run after the R code executes
`timeout`	Amount of time that this build should be allowed to run, to second
`...`	Arguments passed on to `cr_buildstep_r` `name` The docker image that will run the R code, usually from rocker-project.org `r_source` Whether the R code will be from a runtime file within the source or at build time copying over from a local R file in your session `escape_dollar` Default TRUE. This will turn `$` into `$$` within the script to avoid them being recognised as Cloud Build variables. Turn this off if you want that behaviour (e.g. `my_project="$PROJECT_ID"`) `rscript_args` Optional arguments for the R script run by `Rscript`. `r_cmd` should ‘Rscript‘ be run or ‘R‘? `prefix` prefixed to name - set to "" to suppress. Will be suppressed if `name` starts with gcr.io or `*-docker.pkg.dev`
`schedule_type`	If you have specified a schedule, this will select what strategy it will use to deploy it. See details
`schedule_pubsub`	If you have a custom pubsub message to send via an existing topic, use cr_schedule_pubsub to supply it here
`email`	The email that will authenticate the job set via cr_email_set
`region`	The region usually set with cr_region_set
`projectId`	ID of the project
`serviceAccount`	service account email to be used for the build
`launch_browser`	Whether to launch the logs URL in a browser once deployed

Details

The R script will execute within the root directory of whichever Source you supply, usually created via cr_build_source representing a Cloud Storage bucket or a GitHub repository that is copied across before code execution. Bear in mind if the source changes then the code scheduled may need updating.

The `r_image` dictates what R libraries the R environment executing the code of `r` will have, via the underlying Docker container usually supplied by rocker-project.org. If you want custom R libraries beyond the default, create a docker container with those R libraries installed (perhaps via cr_deploy_docker)

Value

If scheduling then a Job, if building immediately then a Build

Scheduling

If `schedule=NULL` then the R script will be run immediately on Cloud Build via cr_build.

If `schedule` carries a cron job string (e.g. `"15 5 * * *"`) then the build will be scheduled via Cloud Scheduler

If `schedule_type="pubsub"` then you will need `googlePubsubR` installed and set-up and scheduling will involve:

1. Creating a PubSub topic called `"{run_name}-topic"` or subscribing to the one you provided in `schedule_pubsub`. It is assumed you have created the PubSub topic beforehand if you do supply your own.
2. Create a Build Trigger called `"{run_name}-trigger"` that will run when the PubSub topic is called
3. Create a Cloud Schedule called `"{run_name}-trigger"` that will send a pubsub message to the topic: either the default that contains just the name of the script, or the message you supplied in `schedule_pubsub`.

Type "pubsub" is recommended for more complex R scripts as you will have more visibility for debugging schedules via inspecting the PubSub topic, build trigger and build logs, as well as enabling triggering the script from other PubSub topics and allowing to pass dynamic parameters into your schedule scripts via the PubSub message.

If `schedule_type="http"` then scheduling will involve:

1. Create a Cloud Build API call with your build embedded within it via cr_schedule_http
2. Schedule the HTTP call using the authentication email supplied in `email` or the default cr_email_get

This is the old default and is suitable for smaller R scripts or when you don't want to use the other GCP services. The authentication for the API call from Cloud Scheduler can cause opaque errors as it will give you invalid response codes whether its that or an error in your R script you wish to schedule.

See Also

If you want to run R code upon certain events like GitHub pushes, look at cr_buildtrigger

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_docker()`, `cr_deploy_packagetests()`, `cr_deploy_pkgdown()`, `cr_deploy_run_website()`, `cr_deploy_run()`

Examples

```
r_lines <- c(
  "list.files()",
  "library(dplyr)",
  "mtcars %>% select(mpg)",
  "sessionInfo()"
)
source <- cr_build_source(RepoSource("googleCloudStorageR",
  branchName = "master"
))
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_email_set("123456@projectid.iam.gserviceaccount.com")
```

```
# check the script runs ok
cr_deploy_r(r_lines, source = source)

# schedule the script
cr_deploy_r(r_lines, schedule = "15 21 * * *", source = source)

## End(Not run)
```

`cr_deploy_run` *Deploy to Cloud Run*

Description

Deploy R api plumber scripts, HTML files or other images create the Docker image, add the build to Cloud Build and deploy to Cloud Run

Usage

```
cr_deploy_run(
  local,
  remote = basename(local),
  dockerfile = NULL,
  image_name = remote,
  tag = "$BUILD_ID",
  region = cr_region_get(),
  bucket = cr_bucket_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  timeout = 600L,
  kaniko_cache = TRUE,
  pre_steps = NULL,
  post_steps = NULL,
  ...
)

cr_deploy_html(
  html_folder,
  remote = basename(html_folder),
  image_name = remote,
  tag = "$BUILD_ID",
  region = cr_region_get(),
  bucket = cr_bucket_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  timeout = 600L,
  ...
)
```

```
cr_deploy_plumber(
  api,
  remote = basename(api),
  dockerfile = NULL,
  image_name = remote,
  tag = "$BUILD_ID",
  region = cr_region_get(),
  bucket = cr_bucket_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  timeout = 600L,
  ...
)
```

Arguments

`local`	A folder containing the scripts and Dockerfile to deploy to Cloud Run
`remote`	The folder on Google Cloud Storage, and the name of the service on Cloud Run
`dockerfile`	An optional Dockerfile built to support the script. Not needed if 'Dockerfile' exists in folder. If supplied will be copied into deployment folder and called "Dockerfile"
`image_name`	The gcr.io image name that will be deployed and/or built
`tag`	The tag or tags to be attached to the pushed image - can use `Build` macros
`region`	The Cloud Run endpoint set by CR_REGION env arg
`bucket`	The Cloud Storage bucket that will hold the code
`projectId`	The projectId where it all gets deployed to
`launch_browser`	Whether to launch the logs URL in a browser once deployed
`timeout`	Amount of time that this build should be allowed to run, to second
`kaniko_cache`	If TRUE will use kaniko cache for Docker builds.
`pre_steps`	Other cr_buildstep to run before the docker build
`post_steps`	Other cr_buildstep to run after the docker build

`...`	Arguments passed on to `cr_buildstep_run`
	`name` Name for deployment on Cloud Run
	`image` The name of the image to create or use in deployment - `gcr.io`
	`allowUnauthenticated` TRUE if can be reached from public HTTP address. If FALSE will configure a service-email called `(name)-cloudrun-invoker@(project-id).iam.gser`
	`concurrency` How many connections each container instance can serve. Can be up to 80.
	`port` Container port to receive requests at. Also sets the $PORT environment variable. Must be a number between 1 and 65535, inclusive. To unset this field, pass the special value "default".
	`max_instances` the desired maximum nuimber of container instances. "default" is 1000, you can get more if you requested a quota instance. For Shiny instances on Cloud Run, this needs to be 1.

- `memory` The format for size is a fixed or floating point number followed by a unit: G, M, or K corresponding to gigabyte, megabyte, or kilobyte, respectively, or use the power-of-two equivalents: Gi, Mi, Ki corresponding to gibibyte, mebibyte or kibibyte respectively. The default is 256Mi
- `cpu` 1 or 2 CPUs for your instance
- `env_vars` Environment arguments passed to the Cloud Run container at runtime. Distinct from `env` that run at build time.
- `gcloud_args` a character string of arguments that can be sent to the gcloud command not covered by other parameters of this function

`html_folder`	the folder containing all the html
`api`	A folder containing the R script using plumber called api.R and all its dependencies

Details

These deploy containers to Cloud Run, a scale 0-to-millions container-as-a-service on Google Cloud Platform.

cr_deploy_html

Deploy html files to a nginx server on Cloud Run.

Supply the html folder to host it on Cloud Run. Builds the dockerfile with the html within it, then deploys to Cloud Run

Will add a `default.template` file to the html folder that holds the nginx configuration

cr_deploy_plumber

The entrypoint for CloudRun will be via a plumber script called api.R - this should be included in your local folder to deploy. From that api.R you can source or call other resources in the same folder, using relative paths.

The function will create a local folder called "deploy" and a tar.gz of that folder which is what is being uploaded to Google Cloud Storage

See Also

For scheduling Cloud Run apps cr_run_schedule_http

cr_deploy_run_website which has more features like rending Rmd files and deploying upon each git commit

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_docker()`, `cr_deploy_packagetests()`, `cr_deploy_pkgdown()`, `cr_deploy_run_website()`, `cr_deploy_r()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_bucket_set("my-bucket")
```

```
cr_deploy_run(system.file("example/", package = "googleCloudRunner"))

## End(Not run)
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_bucket_set("my-bucket")

cr_deploy_html("my_folder")

## End(Not run)
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_bucket_set("my-bucket")

cr_deploy_plumber(system.file("example/", package = "googleCloudRunner"))

## End(Not run)
```

`cr_deploy_run_website` *Deploy HTML built from a repo each commit*

Description

This lets you set up triggers that will update an R generated website each commit.

Usage

```
cr_deploy_run_website(
  repo,
  image = paste0("website-", format(Sys.Date(), "%Y%m%d")),
  rmd_folder = NULL,
  html_folder = NULL,
  image_tag = "$SHORT_SHA",
  timeout = 600L,
  edit_r = NULL,
  r_image = "gcr.io/gcer-public/packagetools:latest",
  allowUnauthenticated = TRUE,
  region = cr_region_get(),
  projectId = cr_project_get()
)
```

Arguments

`repo`	A git repository defined in cr_buildtrigger_repo
`image`	The name of the image you want to build

rmd_folder	A folder of Rmd files within GitHub source that will be built into HTML for serving via render
html_folder	A folder of html to deploy within GitHub source. Will be ignored if rmd_folder is not NULL
image_tag	What to tag the build docker image
timeout	Timeout for the build
edit_r	If you want to change the R code to render the HTML, supply R code via a file or string of R as per cr_buildstep_r
r_image	The image that will run the R code from `edit_r`
allowUnauthenticated	TRUE if can be reached from public HTTP address. If FALSE will configure a service-email called `(name)-cloudrun-invoker@(project-id).iam.gserviceaccount.com`
region	The region for cloud run
projectId	The GCP projectId which will be deployed within

Details

This lets you render the Rmd (or other R functions that produce HTML) in a folder for your repo, which will then be hosted on a Cloud Run enabled with nginx. Each time you push to git with modified Rmd code, it will build the new HTML and push an update to the website.

This default R code is rendered in the rmd_folder:

`lapply(list.files('.', pattern = '.Rmd', full.names = TRUE), rmarkdown::render, output_format = 'html_document')`

See Also

cr_deploy_html that lets you deploy just HTML files and cr_deploy_pkgdown for running pkgdown websites.

Other Deployment functions: `cr_deploy_docker_trigger()`, `cr_deploy_docker()`, `cr_deploy_packagetests()`, `cr_deploy_pkgdown()`, `cr_deploy_run()`, `cr_deploy_r()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
your_repo <- cr_buildtrigger_repo("MarkEdmondson1234/googleCloudRunner")
cr_deploy_run_website(your_repo, rmd_folder = "vignettes")

# change the Rmd rendering to pkgdown
r <- "devtools::install();pkgdown::build_site()"

cr_deploy_run_website(your_repo,
  image = paste0(your_repo, "-pkgdown"),
  rmd_folder = ".",
  edit_r = r
)
```

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

`cr_email_get` *Get/Set cloud build email*

Description

Needed so Cloud Scheduler can run Cloud Build jobs - can also set via environment argument CR_BUILD_EMAIL

Usage

```
cr_email_get()

cr_email_set(cloudbuildEmail)
```

Arguments

`cloudbuildEmail`	The Cloud Build service email

See Also

https://console.cloud.google.com/cloud-build/settings

Examples

```
cr_email_set("myemail@domain.com")
cr_email_get()
```

`cr_jwt_create` *Create a JSON Web Token (JWT) from your service client and call Google services*

Description

This can be used to call authenticated services such as Cloud Run.

Usage

```
cr_jwt_create(the_url, service_json = Sys.getenv("GCE_AUTH_FILE"))

cr_jwt_token(signed_jwt, the_url)

cr_jwt_with_httr(req, token)

cr_jwt_with_curl(h = curl::new_handle(), token)

cr_jwt_async(urls, token, ...)
```

Arguments

`the_url`	The URL of the service you want to call
`service_json`	The account service key JSON that will be used to generate the JWT
`signed_jwt`	A JWT created from cr_jwt_create
`req`	A `httr` request to the service running on `the_url`, using httr verbs such as GET
`token`	The token created via cr_jwt_token
`h`	A curl handle such as set with new_handle
`urls`	URLs to request asynchronously
`...`	Other arguments passed to new_handle

Details

For certain Google services a JWT is needed to authenticate access, which is distinct from OAuth2. An example of this is authenticated Cloud Run such as deployed when using cr_run and parameter `allowUnauthenticated = FALSE`. These functions help you call your services by generating the JWT from your service account key.

The token is set to expire in 1 hour, so it will need refreshing before then by calling this function again.

See Also

Service-to-service authentication on GCP

Other Cloud Run functions: `cr_plumber_pubsub()`, `cr_run_email()`, `cr_run_get()`, `cr_run_list()`, `cr_run_schedule_http()`, `cr_run()`

Examples

```
## Not run:

# The private authenticated access only Cloud Run service
the_url <- "https://authenticated-cloudrun-ewjogewawq-ew.a.run.app/"

# creating the JWT and token
jwt <- cr_jwt_create(the_url)
token <- cr_jwt_token(jwt, the_url)
```

```
# call Cloud Run app using token with any httr verb
library(httr)
res <- cr_jwt_with_httr(
  GET("https://authenticated-cloudrun-ewjogewawq-ew.a.run.app/hello"),
  token
)
content(res)

# call Cloud Run app with curl - you can pass in a curl handle
library(curl)
h <- new_handle()
handle_setopt(h, customrequest = "PUT")
handle_setform(h, a = "1", b = "2")
h <- cr_jwt_with_curl(h, token = token)
r <- curl_fetch_memory("https://authenticated-cloudrun-ewjogewawq-ew.a.run.app/hello", h)
cat(rawToChar(r$content))

# use curls multi-asynch functions
many_urls <- paste0(
  "https://authenticated-cloudrun-ewjogewawq-ew.a.run.app/hello",
  paste0("?param="), 1:6
)
cr_jwt_async(many_urls, token = token)

## End(Not run)
```

`cr_plumber_pubsub` *Plumber - Pub/Sub parser*

Description

A function to use in plumber scripts to accept Pub/Sub messages

Usage

```
cr_plumber_pubsub(message = NULL, pass_f = function(x) x)
```

Arguments

`message`	The pubsub message
`pass_f`	An R function that will work with the data parsed out of the pubsub `message$data` field.

Details

This function is intended to be used within plumb API scripts. It needs to be annotated with a `@post` URL route and a `@param message The pubsub message` as per the plumber documentation.

`pass_f` should be a function you create that accepts one argument, the data from the pubsub `message$data` field. It is unencoded for you. Make sure the function returns a 200 response otherwise pub/sub will keep resending the message! `return(TRUE)` is adequate.

The Docker container for the API will need to include `googleCloudRunner` installed in its R environment to run this function. This is available in the public `gcr.io/gcer-public/cloudrunner` image.

Use cr_pubsub to test this function once deployed.

See Also

Google Pub/Sub tutorial for Cloud Run. You can set up Pub/Sub messages from Google Cloud Storage buckets via gcs_create_pubsub

Other Cloud Run functions: `cr_jwt_create()`, `cr_run_email()`, `cr_run_get()`, `cr_run_list()`, `cr_run_schedule_http()`, `cr_run()`

Examples

```
## Not run:

# within a plumber api.R script:

# example function echos back pubsub message
pub <- function(x) {
  paste("Echo:", x)
}

#' Recieve pub/sub message
#' @post /pubsub #nolint
#' @param message a pub/sub message
function(message = NULL) {
  googleCloudRunner::cr_plumber_pubsub(message, pub)
}

## End(Not run)
```

`cr_project_set` *Get/Set the projectId for your CloudRun services*

Description

Can also use environment argument GCE_DEFAULT_PROJECT_ID

Usage

```
cr_project_set(projectId)

cr_project_get()
```

Arguments

`projectId`	The projectId

Examples

```
cr_project_get()
```

`cr_pubsub` *Send a message to pubsub*

Description

Useful for testing Cloud Run pubsub deployments

Usage

```
cr_pubsub(endpoint, payload = jsonlite::toJSON("hello"))
```

Arguments

`endpoint`	The url endpoint of the PubSub service
`payload`	Will be base64 encoded and placed in `message$data`

`cr_regions` *Cloud Run Regions*

Description

Cloud Run Regions

Usage

```
cr_regions
```

Format

A character vector of valid Cloud Run region names

`cr_region_set` *Get/Set the endpoint for your CloudRun services*

Description

Can also use environment argument CR_REGION

Usage

```
cr_region_set(region = googleCloudRunner::cr_regions)

cr_region_get()
```

Arguments

`region`	Region for the endpoint

Examples

```
cr_region_get()
```

`cr_run` *Create a CloudRun service.*

Description

Deploys an existing gcr.io image.

Usage

```
cr_run(
  image,
  name = basename(image),
  allowUnauthenticated = TRUE,
  concurrency = 1,
  port = NULL,
  max_instances = "default",
  memory = "256Mi",
  cpu = 1,
  timeout = 600L,
  region = cr_region_get(),
  projectId = cr_project_get(),
  launch_browser = interactive(),
  env_vars = NULL,
  gcloud_args = NULL,
  ...
)
```

Arguments

`image`	The name of the image to create or use in deployment - `gcr.io`
`name`	Name for deployment on Cloud Run
`allowUnauthenticated`	TRUE if can be reached from public HTTP address. If FALSE will configure a service-email called `(name)-cloudrun-invoker@(project-id).iam.gserviceaccount.com`
`concurrency`	How many connections each container instance can serve. Can be up to 80.
`port`	Container port to receive requests at. Also sets the $PORT environment variable. Must be a number between 1 and 65535, inclusive. To unset this field, pass the special value "default".
`max_instances`	the desired maximum nuimber of container instances. "default" is 1000, you can get more if you requested a quota instance. For Shiny instances on Cloud Run, this needs to be 1.
`memory`	The format for size is a fixed or floating point number followed by a unit: G, M, or K corresponding to gigabyte, megabyte, or kilobyte, respectively, or use the power-of-two equivalents: Gi, Mi, Ki corresponding to gibibyte, mebibyte or kibibyte respectively. The default is 256Mi
`cpu`	1 or 2 CPUs for your instance
`timeout`	Amount of time that this build should be allowed to run, to second
`region`	The endpoint region for deployment
`projectId`	The GCP project from which the services should be listed
`launch_browser`	Whether to launch the logs URL in a browser once deployed
`env_vars`	Environment arguments passed to the Cloud Run container at runtime. Distinct from `env` that run at build time.
`gcloud_args`	a character string of arguments that can be sent to the gcloud command not covered by other parameters of this function
`...`	Arguments passed on to `cr_buildstep_run`

Details

Uses Cloud Build to deploy an image to Cloud Run

See Also

Google Documentation for Cloud Run

Use cr_deploy_docker or similar to create image, cr_deploy_run to automate building and deploying, cr_deploy_plumber to deploy plumber APIs.

Deploying Cloud Run using Cloud Build

Other Cloud Run functions: `cr_jwt_create()`, `cr_plumber_pubsub()`, `cr_run_email()`, `cr_run_get()`, `cr_run_list()`, `cr_run_schedule_http()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_run("gcr.io/my-project/my-image")
cr_run("gcr.io/cloud-tagging-10302018/gtm-cloud-image:stable",
  env_vars = c("CONTAINER_CONFIG=xxxxxxx")
)

## End(Not run)
```

`cr_run_email` *Create an invoker email for use within authenticated Cloud Run*

Description

Create an invoker email for use within authenticated Cloud Run

Usage

```
cr_run_email(name, projectId = cr_project_get())
```

Arguments

`name`	Name of the Cloud Run service
`projectId`	The projectId where the Cloud Run service will run - set to NULL to only return the processed service name

See Also

Other Cloud Run functions: `cr_jwt_create()`, `cr_plumber_pubsub()`, `cr_run_get()`, `cr_run_list()`, `cr_run_schedule_http()`, `cr_run()`

Examples

```
cr_run_email("my-run-app", "my-project")
```

`cr_run_get` *Get information about a Cloud Run service.*

Description

Get information about a Cloud Run service.

Usage

```
cr_run_get(name, projectId = cr_project_get())
```

Arguments

`name`	The name of the service to retrieve
`projectId`	The projectId to get from

Details

This returns details on a particular deployed Cloud Run service.

See Also

Google Documentation on namespaces.services.get

Other Cloud Run functions: `cr_jwt_create()`, `cr_plumber_pubsub()`, `cr_run_email()`, `cr_run_list()`, `cr_run_schedule_http()`, `cr_run()`

`cr_run_list` *List CloudRun services.*

Description

List the Cloud Run services you have access to

Usage

```
cr_run_list(
  projectId = cr_project_get(),
  labelSelector = NULL,
  limit = NULL,
  summary = TRUE
)
```

Arguments

`projectId`	The GCP project from which the services should be listed
`labelSelector`	Allows to filter resources based on a label
`limit`	The maximum number of records that should be returned
`summary`	If TRUE will return only a subset of info available, set to FALSE for all metadata

See Also

Google Documentation for Cloud Run

Other Cloud Run functions: `cr_jwt_create()`, `cr_plumber_pubsub()`, `cr_run_email()`, `cr_run_get()`, `cr_run_schedule_http()`, `cr_run()`

`cr_run_schedule_http` *Create a Cloud Scheduler HTTP target for a private Cloud Run URI*

Description

This enables Cloud Scheduler to trigger Cloud Run endpoints when they are not public.

Usage

```
cr_run_schedule_http(uri, email, http_method = "GET", body = NULL)
```

Arguments

`uri`	The URI of your Cloud Run application
`email`	The service email that has invoke access to the Cloud Run application. If using cr_run and derivatives to make the email this will include `(name)-cloudrun-invoker@(project-id).ia` - see cr_run_email to help make the email.
`http_method`	The HTTP verb you have set up your Cloud Run application to receive
`body`	(optional) An R list object that will be turned into JSON via toJSON and turned into a base64-encoded string if you are doing a POST, PUT or PATCH request.

Details

Ensure you have a service email with cr_email_set of format `service-{project-number}@gcp-sa-cloudscheduler.iam.g` with Cloud Scheduler Service Agent role as per https://cloud.google.com/scheduler/docs/http-target-auth#add

Value

A HttpTarget object for use in cr_schedule

See Also

https://cloud.google.com/run/docs/triggering/using-scheduler

cr_schedule_http and cr_run and cr_deploy_run

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

Other Cloud Run functions: `cr_jwt_create()`, `cr_plumber_pubsub()`, `cr_run_email()`, `cr_run_get()`, `cr_run_list()`, `cr_run()`

Examples

```
## Not run:
# for unauthenticated apps create a HttpTarget
run_me <- HttpTarget(
  uri = "https://public-ewjogewawq-ew.a.run.app/echo?msg=blah",
  http_method = "GET"
)
cr_schedule("cloud-run-scheduled",
  schedule = "16 4 * * *",
  httpTarget = run_me
)

# for authenticated Cloud Run apps - create with allowUnauthenticated=FALSE
cr_deploy_run("my-app", allowUnauthenticated = TRUE)

## End(Not run)

# deploying via R will help create a service email called my-app-cloudrun-invoker
cr_run_email("my-app")
## Not run:
# use that email to schedule the Cloud Run private micro-service

# schedule the endpoint
my_run_name <- "my-app"
my_app <- cr_run_get(my_run_name)
email <- cr_run_email(my_run_name)
endpoint <- paste0(my_app$status$url, "/fetch_stuff")

app_sched <- cr_run_schedule_http(endpoint,
  http_method = "GET",
  email = email
)

cr_schedule("cloud-run-scheduled-1",
  schedule = "4 16 * * *",
  httpTarget = app_sched
)

## End(Not run)
```

cr_schedule_build *Schedule a Build object via HTTP or PubSub*

Description

Schedule a Build object via HTTP or PubSub

Usage

```
cr_schedule_build(
  build,
  schedule,
  schedule_type = c("http", "pubsub"),
  email = cr_email_get(),
  projectId = cr_project_get(),
  ...
)
```

Arguments

`build`	A Build object
`schedule`	A cron schedule e.g. `"15 5 * * *"`
`schedule_type`	Whether to use HTTP or PubSub styled schedules
`email`	The email that will authenticate the job set via cr_email_set
`projectId`	The GCP project to run within usually set with cr_project_set
`...`	Arguments passed on to `cr_schedule` `region` The region usually set with cr_region_set `overwrite` If TRUE and an existing job with the same name exists, will overwrite it with the new parameters `name` Name to call your scheduled job `httpTarget` A HTTP target object HttpTarget `pubsubTarget` A Pub/Sub target object PubsubTarget such as created via cr_schedule_pubsub `description` Optionally caller-specified in CreateJob or `timeZone` Specifies the time zone to be used in interpreting schedule. If set to `NULL` will be "UTC". Note that some time zones include a provision for daylight savings time.

Details

See also cr_schedule which you can use by to customise your schedule.

Value

`cr_schedule_build` returns a cloud scheduler Job object

cr_schedule_delete *Deletes a scheduled job.*

Description

Deletes a scheduled job.

Usage

```
cr_schedule_delete(
  x,
  region = cr_region_get(),
  projectId = cr_project_get(),
  pubsub_cleanup = FALSE
)
```

Arguments

x	The name of the scheduled job or a Job object
region	The region to run within
projectId	The projectId
pubsub_cleanup	If the Cloud Scheduler is pointing at a Build Trigger/PubSub as deployed by cr_deploy_r will attempt to clean up those resources too.

Value

TRUE if job not found or its deleted, FALSE if it could not delete the job

See Also

cloudscheduler.projects.locations.jobs.delete

Other Cloud Scheduler functions: HttpTarget(), Job(), PubsubTarget(), cr_build_schedule_http(), cr_run_schedule_http(), cr_schedule_get(), cr_schedule_list(), cr_schedule_pause(), cr_schedule_run()

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule_delete("cloud-build-test1")


## End(Not run)
```

`cr_schedule_get` *Gets a scheduler job.*

Description

Gets a scheduler job.

Usage

```
cr_schedule_get(name, region = cr_region_get(), projectId = cr_project_get())
```

Arguments

`name`	Required - a string or a schedule Job object
`region`	The region to run within
`projectId`	The projectId

See Also

Google Documentation

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule_get("cloud-build-test1")

## End(Not run)
```

`cr_schedule_list` *Lists Cloud Scheduler jobs.*

Description

Lists cloud scheduler jobs including targeting, schedule and authentication

Usage

```
cr_schedule_list(region = cr_region_get(), projectId = cr_project_get())
```

Arguments

`region`	The region to run within
`projectId`	The projectId

See Also

Google Documentation

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_pause()`, `cr_schedule_run()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule_list()


## End(Not run)
```

`cr_schedule_pause` *Pauses and resumes a scheduled job.*

Description

If a job is paused then the system will stop executing the job until it is re-enabled via cr_schedule_resume.

Usage

```
cr_schedule_pause(x, region = cr_region_get(), projectId = cr_project_get())

cr_schedule_resume(x, region = cr_region_get(), projectId = cr_project_get())
```

Arguments

`x`	The name of the scheduled job or a Job object
`region`	The region to run within
`projectId`	The projectId

Details

The state of the job is stored in state; if paused it will be set to Job.State.PAUSED. A job must be in Job.State.ENABLED to be paused.

See Also

cloudscheduler.projects.locations.jobs.pause

cloudscheduler.projects.locations.jobs.resume

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_run()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule_pause("cloud-build-test1")
cr_schedule_resume("cloud-build-test1")

## End(Not run)
```

`cr_schedule_run` *Forces a job to run now.*

Description

When this method is called, Cloud Scheduler will dispatch the job, even if the job is already running.

Usage

```
cr_schedule_run(x, region = cr_region_get(), projectId = cr_project_get())
```

Arguments

`x`	The name of the scheduled job or a Job object
`region`	The region to run within
`projectId`	The projectId

See Also

cloudscheduler.projects.locations.jobs.run

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`

Examples

```
## Not run:
cr_project_set("my-project")
cr_region_set("europe-west1")
cr_schedule_run("cloud-build-test1")

## End(Not run)
```

`cr_setup` *A helper setup function for setting up use with googleCloudRunner*

Description

A helper setup function for setting up use with googleCloudRunner

Usage

```
cr_setup()
```

See Also

Other setup functions: `cr_setup_auth()`, `cr_setup_service()`, `cr_setup_test()`

`cr_setup_auth` *Create a service account for googleCloudRunner*

Description

This will use your Google OAuth2 user to create a suitable service account

Usage

```
cr_setup_auth(
  email = Sys.getenv("GARGLE_EMAIL"),
  file = "googlecloudrunner-auth-key.json",
  session_user = NULL
)
```

Arguments

`email`	What email to open OAuth2 with
`file`	Where to save the authentication file
`session_user`	1 for user level, 2 for project level, leave `NULL` to be prompted

Value

TRUE if the file is ready to be setup by cr_setup, FALSE if need to stop

See Also

Other setup functions: `cr_setup_service()`, `cr_setup_test()`, `cr_setup()`

---

cr_setup_service   *Give a service account the right permissions for googleCloudRunner operations*

Description

Give a service account the right permissions for googleCloudRunner operations

Usage

```
cr_setup_service(
  account_email,
  roles = cr_setup_role_lookup("local"),
  json = Sys.getenv("GAR_CLIENT_JSON"),
  email = Sys.getenv("GARGLE_EMAIL")
)

cr_setup_role_lookup(
 type = c("local", "cloudrun", "bigquery", "secrets", "cloudbuild", "cloudstorage",
    "schedule_agent", "run_agent", "compute")
)
```

Arguments

`account_email`	The service account email e.g. `accountId@projectid.iam.gserviceaccount.com` or 12345678@cloudbuild.gserviceaccount.com
`roles`	the roles to grant access - default is all googleCloudRunner functions
`json`	the project clientId JSON
`email`	the email of an Owner/Editor for the project
`type`	the role

See Also

Other setup functions: `cr_setup_auth()`, `cr_setup_test()`, `cr_setup()`

`cr_setup_test` *Run tests over your setup*

Description

This allows you to check if your setup works - run cr_setup first.

Usage

```
cr_setup_test(
  option = c("menu", "all", "docker", "plumber", "r_script", "r_schedule")
)
```

Arguments

`option`	Default will use an interactive menu, select other option to run that test without a menu

See Also

Other setup functions: `cr_setup_auth()`, `cr_setup_service()`, `cr_setup()`

Examples

```
## Not run:
# start the menu for interactive use
cr_setup_test()

# skip menu and run all tests
cr_setup_test("all")

# run just the plumber deployment test
cr_setup_test("plumber")

## End(Not run)
```

`cr_sourcerepo_list` *List source repositories available under a project*

Description

List source repositories available under a project

Usage

```
cr_sourcerepo_list(projectId = cr_project_get())
```

Arguments

`projectId`	The projectId that holds the repositories

`GitHubEventsConfig` *GitHubEventsConfig Object*

Description

GitHubEventsConfig Object

Usage

```
GitHubEventsConfig(
  x,
  event = c("push", "pull"),
  branch = ".*",
  tag = NULL,
  commentControl = c("COMMENTS_DISABLED", "COMMENTS_ENABLED")
)
```

Arguments

`x`	The repository in format owner/repo e.g. MarkEdmondson1234/googleCloudRunner
`event`	Whether to trigger on push or pull GitHub events
`branch`	Regex of branches to match
`tag`	If a push request, regexes matching what tags to build. If not `NULL` then argument `branch` will be ignored
`commentControl`	If a pull request, whether to require comments before builds are triggered.

Details

The syntax of the regular expressions accepted is the syntax accepted by RE2 and described at `https://github.com/google/re2/wiki/Syntax`

Value

GitHubEventsConfig object

See Also

Other BuildTrigger functions: `BuildTrigger()`, `cr_buildtrigger_copy()`, `cr_buildtrigger_delete()`, `cr_buildtrigger_edit()`, `cr_buildtrigger_get()`, `cr_buildtrigger_list()`, `cr_buildtrigger_pubsub()`, `cr_buildtrigger_repo()`, `cr_buildtrigger_run()`, `cr_buildtrigger_webhook()`, `cr_buildtrigger()`

googleCloudRunner — *Launch R scripts into the Google Cloud via Cloud Build, Cloud Run and Cloud Scheduler*

Description

See website for more details: https://code.markedmondson.me/googleCloudRunner/

HttpTarget — *HttpTarget Object*

Description

HttpTarget Object

Usage

```
HttpTarget(
  headers = NULL,
  body = NULL,
  oauthToken = NULL,
  uri = NULL,
  oidcToken = NULL,
  httpMethod = NULL
)
```

Arguments

`headers`	A named list of HTTP headers e.g. `list(Blah = "yes", Boo = "no")`
`body`	HTTP request body. Just send in the R object/list, which will be base64encoded correctly
`oauthToken`	If specified, an OAuth token will be generated and attached as an Authorization header in the HTTP request. This type of authorization should be used when sending requests to a GCP endpoint.
`uri`	Required
`oidcToken`	If specified, an OIDC token will be generated and attached as an Authorization header in the HTTP request. This type of authorization should be used when sending requests to third party endpoints or Cloud Run.
`httpMethod`	Which HTTP method to use for the request

Value

HttpTarget object

See Also

https://cloud.google.com/scheduler/docs/reference/rest/v1/projects.locations.jobs#HttpTarget

Other Cloud Scheduler functions: `Job()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

`Job` *Job Schedule Object*

Description

Job Schedule Object

Usage

```
Job(
  name = NULL,
  description = NULL,
  schedule = NULL,
  timeZone = NULL,
  userUpdateTime = NULL,
  state = NULL,
  status = NULL,
  scheduleTime = NULL,
  lastAttemptTime = NULL,
  retryConfig = NULL,
  attemptDeadline = NULL,
  pubsubTarget = NULL,
  appEngineHttpTarget = NULL,
  httpTarget = NULL
)
```

Arguments

`name`	Name to call your scheduled job
`description`	Optionally caller-specified in CreateJob or
`schedule`	A cron schedule e.g. `"15 5 * * *"`
`timeZone`	Specifies the time zone to be used in interpreting schedule. If set to `NULL` will be "UTC". Note that some time zones include a provision for daylight savings time.
`userUpdateTime`	Output only
`state`	Output only
`status`	Output only
`scheduleTime`	Output only

`lastAttemptTime`	Output only
`retryConfig`	Settings that determine the retry behavior
`attemptDeadline`	The deadline for job attempts
`pubsubTarget`	A Pub/Sub target object PubsubTarget such as created via cr_schedule_pubsub
`appEngineHttpTarget`	App Engine HTTP target
`httpTarget`	A HTTP target object HttpTarget

Details

Configuration for a job.The maximum allowed size for a job is 100KB.

Value

Job object

See Also

Other Cloud Scheduler functions: `HttpTarget()`, `PubsubTarget()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

`PubsubConfig` *Pubsub Config (Build Trigger)*

Description

PubsubConfig describes the configuration of a trigger that creates a build whenever a Pub/Sub message is published.

Usage

```
PubsubConfig(
  subscription = NULL,
  topic = NULL,
  serviceAccountEmail = NULL,
  state = NULL
)
```

Arguments

`subscription`	Output only. Name of the subscription.
`topic`	The name of the topic from which this subscription is receiving messages.
`serviceAccountEmail`	Service account that will make the push request.
`state`	Potential issues with the underlying Pub/Sub subscription configuration. Only populated on get requests.

Value

A PubsubConfig object

See Also

‘https://cloud.google.com/build/docs/api/reference/rest/v1/projects.locations.triggers#BuildTrigger.PubsubConfig‘

`PubsubTarget` *Pubsub Target Object (Cloud Scheduler)*

Description

Pubsub Target Object (Cloud Scheduler)

Usage

```
PubsubTarget(topicName = NULL, data = NULL, attributes = NULL)
```

Arguments

`topicName`	The name of the Cloud Pub/Sub topic to which messages will be published when a job is delivered.
`data`	The message payload for PubsubMessage. An R object that will be turned into JSON via [jsonlite] and then base64 encoded into the PubSub format.
`attributes`	Attributes for PubsubMessage.

Details

Pub/Sub target. The job will be delivered by publishing a message to the given Pub/Sub topic.

Value

PubsubTarget object

See Also

Other Cloud Scheduler functions: `HttpTarget()`, `Job()`, `cr_build_schedule_http()`, `cr_run_schedule_http()`, `cr_schedule_delete()`, `cr_schedule_get()`, `cr_schedule_list()`, `cr_schedule_pause()`, `cr_schedule_run()`

RepoSource *RepoSource Object*

Description

RepoSource Object

Usage

```
RepoSource(
  repoName = NULL,
  tagName = NULL,
  commitSha = NULL,
  branchName = NULL,
  dir = NULL,
  projectId = NULL
)
```

Arguments

`repoName`	Name of the Cloud Source Repository
`tagName`	Regex matching tags to build
`commitSha`	Explicit commit SHA to build
`branchName`	Regex matching branches to build e.g. ".*"
`dir`	Directory, relative to the source root, in which to run the build
`projectId`	ID of the project that owns the Cloud Source Repository

Details

Location of the source in a Google Cloud Source Repository.

Only one of commitSha, branchName or tagName are allowed.

If you want to use GitHub or BitBucket repos, you need to setup mirroring them via Cloud Source Repositories https://source.cloud.google.com/

Value

RepoSource object

See Also

Other Cloud Build functions: `Build()`, `Source()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
## Not run:

my_repo <- cr_build_source(
  RepoSource("github_markedmondson1234_googlecloudrunner",
    branchName = "master"
  )
)

build <- cr_build(
  cr_build_yaml(
    steps =
      cr_buildstep("gcloud", c("-c", "ls -la"),
        entrypoint = "bash",
        dir = ""
      )
  ),
  source = my_repo
)

## End(Not run)
```

`Source` *Source Object*

Description

It is suggested to use cr_build_source instead to build sources

Usage

```
Source(storageSource = NULL, repoSource = NULL)
```

Arguments

`storageSource`	If provided via StorageSource, get the source from this location in Google Cloud Storage
`repoSource`	If provided via RepoSource, get the source from this location in a Cloud Source

Details

Location of the source in a supported storage service.

Value

Source object

See Also

Other Cloud Build functions: `Build()`, `RepoSource()`, `StorageSource()`, `cr_build_artifacts()`, `cr_build_list()`, `cr_build_logs()`, `cr_build_make()`, `cr_build_status()`, `cr_build_targets()`, `cr_build_upload_gcs()`, `cr_build_wait()`, `cr_build_write()`, `cr_build_yaml_artifact()`, `cr_build_yaml_secrets()`, `cr_build_yaml()`, `cr_build()`

Examples

```
cr_project_set("my-project")
cr_bucket_set("my-bucket")
my_gcs_source <- Source(storageSource = StorageSource(
  "my_code.tar.gz",
  "gs://my-bucket"
))
my_repo_source <- Source(repoSource = RepoSource("https://my-repo.com",
  branchName = "master"
))
## Not run:

build1 <- cr_build("cloudbuild.yaml", source = my_gcs_source)
build2 <- cr_build("cloudbuild.yaml", source = my_repo_source)

## End(Not run)
```

`StorageSource` *StorageSource Object*

Description

StorageSource Object

Usage

```
StorageSource(object, bucket = NULL, generation = NULL)
```

Arguments

`object`	Google Cloud Storage object containing the source. This object must be a gzipped archive file (.tar.gz) containing source to build.
`bucket`	Google Cloud Storage bucket containing the source
`generation`	Google Cloud Storage generation for the object. If the generation is omitted, the latest generation will be used.

Details

Location of the source in an archive file in Google Cloud Storage.

Value

StorageSource object

See Also

Other Cloud Build functions: Build(), RepoSource(), Source(), cr_build_artifacts(), cr_build_list(), cr_build_logs(), cr_build_make(), cr_build_status(), cr_build_targets(), cr_build_upload_gcs(), cr_build_wait(), cr_build_write(), cr_build_yaml_artifact(), cr_build_yaml_secrets(), cr_build_yaml(), cr_build()

Examples

```
## Not run:
cr_project_set("my-project")
cr_bucket_set("my-bucket")
# construct Source object
my_gcs_source <- Source(storageSource = StorageSource(
  "my_code.tar.gz",
  "gs://my-bucket"
))
build1 <- cr_build("cloudbuild.yaml", source = my_gcs_source)

# helper that tars and adds to Source() for you
my_gcs_source2 <- cr_build_upload_gcs("my_folder")
build2 <- cr_build("cloudbuild.yaml", source = my_gcs_source2)

## End(Not run)
```

WebhookConfig — *WebhookConfig (Build Triggers)*

Description

WebhookConfig describes the configuration of a trigger that creates a build whenever a webhook is sent to a trigger's webhook URL.

Usage

```
WebhookConfig(secret, state = NULL)
```

Arguments

secret	Resource name for the secret required as a URL parameter.
state	Potential issues with the underlying Pub/Sub subscription configuration. Only populated on get requests.

Value

A WebhookConfig object

Index