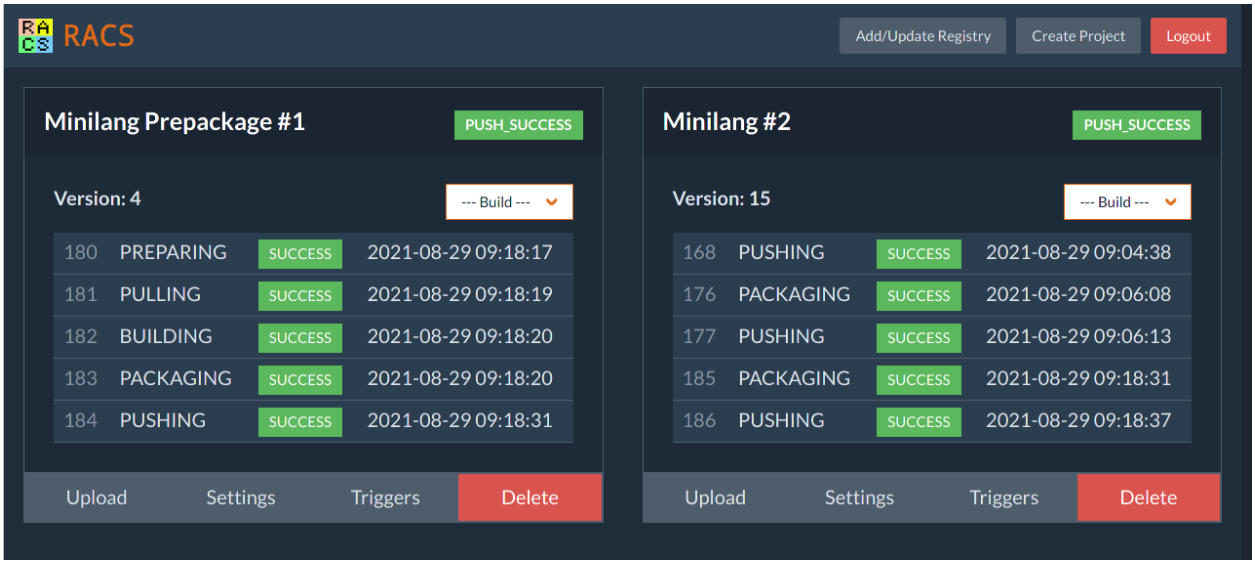
Racs

Raja Mukherji

Jul 29, 2023

CONTENTS:


RACS
Add/Update Registry
Create Project
Logout
Minilang Prepackage #1
PUSH_SUCCESS
Version: 4
--- Build ---
180 PREPARING SUCCESS 2021-08-29 09:18:17
181 PULLING SUCCESS 2021-08-29 09:18:19
182 BUILDING SUCCESS 2021-08-29 09:18:20
183 PACKAGING SUCCESS 2021-08-29 09:18:20
184 PUSHING SUCCESS 2021-08-29 09:18:31
Upload
Settings
Triggers
Delete
Minilang #2
PUSH_SUCCESS
Version: 15
--- Build ---
168 PUSHING SUCCESS 2021-08-29 09:04:38
176 PACKAGING SUCCESS 2021-08-29 09:06:08
177 PUSHING SUCCESS 2021-08-29 09:06:13
185 PACKAGING SUCCESS 2021-08-29 09:18:31
186 PUSHING SUCCESS 2021-08-29 09:18:37
Upload
Settings
Triggers
Delete

CHAPTER

ONE

OVERVIEW

`racs` (Raja's Attempt at a Continuous Something) is a simple tool for building and deploying OCI images (docker, podman, etc) from git repositories.

It is deliberately minimal in options, with a fixed set of build steps for all projects. Unlike many other continuous build tools, `racs` is designed for incremental builds with tools such as `make`, `gradle` and of course rabs.

CHAPTER

TWO

FEATURES

`racs` provides the following features:

- Simple web UI and API, including webhooks for pulling and building projects.
- Support for incremental builds. Every project is given its own working directory that persists between builds. The UI and API provide commands for emptying the working directory and performing a clean checkout if required.
- Projects can trigger other projects after each build. This allows for more complex project build workflows.
- Build scripts can be uploaded outside of the git repository. This means `racs` can work with existing projects without requiring any changes to their codebases.

CHAPTER

THREE

LIMITATIONS

`racs` has several limitations, some due to implementation time constraints and others intentional:

- Hard-coded to use `podman` for all image builds. It is expected that `racs` is running on its own server or container with a working `podman` available. This may become configurable in the future.
- Only supports PAM based authentication, authenticating against the local users. With `racs` running on its own server, this should be sufficient. This may become configurable in the future.
- Fixed build steps for all projects: *clean* → *clone* → *prepare* → *pull* → *build* → *pacakge* → *push*.

CHAPTER

FOUR

INSTALLATION

racs is written in go so assuming go is already installed on a machine, building racs is simply:

```
$ git clone https://github.com/wrapl/racs.git
$ cd racs
$ go build
```

The result racs executable should then be run in the desired directory:

```
$ cd /path/to/projects
$ /path/to/racs -port 8080 -ssl-cert ssl.crt -ssl-key ssl.key -no-login true
```

CHAPTER

FIVE

OPTIONS

-port <num>
Sets the port number for the web server, defaults to 8080.

-no-login
Allows users to perform all operations without logging in.

-ssl-cert
Uses HTTPS instead of HTTP, with the provided SSL cert file.

-ssl-key
The SSL key file to use.

5.1 Usage

5.1.1 Login / Logout

By default, `racs` requires users to login before performing certain operations. Users can login by clicking *LOGIN* in the top bar and entering their credentials. Currently `racs` uses PAM for authentication, effectively users are authenicated against the underlying operating system.

5.1.2 Projects Overview

Each buildable unit in `racs` is called a *project*. Each project is assigned an increment integer identifier, starting at 1. Projects are stored in the `/projects` directory with the following structure:

- projects
  - 1
    - context
    - workspace
      - source
  - 2
    - context
    - workspace
      - source
  - ...

Each project directory has the following contents:

/context
: is the context for the `prepare` and `package` stages.

/workspace
: is the working directory for the **build** and **package** stages.

/workspace/source
: is the cloned source directory.

Note: The `/workspace` directory for each project is preserved between builds and mounted automatically as `/workspace` during the **build** and **package** stages. This allows for builds to be incremental since build output is reused. The **clean** stage can be used to clear the `/workspace/source` directory.

5.1.3 Creating Projects

Projects can be created by clicking *CREATE PROJECT* in the top bar (logging in first if necessary). A dialog appears for entering the new project's details. Note that all the details can also be entered or changed later.

Name
: The name of the project, for display purposes only.

URL
: The URL of the git repository for the project.

Branch
: The git branch to clone / pull.

Destination
: *Optional* An OCI container registry to push the built image.

Tag
: *Optional* A template for the image tag when pushing to an OCI container registry.

The project tag can contain variables of the form `$NAME` which are substituted when an image is created:

`$VERSION`
: Replaced with the latest successful build version, incremented automatically, starting from 1.

After creating a project, at least 2 additional files need to be uploaded before the project can be built.

5.1.4 Project Uploads

Additional files can be uploaded to a project's directory. Users can open the project settings dialog by clicking the button and then switching to the *Upload* tab. Files can be uploaded to any path in the project's directory.

Container Spec Files

`racs` requires 2 OCI container spec files to be available somewhere in the project directory for creating the build image and package image for the project. These files can be located anywhere in the project directory and named anything but by default are expected to reside at `/BuildSpec` and `/PackageSpec` respectively. This allows `racs` to be used to build projects which do not contain the necessary container spec files in their repositories.

The paths of the build and package spec files can be changed using the project settings dialog by clicking the button and switching to the *Settings* tab. For projects that keep the build and package spec files within the git repository, these paths can be changed to something like `/workspace/source/BuildSpec` and `/workspace/source/PackageSpec`.

5.1.5 Build Stages

Every project has a fixed set of build stages. After each stage is complete, the next stage is automatically started. Users can manually restart the build process from a specific using the --*Build*-- dropdown for each project.

Clean
: Deletes the project's `/workspace/source` directory.

Clone
: Recursively clones the selected branch of the project's git repository into a directory called `/source`.

Prepare
: Builds the OCI container (using `BuildSpec`) that will be used for building / updating the project when required.

Pull
: Recursively pulls the latest changes from the git repository. This is the default starting point for each subsequent build after the initial build.

Build
: Runs the build image with the `/workspace` directory mounted. The build image's `ENTRYPOINT` should be the build command for the project.

Package
: Builds the OCI container (using `PackageSpec`) that will be tagged and pushed to the remote registry.

Push
: Pushes the package image to the remote registry. If no destination is specified for this project then this stage does nothing.

5.1.6 Project Version

Each time a project's package stage completes successfully, it's version is incremented. This can be used in the image tag when pushing to a container registry by using `$VERSION` in the project tag setting.

5.1.7 Triggers

Each time a project's push stage completes successfully, it can trigger other projects to start building from a specified stage. Triggers can be configured for a project by clicking the buttons an switching to the *Triggers* tab.

When triggered from another project, the additional environment variable `RACS_TRIGGER` is passed to the build stage with the triggering project's tag value.