redata

Release v0.2.0

Chun Ly, UA Research Data Repository (ReDATA) Team

Jul 13, 2021

CONTENTS:

CHAPTER

ONE

OVERVIEW

This ReadTheDocs landing page provides general documentation for software pertaining to ReDATA, the University of Arizona Research Data Repository.

The GitHub repository is available here.

All ReDATA-related repositories are under the GitHub organization (UAL-RE) of Research Engagement, University of Arizona Libraries

Unless indicated, all software are under an MIT License.

1.1 Repositories Overview

1.1.1 Repositories purposes

Our codebases fall in one of six categories:

1. Common/general software used throughout ReDATA codebases
2. Documentation
3. Identity and access management (IAM)
4. Data curation
5. Data preservation
6. Infrastructure as Code (IaC)

Software name	Category	Purpose
LD-Cool-P	Curation	Python command-line API for data curation
ReBACH	Preservation	Software to support data preservations with `Dart` and other tools
ReQUIAM	IAM	Python command-line API for IAM
ReQUIAM_csv	IAM	Python command-line API and database of groups for IAM
figshare	Curation, Preservation	A forked copy of cognoma's repository used to gather public/private data from Figshare API
ldcoolp-figshare	Curation	Python backend API for access to the Figshare API for Figshare for Institutions instances
redata-commons	General	A set of common modules, code, and external libraries used throughout ReDATA codebases.
redata-docs	Documentation	The repository hosting the current pages you are viewing on Read The Docs
redata-iac	IaC	Repository containing Infrastructure as Code (IaC) and scripts used on the operational side of ReDATA

1.1.2 Repositories details

More details about each repository:

Software name	Tag version	Changelog	Documentation	Main branch	PyPI
LD-Cool-P		CHANGELOG	README	`master`	TBD
ReBACH	N/A	TBC	README	`main`	TBD
ReQUIAM		README	README	`master`	N/A
ReQUIAM_csv		TBC	RTD	`master`	N/A
figshare		N/A	N/A	`master`	N/A
ldcoolp-figshare		CHANGELOG	RTD	`main`	ldcoolp-figshare
redata-commons		CHANGELOG	RTD	`main`	redata
redata-docs		N/A	RTD	`main`	N/A
redata-iac		TBC	N/A	`master`	N/A

1.1.3 Repositories status

Below summarizes open and closed issues and pull requests.

Software name	Open and closed issues	Pull requests
LD-Cool-P		
ReBACH		
ReQUIAM		
ReQUIAM_csv		
figshare	N/A	N/A
ldcoolp-figshare		
redata-commons		
redata-docs		
redata-iac		

1.2 Project Management

1.2.1 An Overview

For software development purposes, we utilize `git` and GitHub extensively for version control and project management. This is crucial since we must keep track of hundreds of bugs, improvements, and changes for several repositories.

We use GitHub tools to track and implement changes to the software. First, we use GitHub issues to identify and track bugs/issues/features, and GitHub pull requests or "PR" so that a developer can suggest a set of changes to be merged into the `master`/`main` branch. Within these issue and PR tracking, we use labels to indicate what these changes/problems pertain to. Each repository has a set of labels. Labels are helpful to understand scope and impact and aids in GitHub search engine optimization. To understand the scope of any work, we use GitHub milestone tracking. Finally, we use GitHub project boards to illustrate and manage issues and PRs. Each repository has its own project board. These are kanban style boards with several columns/lists.

1.2.2 DevOps workflow

The general workflow are as follow when starting any improvement:

1. Create a new GitHub issue if one does not exist. Begin tracking it in the project board
2. Create a new branch locally
3. Commit changes to branch and push them to the new branch on the remote repository (i.e. GitHub)
4. Create a PR within the repository to merge the new branch into the `master`/`main` branch
5. A team member reviews the PR (if enough developers are on staff). Self-review are OK if staff is limited.
6. The changes are merged into the `master`/`main` branch and any associated tags are pushed to the remote repository
7. The software is manually deployed

1.2.3 Branching

It is strongly recommended to use `git` branches for software development. This is because, at any point, multiple features/bugs are being addressed, and changes pushed directly to the main branch could break the software if it is *untested or has not been reviewed*. Branching is a common Developer + Operations ("DevOps") best practice. To create a new `git` branch, use the following `git` commands:

```
$ git pull master
$ git checkout -b <name_of_branch>
```

To checkout an existing branch:

```
$ git branch  # To see existing branches
$ git checkout <name_of_branch>
```

In terms of branch names, it is strongly recommended to name branches so it is clear and concise. We strongly recommend including:

1. The GitHub issue number
2. Whether it is a feature/enhancement or a bug fix
3. A short description

The above ensures an easier understanding to the software development team. Examples include:

1. `feature/235_preserve_prep` for LD-Cool-P#235
2. `hotfix/229_400_error` for LD-Cool-P#229

Note: Our branching model initially followed a `git-flow` workflow with features, hotfixes, and releases; however, we later moved away from that model and now use a GitHub flow workflow where all changes are merged into the `master/main` branch after review and testing.

1.2.4 Versioning and tagging

In all of our software, we conduct version tagging. Here, each new version refers to a change to the codebase that is to be deployed. We loosely follow Semantic versioning (SemVer), which denotes changes as MAJOR, MINOR, and PATCH. There are two differences with our method of versioning against SemVer:

1. We use the patch denotation for both hotfixes and small enhancements to software.
2. We use MINOR denotation for large/larger enhancements (e.g. a completely new feature rather than an improvement to an existing feature).

MAJOR remains the same, for incompatible API changes. We try to avoid the latter as much as possible.

While some open-source software teams may not use version tagging, there are many advantages. First, this step ensures that we have continuous delivery of our software. Second, for some of our software, we automatically deploy them on PyPI, a `python` package manager that allows for easy installation of the software. Finally, our logging tools records version information for each software, so this allows the team to trace an issue back to a specific PR. To tag a specific commit:

```
$ git tag vX.Y.Z -m
```

A `vim` prompt will appear so you can provide a message for the tag. Often a short message referring to the GitHub issue number will suffice. You will then push the tag via:

```
$ git push --tags
```

1.2.5 Merging code

TBD on using `git` over GitHub merge tool.

1.2.6 Milestone tracking

More details needed here.

1.2.7 Status of GitHub repositories

See *Repositories status*

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search