redata

Release v0.4

Yan Han, UA Research Data Repository (ReDATA) Team

Jun 14, 2022

CONTENTS:

CHAPTER

ONE

OVERVIEW

This ReadTheDocs landing page provides general documentation for software pertaining to ReDATA, the University of Arizona Research Data Repository. ReDATA is a Figshare for Institution instance that is managed by Figshare, our third-party Software-as-a-Service (SaaS) vendor.

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 programs and CSV files for ReQUIAM
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		CHANGELOG	RTD	`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

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 `main`/`master` 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 `main`/`master` 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 `main`/`master` branch and any associated tags are pushed to the remote repository
7. The software is manually deployed

Note: The default branch name is set to "main" starting Oct 1, 2020, not "master" anymore. For more info: please see Github rename master to main . Certain repositories still have default branch as "master".

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: (`-b` is to create a new branch)

```
$ git pull origin main
$ git checkout -b <new-branch>
```

To checkout an existing branch:

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

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
3. `chore/242_gitignore` for LD-Cool-P#242

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 `main`/`master` branch after review and testing.

1.2.4 Pushing to a remote branch

After updating files, we can push the changes to remote branch. It is important to push to a branch (not `main`) so that a team member can review the changes over a pull request. use the following `git` commands:

```
$ git add .
$ git commit -m "<message>"
$ git branch                    # list all the branches and * is the current branch
$ git push origin <branchname>  # push to a remote branch
```

In accordance with git's best practices, the commit message should be short but descriptive. Avoid general messages like "updated file.txt" when possible.

1.2.5 Versioning and tagging

Before creating a new tag, we need to make sure all related files updated to reflect the new tag. These files shall be checked and updated if existing:

1. Update __init__.py __version__ number or related configuration
2. Update setup.py variables such as version.
3. Review README.md and update related sentences.
4. Update CHANGELOG.md by adding changelog message

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
```

1.2.6 Merging code

Direct merges to main/master branches are to be avoided. When working collaboratively, all changes must be made to a branch and a pull request opened. The pull request must be reviewed and approved by another team member before being merged to the main/master branch.

1.2.7 Milestone tracking

More details needed here.

1.2.8 Status of GitHub repositories

See *Repositories status*

1.3 Identity and Access Management

1.3.1 IAM Overview

Our Figshare for Institution instance, has a couple of features to maintain identity and access management (IAM) settings and to assist in data repository administration.

First, we have the ability to set a quota of available space for each user. Our default quotas, applicable to most ReDATA users, are:

Classification	Quota
Undergraduates	0 (initially), 100MB after they contact us
Graduates	0.5 GB
Faculty/Staff/DCC	2 GB

Second, we have the ability to assign each users to groups on Figshare (a.k.a. "portals"). This allows for the easily exploration of data through these portals. For our deployment we chose to do it by following common research themes for our University. To identify researcher's discipline, we utilize their primary affiliation at the University.

1.3.2 Software/Services Overview

There are a number of software and services that we use for IAM. They are:

Software/Services	Maintainer(s)	Purpose
Enterprise Directory Service (EDS)	UITS	UArizona's LDAP directory used to gather metadata about their users from a central UA datastore in order to make authorization decisions.
Grouper	UITS	UArizona's tool to create groups for UA organization. This is populated into EDS and Shibboleth
Shibboleth / WebAuth	UITS	UArizona's SAML-based access to UA IAM information
ReQUIAM	ReDATA team	Python command-line API for IAM
ReQUIAM_csv	ReDATA team	Python command-line API and database of groups for IAM

1.3.3 Services

First, we utilize three services provided and administered by University Information Technology Services (UITS):

1. EDS
2. Shibboleth
3. Grouper

Users who login to ReDATA uses their NetID credentials to login (WebAuth). A user who is no longer part of the University will not have NetID and thus will not be able to log in.

1.3.4 Software

The two codebases that the ReDATA team develops and maintains are ReQUIAM and ReQUIAM_csv. The former is the primary software that manages all ReDATA IAM with a daily "cronjob" that sets research theme association ("portals") and quotas through the Grouper API. That information is then propagated into EDS and Shibboleth with users logging in. Also, `ReQUIAM` has a command-line API to enable other manual IAM changes for the ReDATA team, such as setting a higher quota from default quota settings (See *IAM Overview*)

The `ReQUIAM_csv` software contains the mapping between the groups on ReDATA's Figshare for Institution instance and UArizona organizational codes. The spreadsheet is available through Google Docs.

The Grouper-to-Figshare-group mapping is provided as a CSV file to be consumed by `ReQUIAM`, which are publicly available on GitHub at:

1. Raw version
2. Rendered version

1.3.5 Grouper settings

To control IAM, we update Grouper group memberships, which are metadata that is passed into EDS and ultimately Shibboleth and consumed by our Figshare for Institution instance for account creation (for first login) and update when users re-login. This metadata record is called `ismemberof`.

The three `ismemberof` settings that ensures proper IAM are:

`ismemberof`	Type	Purpose
active	Group	This enable login to ReDATA. Non-membership means the individual is no longer an active member by Libraries privileges
portal	Stem	Folder containing various research themes Grouper groups
quota	Stem	Folder containing Grouper groups of quotas in bytes

The Grouper stem prefix for the above is `arizona.edu:Dept:LBRY:figshare`.

`ReQUIAM` maintains direct membership for `portal` and `quota` groups. For the `active` group, this is done using indirect membership from other Grouper groups set by the University Libraries patron software, patron-groups.

Our Figshare instance maps the `portal` and `quota` settings accordingly such that:

1. A quota is set to ensure that a user has enough space for small deposits, which is most often the case. The user can request more space, which a ReDATA administrator would need to approve. The latter allows for the ReDATA team to understand the user's needs and to identify cases where there are large deposits requiring more assistance.
2. A researcher's data deposits are placed in a proper Figshare group/portal.

If a user does not have a `portal` set then their data publication will not appear in any group/portal, but part of the University wide group. If a quota is not set (for undergraduates logging in for the first time), then the quota is set to zero.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search