Knowledge Repo Documentation

Release v0.8.1

Airbnb and contributors

Jul 08, 2018

Contents

CHAPTER 1

Installation

To install the knowledge repository tooling (and all its dependencies), simply run:

```
$ pip install --upgrade "knowledge-repo[all]"
```

You can also skip installing dependencies which are only required in special cases by replacing *all* with one or more of the following (separated by commas): - *ipynb* : Installs the dependencies required for adding/converting Jupyter notebook files - *pdf* : Installs the dependencies required for uploading PDFs using the web editor - *dev*: Installs the dependencies required for doing development, including running the tests

In addition to the *knowledge_repo* Python library, this also installs a *knowledge_repo* script that is used to interact with the Knowledge Repository on the command line. You can verify that everything is set up appropriately by running:

```
$ knowledge_repo --version
```

This should should run and show the current version of the Knowledge Repo. It may then throw an exception due to a Knowledge Repository not being specified, but this is expected. If your shell environment cannot find the *knowledge_repo* executable, you may need to check your *$PATH* and ensure that your Python's executable "bin" folder is on it.

If you are a user expecting to interact with an existing Knowledge Repo, please refer to *Quickstart*. Otherwise, if you are looking to create a new Knowledge Repo installation, please refer to *Deployment*.

CHAPTER 2

Quickstart

This section is targetted toward users who are setting their computers up to work with an existing Knowledge Repo installation, and guides a user through their first knowledge post submission. If you are looking to create a new Knowledge Repo repository or server, please refer instead to *Deployment*.

2.1 Submitting your first Knowledge Post

There are several ways to create a Knowledge Post, of which what follows is just one way. For more workflows, refer to: *Writing*.

Step 1: Install the knowledge_repo tooling

If you have not already done so, follow the installation instructions: *Installation*.

Step 2: Clone the knowledge repository locally [Git repositories only]

If the repository with which you are going to be contributing is a git repository, clone it locally onto your machine.
Note: if you have direct access to a database knowledge repository, this step is not required.

```
$ git clone <git url> <local path>
```

Note: If you are just testing the workflow, you can create a test git repository using:

```
$ knowledge_repo --repo ./test_repo init
```

Note: If you will be primarily using a single knowledge repository, it is possible to avoid passing it to the *knowledge_repo* command every time by setting the `KNOWLEDGE_REPO` environment variable. For example:

```
export KNOWLEDGE_REPO=<repository uri/path>
```

For this to be persistent accross sessions, add it to your shell initialization script. If `KNOWLEDGE_REPO` is set, and points to the knowledge repository with which you would like to interact, you can drop the `--repo` options in the following.

Step 3: Create a Knowledge Post template

Knowledge Post templates are sample files in their original format which you can use to avoid having to remember how metadata is stored and/or added to the underlying post documents.

For a Jupyter notebook template:

```
$ knowledge_repo --repo <path/uri_of_repo> create ipynb example_post.ipynb
```

For an R Markdown template:

```
$ knowledge_repo --repo <path/uri_of_repo> create Rmd example_post.Rmd
```

Other templates may be available. You can see all available templates here.

Step 4: Edit the template as normal

Create whatever code cells / plots / LaTeX / etc that you normally would in your work.

Step 5: Add your post to the knowledge repository

```
$ knowledge_repo --repo <path/uri_of_repo> add example_post.ipynb -p project/
↪example_ipynb
$ knowledge_repo --repo <path/uri_of_repo> add example_post.Rmd -p project/example_
↪rmd
```

Note that the *-p* option specifies the path in the repository to which the post should be added.

Step 6: Preview the added post

Sometimes formatting may differ in the Knowledge Web Application compared to that shown in your native environment. Checking that the rendering is what you expect is a good idea before submitting it for peer review.

```
$ knowledge_repo --repo <path/uri_of_repo> preview <post_path_in_repository>
```

Step 7: Submit post for review

If everything looks good when previewed, the final step of post submission is submitting your local posts upstream for review, and ultimately, publishing.

```
$ knowledge_repo --repo <path/uri_of_repo> submit <post_path_in_repository>
```

CHAPTER 3

Deployment

Deploying the Knowledge Repo in an organization is done in two steps.

1. A knowledge repository (or repositories) must be created. These repositories are where knowledge posts are pooled and made accessible to users.
2. Deploy the Knowledge Repo web application on top of this repository (or repositories), which then acts as the primary gateway for users to access the stored knowledge posts.

3.1 Creating Knowledge Repositories

The Knowledge Repo project supports multiple repository backends, all offering the same programmatic API (and being subclasses of *KnowledgeRepository*). At this stage, two backends have been fully implemented: - *GitKnowledgeRepository*: A repository backed by a local git repository (optionally synced with remote git repository). - *DbKnowledgeRepository*: A repository backed by a database backend (most databases supported by SQLAlchemy can be used).

All backends also allow configuration using a YAML configuration file at '/.knowledge_repo_config.yml' within the repository. A template for creating this file is available here, if one does not already exist or the repository configuration has grown out of sync with upstream changes.

3.1.1 Git Knowledge Repositories

The following command will create a new repository at <*repo_path*>:

```
$ knowledge_repo --repo <repo_path> init
```

The result is a git repository at <*repo_path*> with a *.knowledge_repo_config* copied from the defaults found in the repository source code. If a git repository was already found at <*repo_path*> it will upgrade it to be a knowledge data repository. This is useful if you are starting a repository on a remote service like GitHub, as this allows you to clone the remote repository as per normal; run this script; and then push the initialization back into the remote service using *git push*. Otherwise, if you plan to host this repository on a remote service like GitHub, you must push this repository

to that remote service. This can be done by creating the remote repository through GitHub (or whichever service you plan to use) and then running:

```
$ git remote add origin <url_of_remote_repository>
$ git push -u origin master
```

Users can then clone this repository, and point their local *knowledge_repo* script at it using `--repo <path_of_cloned_repository>`.

3.1.2 Database Knowledge Repositories

Database knowledge repositories are the only backends that currently allow end-to-end publishing of knowledge posts via the web app interface. They are created on demand, where possible. Simply point the *knowledge_repo* script at it using something akin to `--repo mysql://username:password@hostname/database:table_name`. If the table does not exist it will be created if the active user has the appropriate permissions.

Note: Database Knowledge repositories also support having a *.knowledge_repo_config* configuration, but one is not automatically added.

3.1.3 Repository Configuration

As noted earlier, all knowledge repository backends support configuration via a Python file that is imported from the repository. This configuration can override the defaults in the default repository configuration found here.

This configuration file will allow you to:

- Add postprocessors to post contributions from the repo. (see the *postprocessors* array of functions)
- Add rules for which subdirectories posts can be added to. (see the *path_parse()* function)
- **Check and manage the format of author names at contribution time**
  - Add logic to *username_parse()* to check post author names and raise exceptions when they don't match
  - Add logic to *username_to_name()* to manage how user/author names are displayed, ex. "sally_smarts" –> "Sally Smarts"
  - Add logic to *username_to_email()* to manage how user/author names are matched to emails, ex. "sally_smarts" –> "sally.smarts@mycompany.com"

Please refer to the default configuration file itself for further documentation.

Note: Image handling is a good example of where post-processor configuration can be very useful. Knowledge repositories' default behavior is to add the markdown's contents as is to your knowledge post git repository, including images. If you do not have git LFS set up, it may be in your interest to have these images hosted on some type of cloud storage, so that cloning the git repository locally is less cumbersome.

We provide an example postprocessor that adds support for pushing images to cloud storage. To use it, simply import or paste it into your *.knowledge_repo_config* file, and add it by name to your *postprocessors* configuration key.

3.2 Deploying the Web Application

Any user with access to knowledge repositories can create an instance of the Knowledge Repo Web Application that acts as a portal to them. This is achieved by running:

```
$ knowledge_repo --repo <repo_path> runserver
```

which starts a web application instance on *http://127.0.0.1:7000* with the default (insecure) options. The command line also supports some high-level options, such as *–port* and *–dburi* which respectively change the local port on which the server is running, and the sqlalchemy uri where the database can be found and/or initiated.

For shared deployments, however, you will probably need to create a server configuration file. A complete server configuration template can be found here. The configuration file gives you fine-grained control over the deployment, including authentication, access policies, indexing behavior.

Once a configuration file has been created according to the documentation provided in the template, deploying the web application is as simple as:

```
$ knowledge_repo --repo <repo_path> deploy --config <config_file>
```

Supported options are *–port*, *–dburi*,'–workers', *–timeout* and *–config*. The *–config* option allows you to specify a python config file from which to load the extended configuration. A template config file is provided in *knowledge_repo/app/config_defaults.py*. The *–port* and *–dburi* options are as before, with the *–workers* and *–timeout* options specifying the number of threads to use when serving through gunicorn, and the timeout after which the threads are presumed to have died, and will be restarted.

3.2.1 Database Migrations

No matter which knowledge repository backends are used, the web application itself requires a database backend in order to store its cache of the post index and user permissions. The database to be used can be specified via the CLI using the `--dburi` option or via the config file passed in using `--config`. Most datatabase backends supported by SQLAlchemy should work. Database URIs will look something like: `mysql://username:password@hostname/database:table_name`.

If the database does not exist, it is created (if that is possible) and initialised. When updates to the Knowledge Repo require changes to the database structure, migrations are automatically performed (unless disabled in the config to prevent accidental data loss). They can also be performed manually using:

```
$ knowledge_repo --repo <repo_path> db_upgrade --dburi <db>
```

3.2.2 Multiple Repositories

Multiple repositories can be stitched together into a single knowledge repository and served via a single web application instance. This is achieved using a *MetaKnowledgeRepository* instance, which creates a virtual filesystem into which the knowledge repositories are "mounted".

For example, you can mount a git repostory at */* and a database repository at */webposts* using:

```
$ knowledge_repo --repo {}/path/to/git/repo --repo {webposts}<db_uri>:<table> ...
```

3.2.3 Web Editor

The web editor allows the entire post creation and publication process to be done through the web application. To enable the web editor, simply add the path(s) under which web edited posts are allowed to be created to the *WEB_EDITOR_PREFIXES* option in the server configuration. Note that these paths **must** be backed by a database repository.

CHAPTER 4

Workflows

4.1 Browsing

While the web app interface is designed to be intuitive, it would be great to have some documentation. Contributions are welcome!

> TODO: Add the following? Make sure it is correct.
>
> Tags are one of the main organizing principles of the knowledge repo web app, and it is important to have high integrity on these tag names. For example, having one post tagged as "my_project" and another as "my-project" or "my_projects" prevents these posts from being organized together.
>
> We currently have two ways to maintain tag integrity:
>
> - Browse the */cluster?group_by=tags* endpoint to find and match existing tags.
> - After contributing, organize tags in batch with the */batch_tags* end point.

4.2 Writing

4.2.1 TLDR Reference

If you have successfully contributed to a knowledge repository before, this section is probably for you. It provides a quick reference for the commands associated with each step of a knowledge post submission for the most common scenario (i.e. where the repository is backed by a Git repository). If you want more detail, you can find it in the sections below.

In this quick reference, Jupyter / IPython notebooks are used as an example. Other file types, such R Markdown work identically, simply substituting 'ipynb' for 'Rmd', etc. It is also assumed that you have configured the KNOWLEDGE_REPO environment variable to point to a locally accessible uri for the knowledge repository.

1. *knowledge_repo create ipynb ~/Documents/my_post.ipynb*: creates a template with required yaml headers. Templates can also be downloaded by clicking "Write a Post!" the web application. *Make sure your post has these headers with correct values for your post*

2. Do your work in the generated my_post.Rmd file. *Make sure the post runs through from start to finish before attempting to add to the Knowledge Repo!*
3. *knowledge_repo add ~/Documents/my_post.Rmd [-p projects/test_project] [–update]*
4. *knowledge_repo preview projects/test_project*
5. *knowledge_repo submit projects/test_project*
6. From your remote git hosting service, request a review for merging the post. (ie. open a pull request on Github)
7. After it has been reviewed, merge it in to the master branch.

4.2.2 Full Guide

Creating

Once the knowledge data repository has been initialized, it is possible to start adding posts. Each post in the knowledge repository requires a specific header format, used for metadata formatting. To create a new post using a provided template, which has both the header information and example content, run the following command:

```
$ knowledge_repo --repo <repo_path> create {ipynb, Rmd, md} filename
```

The first argument indicates the type of the file that you want created, while the second argument indicates where the file should be created.

If the knowledge data repository is created at *knowledge_data_repo*, running

```
$ knowledge_repo --repo knowledge_data_repo create md ~/Documents/my_first_knowledge_
↪post.md
```

will create a file, *~/Documents/my_first_knowledge_post.md*, the contents of which will be the boilerplate template of the knowledge post.

The help menu for this command (and all following commands) can be reached by adding the *-h* flag, *knowledge_repo –repo <repo_path> create -h*.

Alternatively, by going to the */create* route in the webapp, you can click the button for whichever template you would like to have, and that will download the correct template.

It is also possible to manually add headers to an existing document (if using Jupyter Notebooks, add these headers to a "raw" cell at the top of your notebook). An valid example header looks something like:

```
---
title: I Found that Lemurs Do Funny Dances
authors:
- sally_smarts
- wesley_wisdom
tags:
- knowledge
- example
created_at: 2016-06-29
updated_at: 2016-06-30
tldr: This is short description of the content and findings of the post.
---
```

For a complete set of valid headers, refer to the header reference table below.

Adding

Once you've finished writing a post, the next step is to add it to the knowledge data repository. To do this, run the following command:

```
$ knowledge_repo --repo <repo_path> add <file with format {ipynb, Rmd, md}> [-p
↪<location in knowledge repo>]
```

Using the example from above, if we wanted to add the post *~/Documents/my_first_knowledge_post.md* to *knowledge_data_repo*, we would run:

```
$ knowledge_repo --repo knowledge_data_repo add ~/Documents/my_first_knowledge_post.
↪md -p projects/test_knowledge
```

The *-p* flag specifies the location of the post in the knowledge data repository - in this case, *knowledge_data_repo/projects/test_knowledge*. The *-p* flag does not need to be specified if *path* is included in the header of the knowledge post.

Updating

To update an existing knowledge post, pass the *–update* flag to the *add* command. This will allow the add operation to override exiting knowledge posts.

```
$ knowledge_repo --repo <repo_path> add --update <file with format {ipynb, Rmd, md}>
↪<location in knowledge repo>
```

Previewing

If you would like to see how the post would render on the web app before submitting the post for review, run the following command:

```
$ knowledge_repo --repo <repo_path> preview <path of knowledge post to preview>
```

In the case from above, we would run:

```
$ knowledge_repo --repo knowledge_data_repo preview projects/test_knowledge
```

There are other arguments that can be passed to this command, adding the *-h* flag shows them all along with further information about them.

Submitting

After running the add command, two things should have happened: 1. A new folder should have been created at the path specified in the add command, which ends in *.kp*. This is added automatically to indicate that the folder is a knowledge post. 2. This folder will have been committed to the repository on the branch named *<repo_path>/path_in_add_command*

Running the example command: `knowledge_repo --repo knowledge_data_repo add ~/Documents/my_first_knowledge_post.md -p projects/test_knowledge`, we would have seen:

1. A new folder: *knowledge_data_repo/projects/test_knowledge.kp* which was committed on
2. A branch (that you are now on), called *knowledge_data_repo/projects/test_knowledge*

To submit this post for review, simply run the command:

```
$ knowledge_repo --repo <repo_path> submit <the path of the knowledge post>
```

In this case, we would run:

```
$ knowledge_repo --repo knowledge_data_repo submit knowledge_data_repo/projects/test_
↪knowledge.kp
```

Post Headers Reference

The complete list of supported headers, their purpose and an example is provided below.

header	required	purpose	example
title	required	String at top of post	title: This post proves that 2+2=4 |
authors	required	User entity that wrote the post in organization specified format	authors: - kanye_west - beyonce_knowles |
tags	required	Topics, projects, or any other uniting principle across posts	tags: - hiphop - yeezy |
created_at	required	Date when post was written	created_at: 2016-04-03 |
updated_at	optional	Date when post was last updated	created_at: 2016-10-10 |
tldr	required	Summary of post takeaways that will be visible in /feed	tldr: I'ma let you finish, but Beyonce had one of the best videos of all time! |
path	optional	Instead of specifying post path in the CLI, specify with this post header	path: projects/path/to/post/on/repo |
thumbnail	optional	Specify which image is shown in /feed	thumbnail: 3 OR thumbnail: http://cdn.pcwallart.com/images/giraffe-tongue-wallpaper-1.jpg |
private	optional	If included, post is only visible to authors and editors set in repo configuration	private: true |
allowed_groups	optional	If the post is private, specify additional users or groups who can see the post	allowed_groups: ['jay_z', 'taylor_swift', 'rap_community'] |

4.3 Programmatic Interface

It is possible to use the *knowledge_repo* python library to programmatically interact with knowledge repositories in Python. It would be helpful to have a few examples of this. Contributions welcome!

CHAPTER 5

Under the Hood

This section of the documentation describes the more techical aspects of the way in which the Knowledge Repo is implemented. At present, the details provided here are relatively basic. Eventually, full API documentation should be present here. Contributions welcome!

5.1 Technical Details

5.1.1 What is a Knowledge Repository?

A knowledge repository is a virtual filesystem (such as a git repository or database). A GitKnowledgeRepository, for example, has the following structure:

```
<repo>
    + .git  # The git repository metadata
    + .resources  # A folder into which the knowledge_repo repository is checked out␣
↪(as a git submodule)
    - .knowledge_repo_config.py  # Local configuration for this knowledge repository
    - <knowledge posts>
```

The use of a git submodule to checkout the knowledge_repo into *.resources* allows use to ensure that the client and server are using the same version of the code. When one uses the *knowledge_repo* script, it actually passes the options to the version of the *knowledge_repo* script in *.resources/scripts/knowledge_repo*. Thus, updating the version of knowledge_repo used by client and server alike is as simple as changing which revision is checked out by git submodule in the usual way. That is:

```
$ pushd .resources
$ git pull
$ git checkout <revision>/<branch>
$ popd
$ git commit -a -m 'Updated version of the knowledge_repo'
$ git push
```

Then, all users and servers associated with this repository will be updated to the new version. This prevents version mismatches between client and server, and all users of the repository.

In development, it is often useful to disable this chaining. To use the local code instead of the code in the checked out knowledge repository, pass the `--dev` option as:

```
$ knowledge_repo --repo <repo_path> --dev <action> ...
```

5.1.2 What is a Knowledge Post?

A knowledge post is a directory, with the following structure:

```
<knowledge_post>
        - knowledge.md
        + images/* # [Optional]
        + src/* # [Optional; stores the original source file(s)]
```

Images are automatically extracted from the local paths on your computer, and placed into images. *src* contains the file(s) from which the knowledge post was converted from (which may include additional files depending on which files were specified at creation time by the user).

CHAPTER 6

Contributing

We welcome all manner of contributions, including bug reports, feature suggestions, documentation improvements, and patches to support new or improved functionality.

For developers looking to extend the Knowledge Repo, a few specific common examples are provided below.

6.1 Adding support for new Knowledge Post conversions

Support for conversion of a particular filetype to a knowledge post is added by writing a new *KnowledgePostConverter* object. Each converter should live in its own file in *knowledge_repo/converters*. Refer to the implementation for ipynb, Rmd, and md for more details. If your conversion is site-specific, you can define these subclasses in *.knowledge_repo_config*, whereupon they will be picked up by the conversion code.

6.2 Adding extra structure and/or verifications to the knowledge post conversion process

When a KnowledgePost is constructed by converting from support filetypes, the resulting post is then passed through a series of postprocessors (defined in *knowledge_repo/postprocessors*). This allows one to modify the knowledge post, upload images to remote storage facilities (such as S3), and/or verify some additional structure of the knowledge posts. As above, defining or importing these classes in *.knowledge_repo_config.py* allows for postprocessors to be used locally.

6.3 Something else?

Please don't hesitate to file an issue on our GitHub issue tracker, and we will get back to you when we can.

The Knowledge Repo is a set of tools that manage a curated collection of "Knowledge Posts" along with a web interface to view and disseminate them. Knowledge Posts are representations of technical and non-technical documents

that include document metadata, original source files, and a rendered Markdown file generated from them. The Knowledge Repo includes tooling to create Knowledge Posts from a variety of commonly used formats, such as Jupyter and R Markdown notebooks.

It was created at Airbnb to solve the problem of scaling technical knowledge dissemination within a growing team of data scientists, as sharing progress via one-off emails, documents, slideshows and git repositories was becoming increasingly impractical. It has since grown to be an indispensable method by which knowledge is shared not just between data scientists, but also with the rest of the company. We invite you to join us in deploying and using the Knowledge Repository in your workplace.

For more information about the motivation and inspiration behind this project, we encourage you to read our Medium Post.

CHAPTER 7

Feedback

The Knowledge Repo is a reasonably complete internal publishing service, and we hope that you find it useful. If you have difficulties deploying the Knowledge Repo in your organisation, we would love to hear from you; particularly as it pertains to the following:

- How easy is it to set up the git knowledge post repository?
- How easy is it to set up the web application, and make it live internally within your organization?
- Where are the gaps in our documentation that we should fill in to assist others in understanding the system?
- At a higher level, are there any blockers or barriers to setting up the Knowledge Repo in your organization?