bookstore Documentation

Release 2.2.1

nteract project

Jun 08, 2019

Contents

Release v2.2.1 (*What's new?*).

bookstore provides tooling and workflow recommendations for storing, scheduling, and publishing notebooks.

CHAPTER 1

Table of Contents

1.1 Installation

bookstore may be installed using Python 3.6 and above.

After installation, bookstore can process Python 2 or Python 3 notebooks.

1.1.1 Install from PyPI (recommended)

```
python3 -m pip install bookstore
```

1.1.2 Install from Source

1. Clone this repo:

   ```
   git clone https://github.com/nteract/bookstore.git
   ```

2. Change directory to repo root:

   ```
   cd bookstore
   ```

3. Install dependencies:

   ```
   python3 -m pip install -r requirements.txt
   python3 -m pip install -r requirements-dev.txt
   ```

4. Install package from source:

   ```
   python3 -m pip install .
   ```

Tip: Don't forget the dot at the end of the command

1.2 Configuration

Commonly used configuration settings can be stored in `BookstoreSettings` in the `jupyter_notebook_config.py` file. These settings include:

- workspace location
- published storage location
- S3 bucket information
- AWS credentials for S3

1.2.1 Example configuration

Here's an example of `BookstoreSettings` in the `~/.jupyter/jupyter_notebook_config.py` file:

```
"""jupyter notebook configuration
The location for user installs on MacOS is ``~/.jupyter/jupyter_notebook_config.py``.
See https://jupyter.readthedocs.io/en/latest/projects/jupyter-directories.html for␣
↪additional locations.
"""
from bookstore import BookstoreContentsArchiver


c.NotebookApp.contents_manager_class = BookstoreContentsArchiver

c.BookstoreSettings.workspace_prefix = "/workspace/kylek/notebooks"
c.BookstoreSettings.published_prefix = "/published/kylek/notebooks"

c.BookstoreSettings.s3_bucket = "<bucket-name>"

# If bookstore uses an EC2 instance with a valid IAM role, there is no need to␣
↪specify here
c.BookstoreSettings.s3_access_key_id = <AWS Access Key ID / IAM Access Key ID>
c.BookstoreSettings.s3_secret_access_key = <AWS Secret Access Key / IAM Secret Access␣
↪Key>
```

The root directory of bookstore's GitHub repo contains an example config called `jupyter_config.py.example` that shows how to configure `BookstoreSettings`.

1.3 Usage

Data scientists and notebook users may develop locally on their system or save their notebooks to off-site or cloud storage. Additionally, they will often create a notebook and then over time make changes and update it. As they work, it's helpful to be able to **store versions** of a notebook. When making changes to the content and calculations over time, a data scientist using Bookstore can now request different versions from the remote storage, such as S3, and **clone** the notebook to their local system.

Note: store and clone

store

User saves to Local System ————————> Remote Data Store (i.e. S3)

clone

User requests a notebook to use locally <———————— Remote Data Store (i.e. S3)

After some time working with a notebook, the data scientist may want to save or share a polished notebook version with others. By **publishing a notebook**, the data scientist can display and share work that others can use at a later time.

1.3.1 How to store and clone versions

Bookstore uses automatic notebook version management and specific storage paths when storing a notebook.

Automatic notebook version management

Every *save* of a notebook creates an *immutable copy* of the notebook on object storage. Initially, Bookstore supports S3 for object storage.

To simplify implementation and management of versions, we currently rely on S3 as the object store using versioned buckets. When a notebook is saved, it overwrites the existing file in place using the versioned s3 buckets to handle the versioning.

Storage paths

All notebooks are archived to a single versioned S3 bucket using specific **prefixes** to denote a user's workspace and an organization's publication of a user's notebook. This captures the lifecycle of the notebook on storage. To do this, bookstore allows users to set workspace and published storage paths. For example:

- `/workspace` - where users edit and store notebooks
- `/published` - notebooks to be shared to an organization

Bookstore archives notebook versions by keeping the path intact (until a user changes them). For example, the prefixes that could be associated with storage types:

- Notebook in "draft" form: `/workspace/kylek/notebooks/mine.ipynb`
- Most recent published copy of a notebook: `/published/kylek/notebooks/mine.ipynb`

Note: *Scheduling (Planned for a future release)*

When scheduling execution of notebooks, each notebook path is a namespace that an external service can access. This helps when working with parameterized notebooks, such as with Papermill. Scheduled notebooks may also be referred to by the notebook `key`. In addition, Bookstore can find version IDs as well.

Easing the transition to Bookstore's storage plan

Since many people use a regular filesystem, we'll start with writing to the `/workspace` prefix as Archival Storage (more specifically, writing on save using a `post_save_hook` for the Jupyter contents manager).

1.3.2 How to publish a notebook

To publish a notebook, Bookstore uses a publishing endpoint which is a `serverextension` to the classic Jupyter server. If you wish to publish notebooks, explicitly enable bookstore as a server extension to use the endpoint. By default, publishing is not enabled.

To enable the extension globally, run:

```
jupyter serverextension enable --py bookstore
```

If you wish to enable it only for your current environment, run:

```
jupyter serverextension enable --py bookstore --sys-prefix
```

1.4 REST API

GET /api/bookstore
Info about bookstore

Status Codes

- 200 OK – Successfully requested

GET /api/bookstore/cloned
Landing page for initiating cloning.

This serves a simple html page that allows avoiding xsrf issues on a jupyter server.

Query Parameters

- **s3_bucket** (*string*) – S3_bucket being targeted
- **s3_key** (*string*) – S3 object key being requested

Status Codes

- 200 OK – successful operation
- 400 Bad Request – Must have a key to clone from

POST /api/bookstore/cloned
Trigger clone from s3

Status Codes

- 200 OK – Successfully cloned
- 400 Bad Request – Must have a key to clone from

PUT /api/bookstore/published/{path}
Publish a notebook to s3

Parameters

- **path** (*string*) – Path to publish to, it will be prefixed by the preconfigured published bucket.

Status Codes

- 200 OK – Successfully published.

1.5 Reference

1.5.1 Configuration

Bookstore may be configured by providing `BookstoreSettings` in the `~/.jupyter/jupyter_notebook_config.py` file.

bookstore.bookstore_config

Configuration settings for bookstore.

class bookstore.bookstore_config.**BookstoreSettings**(***kwargs*)

Configuration for archival and publishing.

Settings include storage directory locations, S3 authentication, additional S3 settings, and Bookstore resources.

S3 authentication settings can be set, or they can be left unset when IAM is used.

Like the Jupyter notebook, bookstore uses traitlets to handle configuration, loading from files or CLI.

workspace_prefix

Directory to use for user workspace storage

Type str(`workspace`)

published_prefix

Directory to use for published notebook storage

Type str(`published`)

s3_access_key_id

Environment variable `JPYNB_S3_ACCESS_KEY_ID`

Type str, optional

s3_secret_access_key

Environment variable `JPYNB_S3_SECRET_ACCESS_KEY`

Type str, optional

s3_endpoint_url

Environment variable `JPYNB_S3_ENDPOINT_URL`

Type str(`"https://s3.amazonaws.com"`)

s3_region_name

Environment variable `JPYNB_S3_REGION_NAME`

Type str(`"us-east-1"`)

s3_bucket

Bucket name, environment variable `JPYNB_S3_BUCKET`

Type str(`""`)

max_threads

Maximum threads from the threadpool available for S3 read/writes

Type int(`16`)

bookstore.bookstore_config.**validate_bookstore**(*settings: bookstore.bookstore_config.BookstoreSettings*)

Check that settings exist.

Parameters **settings** (bookstore.bookstore_config.BookstoreSettings) – Instantiated settings object to be validated.

Returns **validation_checks** – Existence of settings by category (general, archive, publish)

Return type dict

1.5.2 Archiving

bookstore.archive

Archival of notebooks

class bookstore.archive.**ArchiveRecord**
Bases: tuple

Represents an archival record.

An *ArchiveRecord* uses a Typed version of *collections.namedtuple()*. The record is immutable.

Example

An archive record (*filepath*, *content*, *queued_time*) contains:

- a *filepath* to the record
- the *content* for archival
- the *queued time* length of time waiting in the queue for archiving

content
Alias for field number 1

filepath
Alias for field number 0

queued_time
Alias for field number 2

class bookstore.archive.**BookstoreContentsArchiver**(**args*, ***kwargs*)
Bases: notebook.services.contents.filemanager.FileContentsManager

Manages archival of notebooks to storage (S3) when notebook save occurs.

This class is a custom Jupyter FileContentsManager which holds information on storage location, path to it, and file to be written.

Example

- Bookstore settings combine with the parent Jupyter application settings.
- A session is created for the current event loop.
- To write to a particular path on S3, acquire a lock.
- After acquiring the lock, *archive* method authenticates using the storage service's credentials.
- If allowed, the notebook is queued to be written to storage (i.e. S3).

path_locks
Dictionary of paths to storage and the lock associated with a path.

Type dict

path_lock_ready
A mutex lock associated with a path.

Type asyncio mutex lock

archive(*record: bookstore.archive.ArchiveRecord*)
Process a record to write to storage.

Acquire a path lock before archive. Writing to storage will only be allowed to a path if a valid *path_lock* is held and the path is not locked by another process.

Parameters **record** (ArchiveRecord) – A notebook and where it should be written to storage

run_pre_save_hook(*model*, *path*, ***kwargs*)
Send request to store notebook to S3.

This hook offloads the storage request to the event loop. When the event loop is available for execution of the request, the storage of the notebook will be done and the write to storage occurs.

Parameters

- **model** (*str*) – The type of file
- **path** (*str*) – The storage location

1.5.3 bookstore.handlers module

Handlers for Bookstore API

class bookstore.handlers.**BookstoreVersionHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs*)

Bases: notebook.base.handlers.APIHandler

Handler responsible for Bookstore version information

Used to lay foundations for the bookstore package. Though, frontends can use this endpoint for feature detection.

get()

bookstore.handlers.**load_jupyter_server_extension**(*nb_app*)

1.5.4 bookstore.s3_paths module

S3 path utilities

bookstore.s3_paths.**s3_display_path**(*bucket*, *prefix*, *path=''*)
Create a display name for use in logs

Parameters

- **bucket** (*str*) – S3 bucket name
- **prefix** (*str*) – prefix for workspace or publish
- **path** (*str*) – The storage location

bookstore.s3_paths.**s3_key**(*prefix*, *path=''*)
Compute the s3 key

Parameters

- **prefix** (*str*) – prefix for workspace or publish
- **path** (*str*) – The storage location

bookstore.s3_paths.**s3_path**(*bucket*, *prefix*, *path=''*)
Compute the s3 path.

Parameters

- **bucket** (*str*) – S3 bucket name
- **prefix** (*str*) – prefix for workspace or publish
- **path** (*str*) – The storage location

1.5.5 Clone

bookstore.clone

Handler to clone notebook from storage.

class bookstore.clone.**BookstoreCloneHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs*)
Handle notebook clone from storage.

Provides API handling for `GET` and `POST` when cloning a notebook from storage (S3). Launches a user interface cloning options page when `GET` is sent.

initialize(*self*)
Helper to access bookstore settings.

get(*self*)
Checks for valid storage settings and render a UI for clone options.

construct_template_params(*self*, *s3_bucket*, *s3_object_key*)
Helper to populate Jinja template for cloning option page.

post(*self*)
Clone a notebook from the location specified by the payload.

get_template(*self*, *name*)
Loads a Jinja template and its related settings.

See also:

Jupyter Notebook reference on Custom Handlers

construct_template_params(*s3_bucket*, *s3_object_key*)
Helper that takes valid S3 parameters and populates UI template

get()
GET /api/bookstore/cloned

Renders an options page that will allow you to clone a notebook from a specific bucket.

get_template(*name*)
Loads a Jinja template by name.

initialize()
Helper to retrieve bookstore setting for the session.

post()
POST /api/bookstore/cloned

Clone a notebook to the path specified in the payload.

The payload type for the request should be:

```
{
"s3_bucket": string,
"s3_key": string,
"target_path"?: string
}
```

The response payload should match the standard Jupyter contents API POST response.

1.5.6 Notebook Client

bookstore.client.nb_client

1.5.7 Bookstore Client

bookstore.client.store_client

1.6 Project

1.6.1 Contributing

Oh, hello there! You're probably reading this because you are interested in contributing to nteract. That's great to hear! This document will help you through your journey of open source. Love it, cherish it, take it out to dinner, but most importantly: read it thoroughly!

What do I need to know to help?

Read the README.md file. This will help you set up the project. If you have questions, please ask on the nteract Slack channel. We're a welcoming project and are happy to answer your questions.

How do I make a contribution?

Never made an open source contribution before? Wondering how contributions work in the nteract world? Here's a quick rundown!

1. Find an issue that you are interested in addressing or a feature that you would like to address.
2. Fork the repository associated with the issue to your local GitHub organization.
3. Clone the repository to your local machine using:

```
git clone https://github.com/github-username/repository-name.git
```

4. Create a new branch for your fix using:

```
git checkout -b branch-name-here
```

5. Make the appropriate changes for the issue you are trying to address or the feature that you want to add.
6. You can run python unit tests using `pytest`. Running integration tests locally requires a more complicated setup. This setup is described in running_ci_locally.md

#. Add and commit the changed files using `git add` and `git commit`. #.

Push the changes to the remote repository using:

```
git push origin branch-name-here
```

1. Submit a pull request to the upstream repository.
2. Title the pull request per the requirements outlined in the section below.
3. Set the description of the pull request with a brief description of what you did and any questions you might have about what you did.
4. Wait for the pull request to be reviewed by a maintainer.
5. Make changes to the pull request if the reviewing maintainer recommends them.
6. Celebrate your success after your pull request is merged! :tada:

How should I write my commit messages and PR titles?

Good commit messages serve at least three important purposes:

- To speed up the reviewing process.
- To help us write a good release note.
- To help the future maintainers of nteract/nteract (it could be you!), say five years into the future, to find out why a particular change was made to the code or why a specific feature was added.

Structure your commit message like this:

```
> Short (50 chars or less) summary of changes
>
> More detailed explanatory text, if necessary.  Wrap it to about 72
> characters or so.  In some contexts, the first line is treated as the
> subject of an email and the rest of the text as the body.  The blank
> line separating the summary from the body is critical (unless you omit
> the body entirely); tools like rebase can get confused if you run the
> two together.
>
> Further paragraphs come after blank lines.
>
>   - Bullet points are okay, too
>
>   - Typically a hyphen or asterisk is used for the bullet, preceded by a
>     single space, with blank lines in between, but conventions vary here
>
```

Source: https://git-scm.com/book/ch5-2.html

DO

- Write the summary line and description of what you have done in the imperative mode, that is as if you were commanding. Start the line with "Fix", "Add", "Change" instead of "Fixed", "Added", "Changed".
- Always leave the second line blank.
- Line break the commit message (to make the commit message readable without having to scroll horizontally in gitk).

DON'T

- Don't end the summary line with a period - it's a title and titles don't end with a period.

Tips

- If it seems difficult to summarize what your commit does, it may be because it includes several logical changes or bug fixes, and are better split up into several commits using `git add -p`.

References

The following blog post has a nice discussion of commit messages:

- "On commit messages" http://who-t.blogspot.com/2009/12/on-commit-messages.html

How fast will my PR be merged?

Your pull request will be merged as soon as there are maintainers to review it and after tests have passed. You might have to make some changes before your PR is merged but as long as you adhere to the steps above and try your best, you should have no problem getting your PR merged.

That's it! You're good to go!

1.6.2 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project maintainer at [rgbkrk@gmail.com]. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available from http://contributor-covenant.org/version/1/4/

1.6.3 Local Continuous Integration

It helps when developing to be able to run integration tests locally. Since bookstore relies on accessing S3, this requires that we create a local server that can model how S3 works.

We will be using minio to mock S3 behavior.

Setup Local CI environment

To run the ci tests locally, you will need to have a few things set up:

- a functioning `docker` service
- define `/mnt/data/` and `/mnt/config/` and give full permissions (e.g., `chmod 777 /mnt/data`). = add `/mnt/data` and `/mnt/config` to be accessible from `docker`. You can do so by modifying Docker's preferences by going to `Docker → Preferences → File Sharing` and adding `/mnt/data` and `/mnt/config` to the list there.
- an up-to-date version of `node`.

Run Local tests

1. Open two terminals with the current working directory as the root `bookstore` directory.
2. In one terminal run `yarn test:server`. This will start up minio.
3. In the other terminal run `yarn test`. This will run the integration tests.

Interactive python tests

The CI scripts are designed to be self-contained and run in an automated setup. This makes it makes it harder to iterate rapidly when you don't want to test the *entire* system but when you do need to integrate with a Jupyter server.

In addition the CI scripts, we have included `./ci/clone_request.py` for testing the clone endpoint. This is particularly useful for the `/api/bookstore/cloned` endpoint because while it is an API to be used by other applications, it also acts as a user facing endpoint since it provides a landing page for confirming whether or not a clone is to be approved.

It's often difficult to judge whether what is being served makes sense from a UI perspective without being able to investigate it directly. At the same time we'll need to access it as an API to ensure that the responses are well-behaved from an API standpoint. By using python to query a live server and a browser to visit the landing page, we can rapidly iterate between the API and UI contexts from the same live server's endpoint.

We provide examples of `jupyter notebook` commands needed in that file as well for both accessing the `nteract-notebooks` S3 bucket as well as the Minio provided `bookstore` bucket (as used by the CI scripts).

1.6.4 Running Python Tests

The project uses pytest to run Python tests and tox as a tool for running tests in different environments.

Setup Local development system

Using Python 3.6+, install the dev requirements:

```
pip install -r requirements-dev.txt
```

Run Python tests

Important: We recommend using tox for running tests locally. Please deactivate any conda environments before running tests using tox. Failure to do so may corrupt your virtual environments.

To run tests for a particular Python version (3.6 or 3.7):

```
tox -e py36  # or py37
```

This will run the tests and display coverage information.

Run linters

```
tox -e flake8
tox -e black
```

Run type checking

```
tox -e mypy
```

Run All Tests and Checks

```
tox
```

1.6.5 Releasing

Pre-release

- [ ] First check that the CHANGELOG is up to date for the next release version.
- [ ] Update docs

Installing twine package

Install and upgrade, if needed,`twine` with `python3 -m pip install -U twine`. The long description of the package will not render on PyPI unless an up-to-date version is used.

Create the release

- [ ] Update version number `bookstore/_version.py`
- [ ] Commit the updated version
- [ ] Clean the repo of all non-tracked files: `git clean -xdfi`
- [ ] Commit and tag the release

```
git commit -am"release $VERSION"
git tag $VERSION
```

- [ ] Push the tags and remove any existing `dist` directory files

```
git push && git push --tags
rm -rf dist/*
```

- [ ] Build `sdist` and `wheel`

```
python setup.py sdist
python setup.py bdist_wheel
```

Test and upload release to PyPI

- [ ] Test the wheel and sdist locally
- [ ] Upload to PyPI using `twine` over SSL

```
twine upload dist/*
```

- [ ] If all went well:
  - Change `bookstore/_version.py` back to `.dev`
  - Push directly to `master` and push `--tags` too.

1.7 Change Log

1.7.1 2.3.0 Unreleased

2.3.0 on Github

Significant changes

Validation information is now exposed as a dict at the `/api/bookstore` endpoint.

This allows us to distinguish whether different features have been enabled on bookstore.

The structure for 2.3.0 is:

```
validation_checks = {
    "bookstore_valid": all(general_settings),
    "archive_valid": all(archive_settings),
    "publish_valid": all(published_settings),
}
```

1.7.2 Releases prior to 2.3.0

2.2.1 (2019-02-03)

2.2.0 (2019-01-29)

2.1.0 (2018-11-20)

2.0.0 (2018-11-13)

0.1 (2018=10-16)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

HTTP Routing Table

/api

Index

V

W