Zocalo Documentation

Release 0.26.0

Markus Gerstel

Jan 06, 2023

CONTENTS:

CHAPTER

ONE

ZOCALO

> M. Gerstel, A. Ashton, R.J. Gildea, K. Levik, and G. Winter, "Data Analysis Infrastructure for Diamond Light Source Macromolecular & Chemical Crystallography and Beyond", in Proc. ICALEPCS'19, New York, NY, USA, Oct. 2019, pp. 1031-1035.

Zocalo is an automated data processing system designed at Diamond Light Source. This repository contains infrastructure components for Zocalo.

The idea of Zocalo is a simple one - to build a messaging framework, where text-based messages are sent between parts of the system to coordinate data analysis. In the wider scope of things this also covers things like archiving, but generally it is handling everything that happens after data aquisition.

Zocalo as a wider whole is made up of two repositories (plus some private internal repositories when deployed at Diamond):

- DiamondLightSource/python-zocalo - Infrastructure components for automated data processing, developed by Diamond Light Source. The package is available through PyPi and conda-forge.
- DiamondLightSource/python-workflows - Zocalo is built on the workflows package. It shouldn't be necessary to interact too much with this package, as the details are abstracted by Zocalo. workflows controls the logic of how services connect to each other and what a service is, and actually send the messages to a message broker. Currently this is an ActiveMQ broker (via STOMP) but support for a RabbitMQ broker (via pika) is being added. This is also available on PyPi and conda-forge.

As mentioned, Zocalo is currently built on top of ActiveMQ. ActiveMQ is an apache project that provides a message broker server, acting as a central dispatch that allows various services to communicate. Messages are plaintext, but from the Zocalo point of view it's passing aroung python objects (json dictionaries). Every message sent has a destination to

help the message broker route. Messages may either be sent to a specific queue or broadcast to multiple queues. These queues are subscribed to by the services that run in Zocalo. In developing with Zocalo, you may have to interact with ActiveMQ or RabbitMQ, but it is unlikely that you will have to configure it.

Zocalo allows for the monitoring of jobs executing `python-workflows` services or recipe wrappers. The `python-workflows` package contains most of the infrastructure required for the jobs themselves and more detailed documentation of its components can be found in the `python-workflows` GitHub repository and the Zocalo documentation.

1.1 Core Concepts

There are two kinds of task run in Zocalo: *services* and *wrappers*. A service should handle a discrete short-lived task, for example a data processing job on a small data packet (e.g. finding spots on a single image in an X-ray crystallography context), or inserting results into a database. In contrast, wrappers can be used for longer running tasks, for example running data processing programs such as xia2 or fast_ep.

- A **service** starts in the background and waits for work. There are many services constantly running as part of normal Zocalo operation. In typical usage at Diamond there are ~100 services running at a time.
- A **wrapper** on the other hand, is only run when needed. They wrap something that is not necessarily aware of Zocalo - e.g. downstream processing software such as xia2 have no idea what zocalo is, and shouldn't have to. A wrapper takes a message, converts to the instantiation of command line, runs the software - typically as a cluster job, then reformats the results into a message to send back to Zocalo. These processes have no idea what Zocalo is, but are being run by a script that handles the wrapping.

At Diamond, everything goes to one service to start with: the **Dispatcher**. This takes the initial request message and attaches useful information for the rest of Zocalo. The implementation of the Dispatcher at Diamond is environment specific and not public, but it does some things that would be useful for a similar service to do in other contexts. At Diamond there is interaction with the ISPyB database that stores information about what is run, metadata, how many images, sample type etc. Data stored in the database influences what software we want to be running and this information might need to be read from the database in many, many services. We obviously don't want to read the same thing from many clients and flood the database, and don't want the database to be a single point of failure. The dispatcher front-loads all the database operations - it takes the data collection ID (DCID) and looks up in ISPyB all the information that could be needed for processing. In terms of movement through the system, it sits between the initial message and the services:

```
message -> Dispatcher -> [Services]
```

At end of processing there might be information that needs to go back into the databases, for which Diamond has a special ISPyB service to do the writing. If the DB goes down, that is fine - things will queue up for the ISPyB service and get processed when the database becomes available again, and written to the database when ready. This isolates us somewhat from intermittent failures.

The only public Zocalo service at present is `Schlockmeister`, a garbage collection service that removes jobs that have been requeued mutliple times. Diamond operates a variety of internal Zocalo services which perform frequently required operations in a data analysis pipeline.

1.2 Working with Zocalo

Graylog is used to manage the logs produced by Zocalo. Once Graylog and the message broker server are running then services and wrappers can be launched with Zocalo.

Zocalo provides the following command line tools::

- `zocalo.go`: trigger the processing of a recipe
- `zocalo.wrap`: run a command while exposing its status to Zocalo so that it can be tracked
- `zocalo.service`: start a new instance of a service
- `zocalo.shutdown`: shutdown either specific instances of Zocalo services or all instances for a given type of service
- `zocalo.queue_drain`: drain one queue into another in a controlled manner

Services are available through `zocalo.service` if they are linked through the `workflows.services` entry point in `setup.py`. For example, to start a Schlockmeister service:

```
$ zocalo.service -s Schlockmeister
```

Q: How are services started?
A: Zocalo itself is agnostic on this point. Some of the services are self-propagating and employ simple scaling behaviour - in particular the per-image-analysis services. The services in general all run on cluster nodes, although this means that they can not be long lived - beyond a couple of hours there is a high risk of the service cluster jobs being terminated or pre-empted. This also helps encourage programming more robust services if they could be killed.

Q: So if a service is terminated in the middle of processing it will still get processed?
A: Yes, messages are handled in transactions - while a service is processing a message, it's marked as "in-progress" but isn't completely dropped. If the service doesn't process the message, or it's connection to ActiveMQ gets dropped, then it get's requeued so that another instance of the service can pick it up.

1.3 Repeat Message Failure

How are repeat errors handled? This is a problem with the system - if e.g. an image or malformed message kills a service then it will get requeued, and will eventually kill all instances of the service running (which will get re-spawned, and then die, and so forth).

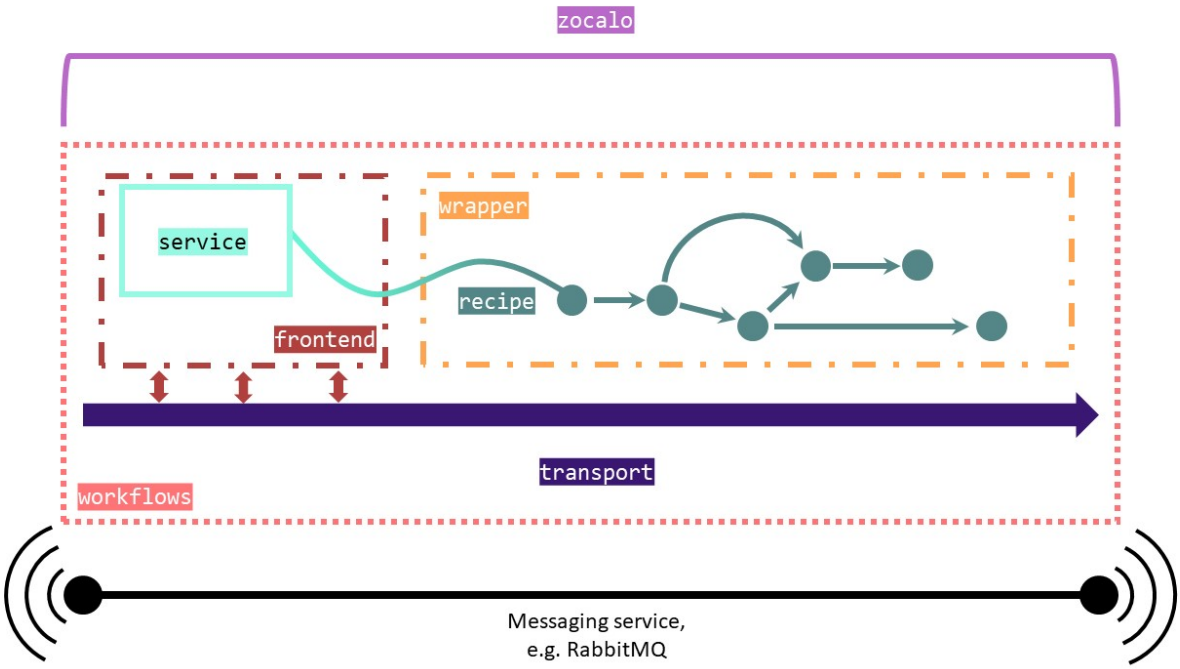
We have a special service that looks for repeat failures and moves them to a special "Dead Letter Queue". This service is called Schlockmeister, and is the only service at time of writing that has migrated to the public zocalo repository. This service looks inside the message that got sent, extracts some basic information from the message in as safe a way as possible and repackages to the DLQ with information on what it was working on, and the "history" of where the message chain has been routed.

CHAPTER

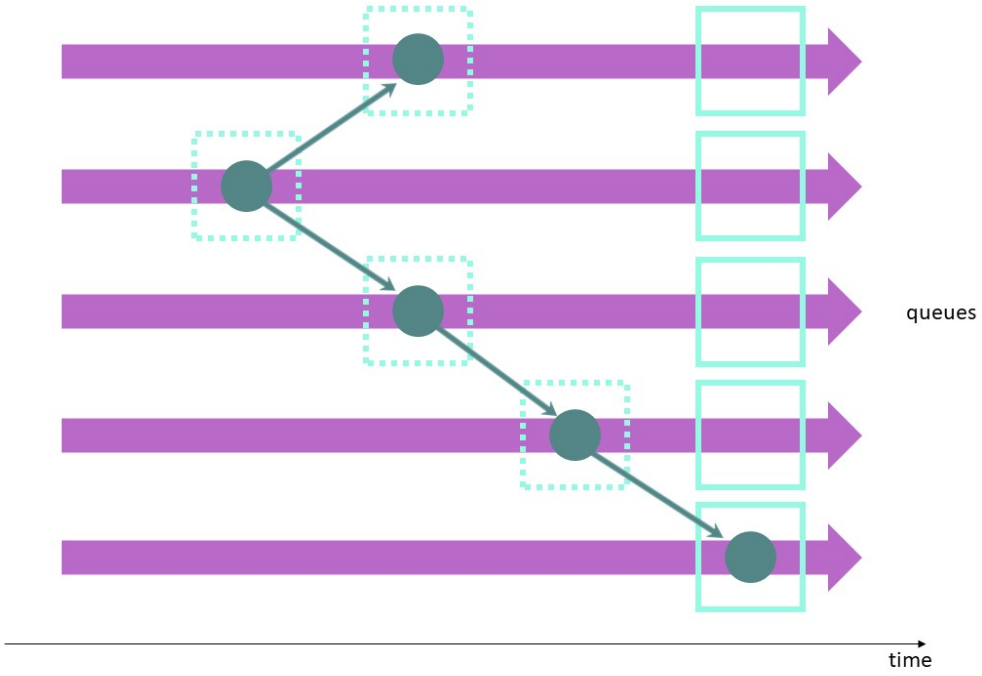
TWO

WORKFLOWS

Zocalo is built on top of the *python-workflows* package. This provides the facilities with which services and recipes for Zocalo are constructed.



`python-workflows` interfaces directly with an externally provided client library for a message broker such as ActiveMQ or RabbitMQ through its `transport` module. Services then take messages, process them, and maybe produce some output. The outputs of services can be piped together through a recipe. Services can also be used to monitor message queues. `python-zocalo` runs `python-workflows` services and recipes, wrapping them so that they are all visible to Zocalo.



This diagram illustrates the overall task management model of Zocalo. Services run continuously, consuming from the relevant queues. Recipes inside of wrappers dictate the flow of data from queue to queue and, therefore, from service to service. The nodes represent input data which is given to the service with the output of a service becoming the input for the next.

CHAPTER

THREE

CONFIGURATION

Zocalo will need to be customised for your specific installation to control aspects such as the settings for the underlying messaging framework, centralised logging, and more.

To achieve this, Zocalo supports the concept of a site configuration file. An example configuration file is included in the Zocalo repository.

3.1 Discovery

Zocalo will, by default, look for the main configuration file at the location specified in the environment variable `ZOCALO_CONFIG`.

You can also specify locations directly if you use Zocalo programmatically, eg.:

```
import zocalo.configuration
zc = zocalo.configuration.from_file("/alternative/configuration.yml")
```

or you can load configurations from a YAML string:

```
zc = zocalo.configuration.from_string("version: 1\n\n...")
```

3.2 Configuration file format

The configuration file is in YAML format. If you are not familiar with YAML then this YAML primer may prove useful.

This documentation describes version 1 of the configuration file format. There is currently no other version. Every site configuration file must declare its version by including, at the top level:

```
version: 1
```

Beyond the version specification every configuration file can contain three different types of blocks:

1. plugin configurations
2. environment definitions
3. references to further configuration files

Let's look at them individually.

3.3 Plugin configurations

Each plugin configuration block follows this basic format:

```
some-unique-name:
    plugin: plugin-name
    ...
```

The name of the plugin configuration blocks (`some-unique-name`) can be chosen freely, and their only restriction is that they should not collide with the names of other blocks that you configure – otherwise the previous definition will be overwritten.

The name of the plugin (`plugin-name`) on the other hand refers to a specific Zocalo configuration plugin. Through the magic of Python entry points the list of potentially available plugins is infinite, and you can easily develop and distribute your own, independently from Zocalo.

Just because a plugin configuration exists does not mean that it is *active*. For this you will need to add the configuration to an environment and activate this environment (see below under *Environment definitions*).

The configuration file may also include configurations for plugins that are not installed. This will raise a warning when you try to enable such a plugin configuration, but it will not cause the rest of the configuration to crash and burn.

Zocalo already includes a few basic plugins, and others may be available to you via other Python packages, such as workflows. A few of the included plugins are detailed here:

3.3.1 Storage plugin

tbd.

3.3.2 Logging plugin

This plugin allows site-wide logging configuration. For example:

```
some-unique-name:
    plugin: logging
    loggers:
      zocalo:
        level: WARNING
      workflows:
        level: WARNING
    verbose:
      - loggers:
          zocalo:
            level: INFO
      - loggers:
          zocalo:
            level: DEBUG
          workflows:
            level: DEBUG
```

would set the Python loggers `zocalo` and `workflows` to only report messages of level `WARNING` and above. Apart from the additional `plugin:`- and `verbose:`-keys the syntax follows the Python Logging Configuration Schema. This allows not only the setting of log levels, but also the definition of log handlers, filters, and formatters.

A plugin definition will, by default, overwrite any previous logging configuration. While it is fundamentally possible to combine multiple configurations (using the `incremental` key), this will cause all sorts of problems and is therefore strongly discouraged.

Please note that Zocalo commands will currently always add a handler to log to the console. This behaviour may be reviewed in the future.

The Zocalo configuration object exposes a facility to read out and increase a verbosity level, which will apply incremental changes to the logging configuration. In the above example setting `zc.logging.verbosity = 1` would change the log level for `zocalo` to `INFO` while leaving `workflows` at `WARNING`. Setting `zc.logging.verbosity = 2` would change both to `DEBUG`.

Note that the verbosity level cannot be decreased, and due to the Python Logging model verbosity changes should be done close to the initial logging setup, as otherwise child loggers may have been set up inheriting previous settings.

The logging plugin offers two Graylog handlers (`GraylogUDPHandler`, `GraylogTCPHandler`). These are based on graypy, but offer slightly improved performance by front-loading DNS lookups and apply a patch to `graypy` to ensure syslog levels are correctly reported to Graylog. To use these handlers you can declare them as follows:

```
some-unique-name:
    plugin: logging
    handlers:
      graylog:
        (): zocalo.configuration.plugin_logging.GraylogUDPHandler
        host: example.com
        port: 1234
    root:
      handlers: [ graylog ]
```

The logging plugin offers a log filter (`DowngradeFilter`), which can be attached to loggers to reduce the severity of messages. It takes two parameters, `reduce_to` (default: `WARNING`) and `only_below` (default: `CRITICAL`), and messages with a level between `reduce_to` and `only_below` have their log level changed to `reduce_to`:

```
some-unique-name:
    plugin: logging
    filters:
      downgrade_all_warnings_and_errors:
        (): zocalo.configuration.plugin_logging.DowngradeFilter
        reduce_to: INFO
    loggers:
      pika:
        filters: [ downgrade_all_warnings_and_errors ]
```

3.3.3 Graylog plugin

This should be considered deprecated and will be removed at some point in the future. Use the Logging plugin instead.

3.4 Environment definitions

```
environments:
  env-name:
    plugins:
      - some-unique-name
      - ...
```

Environments aggregate multiple plugin configuration blocks together, and environments are what you load to set up specific plugin configurations. The environment names (`env-name`) can again be chosen freely. Underneath environments you can optionally declare groups (here: `plugins`). These groups affect the order in which the plugin configurations take effect, and they also play a role when a configuration file is split up across multiple files. If you don't specify a group name then the default group name `plugins` is used.

Groups are loaded alphabetically, with one exception: `plugins` is special and is always loaded last. Within each group the plugin configurations are loaded in the specified order.

A special environment name is `default`, which is the environment that will be loaded if no other environment is loaded. You can use aliasing (see below under *Environment aliases*) to point `default` to a different, more self-explanatory environment name.

3.4.1 Environment aliases

You can create aliases for environment names by just giving the name of the underlying environment name. You can only do pure aliasing here, you can not override parts of the referenced environment at this time.

This configuration gives you an `alias` environment, that is exactly identical to the environment named `real`:

```
environments:
  real:
    plugins:
      - ...
  alias: real
```

Aliases are resolved immediately when they are encountered. The aliased environment therefore has to be specified in the same configuration file.

3.5 References to further files

tbd.

CHAPTER

FOUR

WRITING YOUR OWN PLUGINS

tbd.

CHAPTER

FIVE

INSTALLATION

5.1 Stable release

To install Zocalo, run this command in your terminal:

```
$ pip install zocalo
```

This is the preferred method to install Zocalo, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

5.2 From sources

The sources for Zocalo can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/DiamondLightSource/zocalo-python
```

Or download the tarball:

```
$ curl  -OL https://github.com/DiamondLightSource/zocalo-python/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

SIX

GETTING STARTED

Zocalo requires both ActiveMQ and Graylog to be setup and running. The easiest way of setting these up is via docker.

6.1 Active MQ

Pull and run the following image https://hub.docker.com/r/rmohr/activemq Follow the steps on docker hub for extracting the config and data into local mounts

Configure DLQ locations, see https://activemq.apache.org/message-redelivery-and-dlq-handling for more info.

In *conf/activemq.xml* under *policyEntries* add:

```
<policyEntry queue=">">
    <deadLetterStrategy>
        <individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true"/
↪>
    </deadLetterStrategy>
</policyEntry>
```

Make sure to enable scheduling, in *conf/activemq.xml* in the *broker* tag add the following property:

```
schedulerSupport="true"
```

Its also a good idea to enable removal of unused queues, see https://activemq.apache.org/delete-inactive-destinations

In *conf/activemq.xml* in the *broker* tag add the following property:

```
schedulePeriodForDestinationPurge="10000"
```

Then in the *policyEntry* tag for *queue=">"* add the following properties:

```
gcInactiveDestinations="true" inactiveTimoutBeforeGC="120000"
```

Which will purge unused queues on a 120s basis.

Then start ActiveMQ:

```
docker run --name activemq -p 61613:61613 -p 8161:8161 \
    -v "$(pwd)/conf:/opt/activemq/conf" \
    -v "$(pwd)/data:/opt/activemq/data" \
    rmohr/activemq
```

The container exposes the following ports:

Port	Description
61613	Stomp transport
8161	Web Console / Jolokia REST API

A preconfigured docker image with these options applied is available here https://hub.docker.com/r/esrfbcu/zocalo-activemq

6.2 Graylog

This can be started easily with a docker-compose.yml. See https://docs.graylog.org/en/3.3/pages/installation/docker.html for full details.

```
version: '3'
services:
# MongoDB: https://hub.docker.com/_/mongo/
mongo:
    image: mongo:4.2
    networks:
    - graylog
# Elasticsearch: https://www.elastic.co/guide/en/elasticsearch/reference/6.x/docker.html
elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch-oss:7.10.0
    environment:
    - http.host=0.0.0.0
    - transport.host=localhost
    - network.host=0.0.0.0
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
    memlock:
        soft: -1
        hard: -1
    deploy:
    resources:
        limits:
        memory: 1g
    networks:
    - graylog
# Graylog: https://hub.docker.com/r/graylog/graylog/
graylog:
    image: graylog/graylog:4.0
    environment:
    - GRAYLOG_PASSWORD_SECRET=mysecret
    # Password: admin
    - GRAYLOG_ROOT_PASSWORD_
↪SHA2=8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918
    - GRAYLOG_HTTP_EXTERNAL_URI=http://localhost:9000/
    networks:
    - graylog
    restart: always
```

(continues on next page)

(continued from previous page)

```
    depends_on:
    - mongo
    - elasticsearch
    ports:
    # Graylog web interface and REST API
    - 9000:9000
    # Syslog TCP
    - 1514:1514
    # Syslog UDP
    - 1514:1514/udp
    # GELF TCP
    - 12201:12201
    # GELF UDP
    - 12201:12201/udp
networks:
graylog:
    driver: bridge
```

Then start with:

```
docker-compose up
```

Graylog admin console should be available on http://localhost:9000 Port 12201 is available for python GELF logging. Configure an input in the graylog web console to enable receiving messages.

6.3 Zocalo

For developing create a new conda / virtual environment, clone zocalo, and install:

```
conda create -n zocalo
conda activate zocalo
git clone https://github.com/DiamondLightSource/python-zocalo
cd python-zocalo
pip install -e .
```

For production, install with pip:

```
pip install zocalo
```

6.4 Configure

Copy *contrib/site-configuration.yml*. At minimum *graylog* and *activemq* must be configured. Environments should be defined for *live* and *test*. Paths to recipes and drop files must also be specified. Messages are written to drop files if ActiveMQ is unavailable.

The config file to use is specified from the environment variable *ZOCALO_CONFIG*.

Sample recipes can be used:

```
storage:
  plugin: storage
  zocalo.recipe_directory: .../python-zocalo/examples/recipes
```

6.4.1 JMX

To make use of *zocalo.queue_monitor* and *zocalo.status_monitor* JMX needs to be configured. The JMX configuration points to the Jolokia REST API. When starting ActiveMQ the logs will tells you where the REST API is running

```
INFO | ActiveMQ Jolokia REST API available at http://0.0.0.0:8161/api/jolokia/
```

So configuration should be

```
port: 8161
host: localhost
base_url: api/jolokia
```

Username and password are the same as the web console and defined in *users.properties*

6.5 Starting Up

-e test will make use of the test environment

Start the dispatcher

```
conda activate zocalo
zocalo.service -s Dispatcher (-e test)
```

Start the process runner

```
zocalo.service -s Runner (-e test)
```

Run the test recipe:

```
zocalo.go -r example -s workingdir="$(pwd)" 1234 (-e test)
```

6.6 Dead Letter Queue (DLQ)

The dead letter queue is where rejected messages end up. One dlq is available per topic to easily identify where messages are being rejected. For details on dlq see https://activemq.apache.org/message-redelivery-and-dlq-handling

Messages can be purged using:

```
zocalo.dlq_purge --output-directory=/path/to/dlq (-e test)
```

And re-injected with:

```
zocalo.dlq_reinject dlq_file (-e test)
```

CHAPTER

SEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/DiamondLightSource/zocalo-python/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

Zocalo could always use more documentation, whether as part of the official Zocalo docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/DiamondLightSource/zocalo-python/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up zocalo for local development.

1. Fork the *zocalo-python* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/zocalo-python.git zocalo
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv zocalo
$ cd zocalo/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 zocalo tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for all currently supported Python versions. Tests will be run automatically when you create the pull request.

7.4 Tips

To run a subset of tests:

```
$ py.test tests.test_zocalo
```

7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

EIGHT

CREDITS

- Daniel Hatton
- Markus Gerstel
- Richard Gildea
- Stu Fisher

CHAPTER

NINE

HISTORY

9.1 Unreleased

- Add Dockerfile and build-and-push-docker-image GitHub workflow

9.2 0.26.0 (2022-11-04)

- Add dispatcher service
- Add support for Python 3.11

9.3 0.25.1 (2022-10-19)

- `JSONLines` service: trigger `process_messages` immediately when reaching 100 stored messages

9.4 0.25.0 (2022-10-13)

- Add `JSONLines` service for appending messages to a file in jsonlines format

9.5 0.24.2 (2022-10-04)

- `zocalo.configure_rabbitmq` cli: downgrade "No matching queue found" error to warning

9.6 0.24.1 (2022-08-24)

- `zocalo.configure_rabbitmq` cli: additional debugging output in event of rare `IndexError`

9.7 0.24.0 (2022-08-17)

- `zocalo.configure_rabbitmq` cli: enable configuration of vhosts

9.8 0.23.0 (2022-08-02)

- Remove deprecated `zocalo.enable_graylog()` function
- Use `LoggingAdapter` to append `recipe_ID` to wrapper logs. This was inadvertantly broken for the logging plugin added in #176. Derived wrappers should now use self.log rather than instantiating a logger directly.

9.9 0.22.0 (2022-07-12)

- `zocalo.wrapper`: Enable access to `zocalo.configuration` object through `BaseWrapper.config` attribute
- `zocalo.configure_rabbitmq` cli: check response status codes to catch failed API calls
- `zocalo.configure_rabbitmq` cli: don't set x-single-active-consumer for streams

9.10 0.21.0 (2022-06-28)

- `zocalo.configure_rabbitmq` cli: require passing user config via explicit `--user-config` parameter
- `zocalo.configure_rabbitmq` cli: optionally disable implicit dlq creation via `dead-letter-queue-create: false`

9.11 0.20.0 (2022-06-17)

- `zocalo.configure_rabbitmq` cli: require explicit *dead-letter-routing-key-pattern* when requesting creation of a DLQ for a given queue.

9.12 0.19.0 (2022-05-24)

- `zocalo.configure_rabbitmq` cli: advanced binding configuration

9.13 0.18.0 (2022-04-12)

- Added a logging configuration plugin to comprehensively configure logging across applications.

9.14 0.17.0 (2022-03-03)

- **`zocalo.configure_rabbitmq` cli:**
  - Support for explicitly declaring exchanges
  - Allow queues to bind to more than one exchange

9.15 0.16.0 (2022-02-21)

- Add `Mailer` service for sending email notifications. Subscribes to the `mailnotification` queue. SMTP settings are specified via the `smtp` plugin in `zocalo.configuration`.

9.16 0.15.0 (2022-02-16)

- Fix for getting user information from the RabbitMQ management API
- Major changes to the RabbitMQ configuration command line tool. Users are now updated and deleted, and the tool now understands zocalo environment parameters. Configuration files are now mandatory, and the `--seed` parameter has been removed.

9.17 0.14.0 (2021-12-14)

- `zocalo.dlq_purge` offers a `--location` flag to override where files are being written
- `zocalo.dlq_reinject` can again understand `zocalo.dlq_purge` output passed on stdin
- Reinjected messages now carry a `dlq-reinjected: True` header field

9.18 0.13.0 (2021-12-01)

- `zocalo.queue_drain` now allows the automatic determination of destination queues for recipe messages
- `zocalo.queue_drain` fixed for use in a RabbitMQ environment
- `zocalo.dlq_purge` fixed for use in a RabbitMQ environment
- New functions in `zocalo.util` to easily annotate log messages with system context information

9.19 0.12.0 (2021-11-15)

- Add support for queue/exchange bindings to `RabbitMQAPI`
- Drop support for Python 3.6 and 3.7

9.20 0.11.1 (2021-11-08)

- Add a RabbitMQ HTTP API in `zocalo.util.rabbitmq`

9.21 0.11.0 (2021-11-03)

- Add command line tools for handling dead-letter messages
- `zocalo.dlq_check` checks dead-letter queues for messages
- `zocalo.dlq_purge` removes messages from specified DLQs and dumps them to a directory specified in the Zocalo configuration
- `zocalo.dlq_reinject` takes a serialised message produced by `zocalo.dlq_purge` and places it back on a queue
- Use `argparse` for all command line tools and make use of `workflows` transport argument injection. Minimum `workflows` version is now 2.14
- New `zocalo.util.rabbitmq.RabbitMQAPI()` providing a thin wrapper around the RabbitMQ HTTP API

9.22 0.10.0 (2021-10-04)

- New `zocalo.shutdown` command to shutdown Zocalo services
- New `zocalo.queue_drain` command to drain one queue into another in a controlled manner
- New `zocalo.util.rabbitmq.http_api_request()` utility function to return a `urllib.request.Request` object to query the RabbitMQ API using the credentials specified via `zocalo.configuration`.
- `zocalo.wrap` now emits tracebacks on hard crashes and `SIGUSR2` signals

9.23 0.9.1 (2021-08-18)

- Expand ~ in paths in configuration files

9.24 0.9.0 (2021-08-18)

- Removed –live/–test command line arguments, use -e/–environment instead
- zocalo.go, zocalo.service, zocalo.wrap accept -t/–transport command line options, and the default can be set via the site configuration.

9.25 0.8.1 (2021-07-08)

- Keep wrapper status threads alive through transport disconnection events

9.26 0.8.0 (2021-05-18)

- Support for Zocalo configuration files

9.27 0.7.4 (2021-03-17)

- Documentation improvements

9.28 0.7.3 (2021-01-19)

- Ignore error when logserver hostname can't be looked up immediately

9.29 0.7.2 (2021-01-18)

- Add a symbolic link handling library function
- Cache the logserver hostname by default

9.30 0.7.1 (2020-11-13)

- Add a –dry-run option to zocalo.go

9.31 0.7.0 (2020-11-02)

- Drop support for Python 3.5
- Update language constructs for Python 3.6+

9.32 0.6.4 (2020-11-02)

- Add support for Python 3.9

9.33 0.6.3 (2020-05-25)

- Remove stomp.py requirement - this is pulled in via workflows only

9.34 0.6.2 (2019-07-16)

- Set live flag in service environment if service started with '–live'

9.35 0.6.0 (2019-06-17)

- Start moving dlstbx scripts to zocalo package: * zocalo.go * zocalo.wrap
- Entry point 'dlstbx.wrappers' has been renamed 'zocalo.wrappers'
- Dropped Python 3.4 support

9.36 0.5.4 (2019-03-22)

- Compatibility fixes for graypy >= 1.0

9.37 0.5.2 (2018-12-11)

- Don't attempt to load non-existing file

9.38 0.5.1 (2018-12-04)

- Fix packaging bug which meant files were missing from the release

9.39 0.5.0 (2018-12-04)

- Add zocalo.service command to start services

9.40 0.4.0 (2018-12-04)

- Add status notification thread logic

9.41 0.3.0 (2018-12-04)

- Add schlockmeister service and base wrapper class

9.42 0.2.0 (2018-11-28)

- Add function to enable logging to graylog

9.43 0.1.0 (2018-10-19)

- First release on PyPI.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search