

MyMONIT

Collecting measurements to monitor CERN's experiments – README

Team 2: Ahmed, Padukka, Rossotto, Thompson, Wimalendran

Table of Contents

Application composition.....	3
Overview.....	3
Components.....	3
Class diagram of APP.....	6
Overview.....	6
RestListener.....	6
RabbitListener.....	7
Logging.....	8
Storage.....	10
Dependency Injection.....	11
Orchestration.....	13
Technical specifications.....	15
Requirements.....	15
Containers overview.....	15
Summary of container images.....	15
Production configuration.....	16
Scalability of App.....	16
Scalability of MySQL, Nginx, and RabbitMQ.....	16
Scalability of Logstash, Elasticsearch, and Kibana.....	17
Ports.....	17

Setup.....	17
Authentication.....	18
User creation.....	18
Authentication process.....	20
Testing.....	21
Unit testing.....	21
Automatic scanners.....	21
Bandit.....	22
Coverage.....	22
Pylint.....	23
Manual testing.....	24
APIs sequence.....	25
Calling the APIs.....	26
API documentation.....	27
List of attachments.....	28
References.....	29

Application composition

Overview

As mentioned in the analysis (Rossotto et al, 2022), CERN uses a variety of independently developed systems to monitor the infrastructure used in the experiments (Aimar et al., 2019). MyMONIT unifies the monitoring and integrates the streams of different applications into a single repository.

MyMonit collects the input from the experiments (the measures) via message broker and expose REST APIs to consume them. MyMonit exposes a separate interface for auditing the solution itself. The whole solution is composed of different containers orchestrated by Docker Compose.

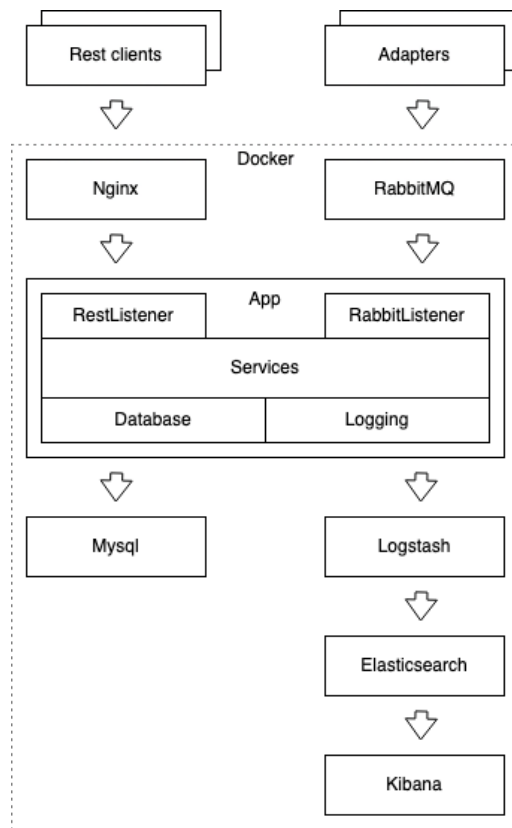
Components

The application is composed of seven containers running in Docker.

- App is the core component. It hosts the business logic and it is written in Python. It exposes REST APIs and consumes messages via AMQP. Its stack is composed of
 - Flask (Flask, N.D.): framework used to create REST APIs.
 - Pika (Pika, N.D.): framework to consume messages from the AMQP interface.
 - MySQL Connector (MySQL Connector, N.D.): database connector.
 - Python Logstash (Elastic, N.D.): connector to the Logstash server to store logs for auditing purpose.

- Firebase (Firebase, N.D.): connector to Firebase for the authentication of Json Web Tokens.
- Dependency Injector (Dependency Injector, N.D.): framework to implement dependency injection in Python.
- Colorama (Hartley J. (2021): framework to output ANSI colored text in console.
- MySQL (MySQL, N.D.) is the core database. It contains the users with access to App, and the data relative to the experiments.
- RabbitMQ (RabbitMQ, N.D.) is the AMQP message broker responsible for the collection of the measures and the delivery to App.
- Nginx (Nginx, N.D.) is the HTTP/S proxy that exposes App's REST APIs to the external world.
- Logstash (Logstash, N.D.) is responsible for log collection, formatting, and correct storage in Elasticsearch.
- Elasticseach (Elasticsearch, N.D.) is the database responsible for the collection of all the logs from all other containers.
- Kibana (Kibana, N.D.) is the frontend application that visualizes the content of Elasticsearch.

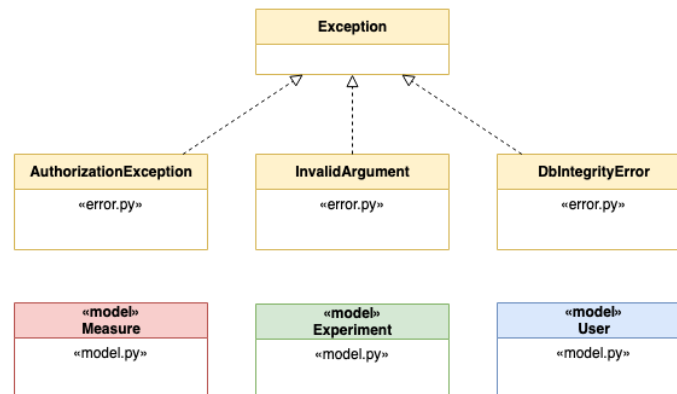
Additionally, the application supports "adapters": small applications designed to translate external input into a correctly formatted message that can be enqueued in RabbitMQ.



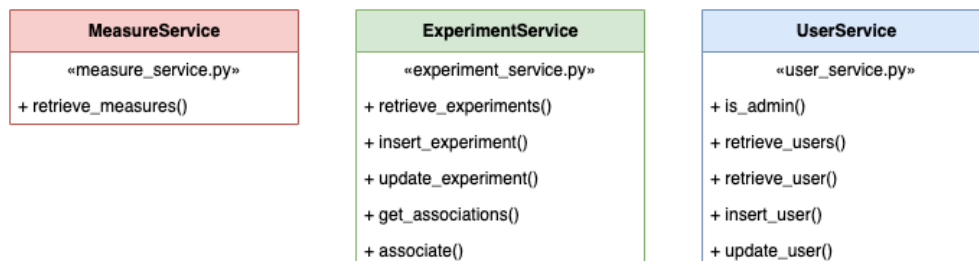
Class diagram of APP

Overview

The model is composed of Measure, Experiment, User, and Exceptions.

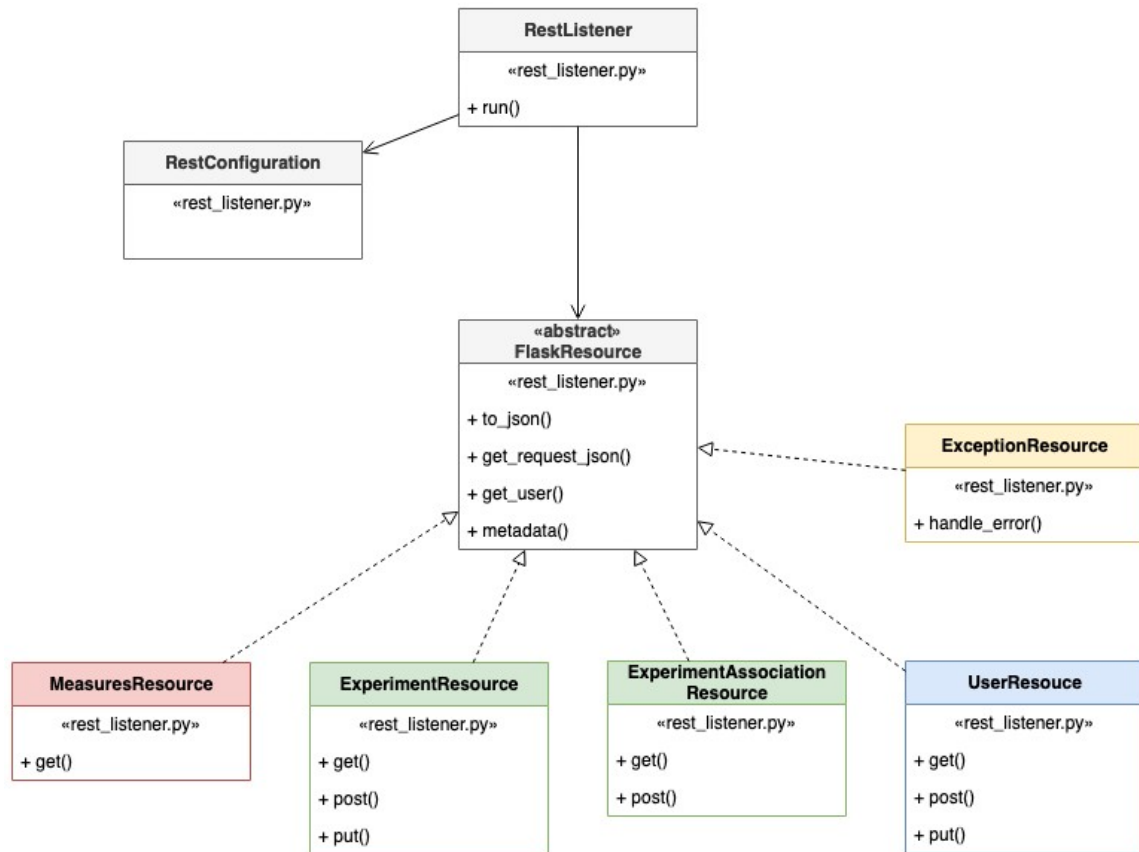


A specialised service is responsible for the three main elements of the model.



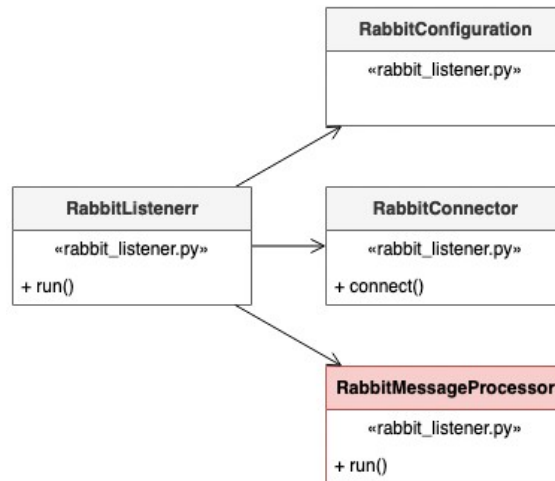
RestListener

RestListener is responsible for the REST interface of the application. RestListener uses an object for the configuration, and delegates the handling of the resources to specialised components.



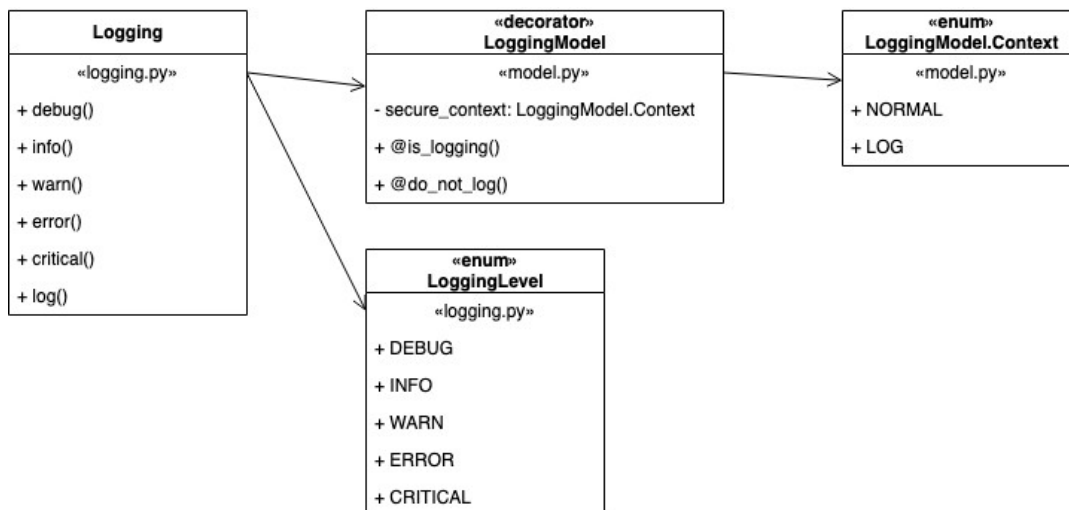
RabbitListener

RabbitListener is the service responsible for the connection to RabbitMQ and it uses a RabbitConfiguration to group the configuration. RabbitConnector is a component of the listener used to extract the IO functions and simplify unit testing. RabbitMessageProcessor contains the logic to store the measures in the database.



Logging

Logging is a cross-cutting service used in different components.



Logging supports faceted data (Schmitz, 2016) via two decorators `@is_logging` and `@do_not_log` to respectively mark functions that perform logging, and parts of the model that should not appear in the logs with their real value to prevent data leaks.

`@is_logging` wraps the decorated call setting the context's value.


```

@staticmethod
def is_logging(func):
    def inner(*args, **kwargs):
        LoggingModel._set_context_value(LoggingModel.Context.LOG)
        func(*args, **kwargs)
        LoggingModel._set_context_value(LoggingModel.Context.NORMAL)
    return inner

```

@do_no_log checks the context's value and redirects the call to the original method or returns "***" depending on the context's value. This simple mechanism can hide secrets in the logs without implementing a case-by-case logic in the logging component.

```

@staticmethod
def do_not_log(func):
    def inner(*args, **kwargs):
        return func(*args, **kwargs) if LoggingModel._get_context_value() == \
            LoggingModel.Context.NORMAL else '***'
    return inner

```

The component supports multithreading with a dictionary of contexts indexed by thread id.

```

@staticmethod
def _get_context():
    return LoggingModel.Context.secure_context

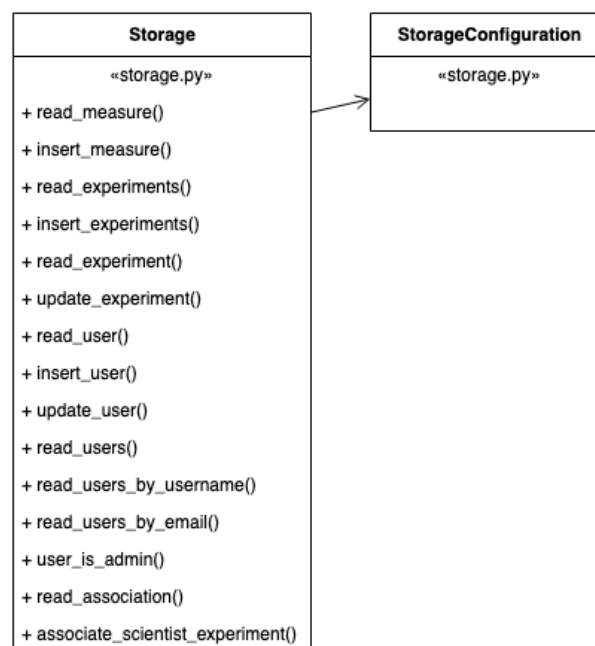
@staticmethod
def _get_context_value():
    thread_id = threading.get_native_id()
    context = LoggingModel._get_context()
    if thread_id not in context:
        context[thread_id] = LoggingModel.Context.NORMAL
    return context[thread_id]

@staticmethod
def _set_context_value(value:Context):
    thread_id = threading.get_native_id()
    context = LoggingModel._get_context()
    context[thread_id] = value

```

Storage

Storage is responsible for the IO with the database.



Dependency Injection

Container is responsible for the configuration and dependency injection. Thanks to

Container, components do not instantiate one another simplifying unit testing.

Container
«app.py» - config: dependency_injection.provider.Configuration + main() + init()

The configuration of Container is in app.py. The parameters have a default value that can be overridden by the environment's parameters.

```
def init():
    '''initialise the application'''
    container = Container()

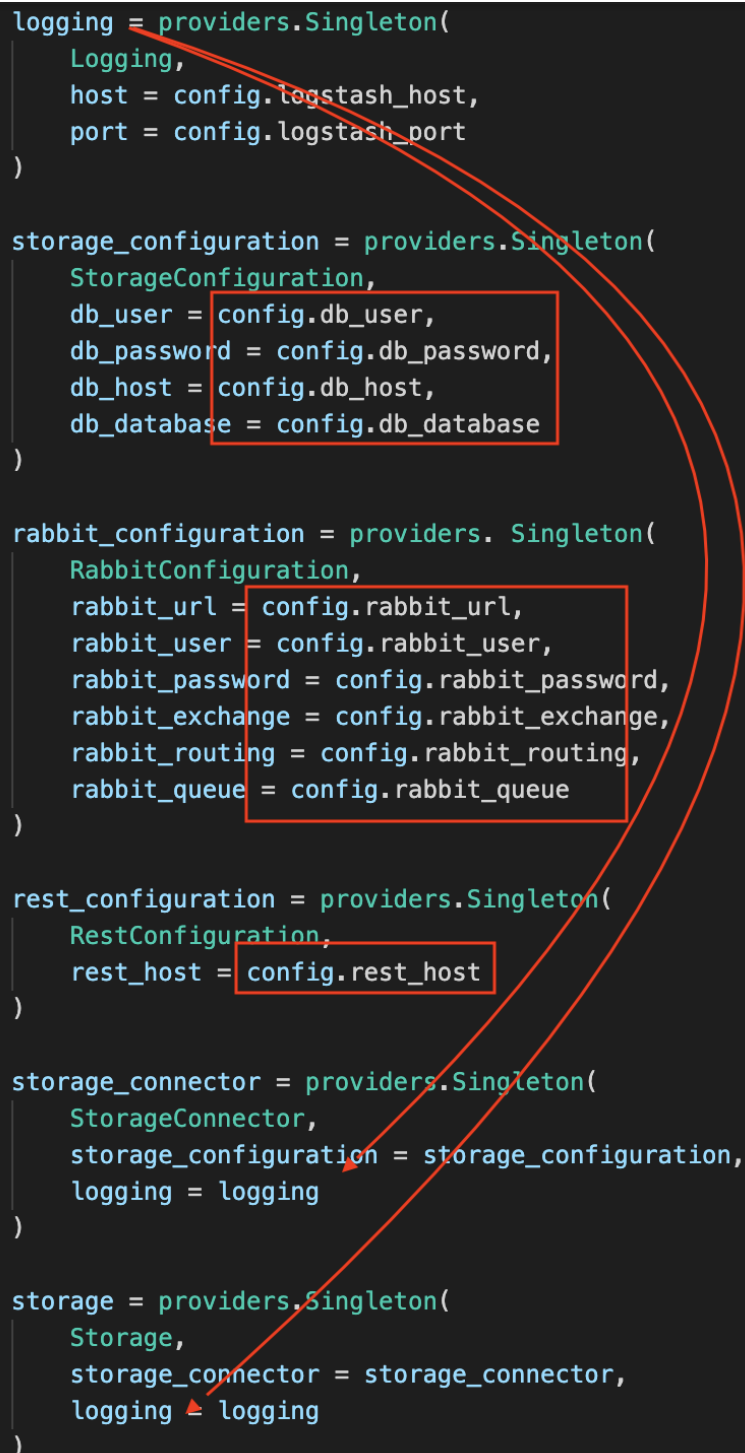
    # db
    container.config.db_user.from_env('DB_USER', default = 'root', as_ = str)
    container.config.db_password.from_env('DB_PASSWORD', default = 'password', as_ = str)
    container.config.db_host.from_env('DB_HOST', default = 'localhost', as_ = str)
    container.config.db_database.from_env('DB_DATABASE', 'my_monit', as_ = str)
    # rabbit
    container.config.rabbit_url.from_env('RABBIT_URL', default = 'localhost', as_ = str)
    container.config.rabbit_user.from_env('RABBIT_USER', default = 'guest', as_ = str)
    container.config.rabbit_password.from_env('RABBIT_PASSWORD', default = 'guest', as_ = str)
    container.config.rabbit_exchange.from_env('RABBIT_EXCHANGE', default = 'mymonit', as_ = str)
    container.config.rabbit_routing.from_env('RABBIT_ROUTING', default = 'measures', as_ = str)
    container.config.rabbit_queue.from_env('RABBIT_QUEUE', default = 'measures', as_ = str)
    # rest
    container.config.rest_host.from_env('REST_HOST', default = '0.0.0.0', as_ = str)
    # logstash
    container.config.logstash_host.from_env('LOGSTASH_HOST', default = 'localhost', as_ = str)
    container.config.logstash_port.from_env('LOGSTASH_PORT', default = 5959, as_ = int)
    # firebase
    if 'GOOGLE_APPLICATION_CREDENTIALS' not in os.environ:
        # os.getcwd() is the whole project's root
        os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = './containers/app/private_key.json'
```

The dependency injection mechanism in Container uses the containers.config to create some object and inject the objects one into another.

For example, logging is a basic object without dependencies and it is injected into storage.

Dependency injection is essential to perform accurate unit testing because it makes object-mocking easy.

```
logging = providers.Singleton(  
    Logging,  
    host = config.logstash_host,  
    port = config.logstash_port  
)  
  
storage_configuration = providers.Singleton(  
    StorageConfiguration,  
    db_user = config.db_user,  
    db_password = config.db_password,  
    db_host = config.db_host,  
    db_database = config.db_database  
)  
  
rabbit_configuration = providers.Singleton(  
    RabbitConfiguration,  
    rabbit_url = config.rabbit_url,  
    rabbit_user = config.rabbit_user,  
    rabbit_password = config.rabbit_password,  
    rabbit_exchange = config.rabbit_exchange,  
    rabbit_routing = config.rabbit_routing,  
    rabbit_queue = config.rabbit_queue  
)  
  
rest_configuration = providers.Singleton(  
    RestConfiguration,  
    rest_host = config.rest_host  
)  
  
storage_connector = providers.Singleton(  
    StorageConnector,  
    storage_configuration = storage_configuration,  
    logging = logging  
)  
  
storage = providers.Singleton(  
    Storage,  
    storage_connector = storage_connector,  
    logging = logging  
)
```



The diagram illustrates the dependency injection process. A red arrow originates from the `logging` variable and points to the `logging` parameter in the `StorageConnector` constructor. Another red arrow originates from the `storage_configuration` variable and points to the `storage_configuration` parameter in the `StorageConnector` constructor. A third red arrow originates from the `storage_connector` variable and points to the `storage_connector` parameter in the `Storage` constructor. A fourth red arrow originates from the `logging` variable and points to the `logging` parameter in the `Storage` constructor. Red boxes highlight the configuration parameters for `StorageConfiguration`, `RabbitConfiguration`, `RestConfiguration`, and `StorageConnector`.

Orchestration

The orchestration uses Docker Compose (Docker Compose, N.D.) via the descriptor `docker-compose.yml`.

```
app:
  image: safe_repository:latest
  container_name: app
  environment:
    - DB_USER=root
    - DB_PASSWORD=password
    - DB_HOST=mysql
    - DB_DATABASE=my_monit
    - RABBIT_URL=rabbit
    - RABBIT_USER=guest
    - RABBIT_PASSWORD=guest
    - RABBIT_EXCHANGE=mymonit
    - RABBIT_ROUTING=measures
    - RABBIT_QUEUE=measures
    - REST_HOST=0.0.0.0
    - LOGSTASH_HOST=logstash
    - LOGSTASH_PORT=5959
    - GOOGLE_APPLICATION_CREDENTIALS=/firebase_config.json
  depends_on:
    - elasticsearch
    - rabbit
    - mysql
  volumes:
    - ./containers/app/private_key.json:/firebase_config.json
```

The environment list specifies the variables for the container, including the secrets.

The included example does not use strong secrets and it is supposed to be only an example. It is recommended to change the secrets in a production environment. It is also recommended to use this configuration as a model to deploy into Kubernetes that offers a stronger solution for deployment, including the possibility to separate the secrets from the code (Kubernetes, N.D.).

```
rabbit:  
  image: "rabbitmq:3-management"  
  container_name: rabbit  
  ports:  
    - "5672:5672"  
    - "15672:15672"
```

Some port are exposed to easily demonstrate the usage of the components. Only the ports mentioned in the section “production configuration” are required for a correct functioning.

Technical specifications

Requirements

It is required to have the Docker daemon running. For the development, the chosen platform was Docker Desktop. It is recommended to assign at least 4gb of ram to Docker: with less memory, some containers may fail. It is always possible to restart failing containers should that happen.

All scripts require a unix-like system (eg. Mac).

Python 3.9 is required to build App alone. All dependencies are specified in setup.py and can be installed with “pip3 install -e .”

Containers overview

The configuration of all containers is visible in “docker-compose.yml”. The file is meant to be an example and it is not suitable for production environment because it contains unencrypted secrets and exposes internal ports.

Summary of container images

Container's name	Base image	Customisation
App	Python:3.9	Injection of the application
Nginx	Nginx:latest	Inject of configuration and SSL certificates
Rabbit	Rabbit:3-management	None

Container's name	Base image	Customisation
Mysql	Mysql:5.7	None
Elasticsearch	Elasticsearch:6.5.4	None
Kibana	Kibana:6.5.4	None
Logstash	Logstash:6.5.4	None

Production configuration

It is recommended to deploy the application in Kubernetes. Kubernetes makes it easy to share secrets in a safe way without embedding them in the code, and offers advanced functions to ensure scalability.

Scalability of App

The core Python application is stateless and it is possible to start multiple instances in parallel to improve performances.

If the goal is to increase performances of the REST APIs, it is required a change in the Nginx configuration to implement load-balancing between the nodes.

If the goal is to increase performances of the consumption of AMQP messages through the RabbitMQ interface, it is sufficient to increase the number of nodes to multiply the consumers. All the calls to the REST APIs will be handled by one single node, but this does not represent an issue.

Scalability of MySQL, Nginx, and RabbitMQ

These components natively support clustering through configuration (MySQL – Clustering, N.D.; Nginx – Clustering, N.D; RabbitMQ – Clustering, N.D.).

Scalability of Logstash, Elasticsearch, and Kibana

The ELK stack supports clustering through configuration (Elastic – Clustering. N.D.).

It is reasonable to expect that the components under pressure may be Logstash and Elasticsearch because they have the responsibility to store logs from all components.

Kibana's users will be limited and it is not expected to suffer of scalability problems.

Ports

The only ports that need to be published in a production environments are:

- 443: Nginx
- 5672: RabbitMQ
- 5601: Kibana

All other ports in the docker compose descriptor are exposed only for testing and should remain closed to external connections in a production environment.

Setup

The script "run.sh" automatically builds and starts all the containers.

The script "run-infra.sh" starts only a subset of the containers (Mysql and RabbitMQ) and it is convenient for the development of the core application in Python.

The Python application can be built with "pip3 install -e ." launched in the App's container base folder. The command downloads automatically the dependencies specified in "setup.py".

Authentication

The authentication process is externalized. The application contains GDPR sensitive data (namely emails), but no passwords in any form.

The chosen platform for the external authentication is Firebase (Firebase, N.D.) hosted in the cloud by Google.

Users must register and verify their identity only with Firebase to generate a signed Json Web Token that contains their unique id and can be verified by App during a REST call.

User creation

A user must be created in the Firebase console (Firebase – Console, N.D.). The console can be accessed using the credentials in the attachment.

The console shows all the users and their User UID which is the information that links a Firebase's User with App's User.

Authentication

[Users](#) [Sign-in method](#) [Templates](#) [Usage](#)

Search by email address, phone number, or user UID					Add user		
Identifier	Providers	Created ↓	Signed In	User UID			
kbouman@home.cern	✉	May 5, 2022	May 6, 2022	hirbBCceXIWUGdpNHUpaNtworjm2			
phigg@home.cern	✉	May 5, 2022	May 14, 2022	uJayGH994iVa2TY5auSNYZoUpw...			
aalcorn@home.cern	✉	May 5, 2022	May 13, 2022	5MS0kr1nE5Qm7ZNzAyHNCVfap...			
padukka.sathira@gmail.com	✉	May 3, 2022	May 3, 2022	MIUWT1Prh6boAsUB0eihfFVERW...			
sathira.padukka@gmail.com	✉	May 2, 2022	May 12, 2022	zlp8BfmXEIZ8ygj7yOgsZFo3uPt2			
alberto.rossotto@gmail.com	✉	Apr 30, 2022	May 5, 2022	j2GDTte91eYUcuHmgw0D9yIovX...			
testingperson2496@gmail...	✉	Apr 30, 2022	May 5, 2022	sexV7ZPFOxYOWLxvpdyDSADvoT...			

Rows per page: 50 1 – 7 of 7

If a new user needs to be created, the user can navigate to the landing page and enter the user email and a password and click on register, this will create a new user in firebase. After which newly created user can be used to login to the application.

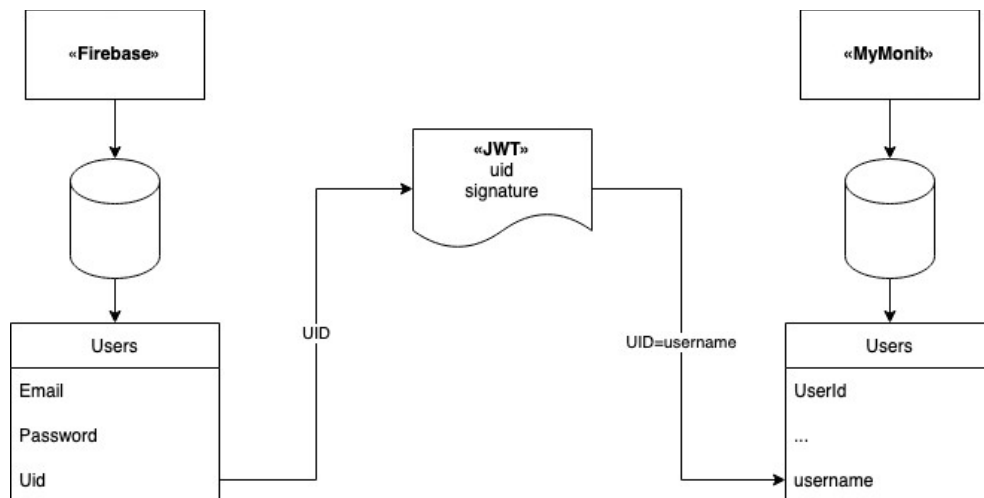
The corresponding user must be then created in APP using the User API.

```
curl --location --request POST 'http://mymonit/users/' \
--header 'Content-Type: application/json' \
--header 'Authorization: Bearer ...' \
--data-raw '{
  "user_id": "S123",
  "name": "Jim Apple",
  "username": "firebaseUid",
  "email": "jim@foobar.com",
  "role": "SCIENTIST"
}'
```

The field “username” must correspond to the User’s uid in Firebase.

Authentication process

The authentication generates a signed Json Web Token with the user’s information on the Firebase side. The token is included as Bearer token to the App’s backend in each REST API call. App’s backend verifies the signature and maps the Firebase’s Uid with App’s username. App can then use the authenticated user for the authorization of each call..



Testing

Unit testing

Unit tests cover a vast portion of the application to help detect regressions during the development.

Unit test use of Mock and MagicMock (Python Mock, N.D.) to replace the dependencies in the class under test. The mocks can return values set programmatically and make it easy to test how the class under test called them.

For example:

```
def test_insert_experiment_ok(self):  
    '''verify insert'''  
    self.user_service.is_admin = MagicMock(return_value = True)  
    self.experiment_service.insert_experiment({'name': 'abc'}, '1234')  
    self.storage.insert_experiment.assert_called_with(Experiment(experiment_id = None,  
                                                                    name = 'abc'))
```

In this test, user_service and storage are mocks. MagicMock ensures that the call to is_admin returns True in order to pass an authorization check, and the test verifies that the class under test, experiment_service, calls insert_experiment on storage with specific parameters.

Automatic scanners

The project uses Pylint (Pylint, N.D.) and Bandit (Python Code Quality Authority, 2022) to spot unused variables, formatting issues, and potential vulnerabilities. Coverage.py (Coverage.py, N.D.) measures the code coverage. It is possible to

launch the three utilities with the run-checks.sh script in containers/app with results in containers/app/reports.

Bandit

Bandit reports a possible vulnerability:

>> Issue: [B104:hardcoded_bind_all_interfaces] Possible binding to all interfaces.

Severity: Medium Confidence: Medium

This refers to the default configuration of the APP container that binds Flask on 0.0.0.0. This is not a vulnerability: the restriction on the connectivity is delegated to Docker.

Coverage

The test coverage is above 80%. The report shows precisely which lines are not covered and they are mostly lines that would not be useful to test (eg. getter) or it would require heavy usage of mocking rendering the test insignificant (eg. I/O).

Coverage report: 81%

<i>Module</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
my_monit/__init__.py	0	0	0	100%
my_monit/app.py	57	0	0	100%
my_monit/errors.py	9	3	0	67%
my_monit/experiment_service.py	45	6	0	87%
my_monit/logging.py	48	2	0	96%
my_monit/measure_service.py	12	0	0	100%
my_monit/model.py	115	7	0	94%
my_monit/rabbit_listener.py	71	12	0	83%
my_monit/rest_listener.py	157	49	0	69%
my_monit/storage.py	107	42	0	61%
my_monit/user_service.py	50	6	0	88%
Total	671	127	0	81%

Pylint

The report mentions one possible defect:

***** Module my_monit.logging

my_monit/logging.py:78:15: W0703: Catching too general exception Exception
(broad-except)

Your code has been rated at 9.98/10

It refers to a line in a method that performs logging. Although it is not a good practice, catching Exceptions aims to capture any unforeseen issue during logging and print it in stdout as a fallback.

Manual testing

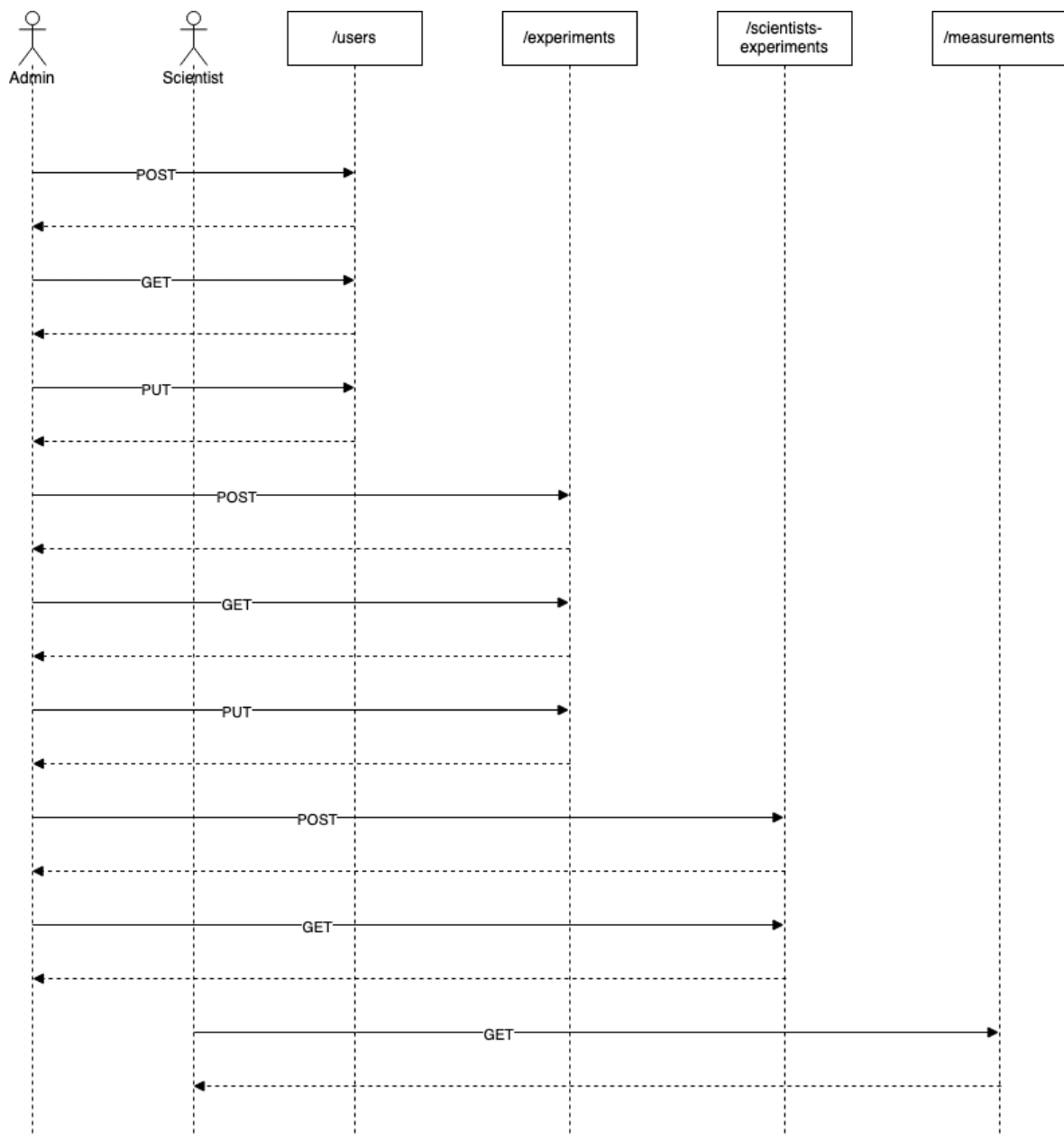
Manual testing detected the differences with the initial specifications and ensured the overall quality of the application.

The following are useful url to inspect the application when deployed with the provided test configuration.

Service	URL
Base APIs URL	https://localhost
RabbitMQ Console	http://localhost:15672/
Kibana	http://localhost:5601/login
Mysql Console	localhost:3306

APIs sequence

The following diagram illustrates a valid sequence of calls.

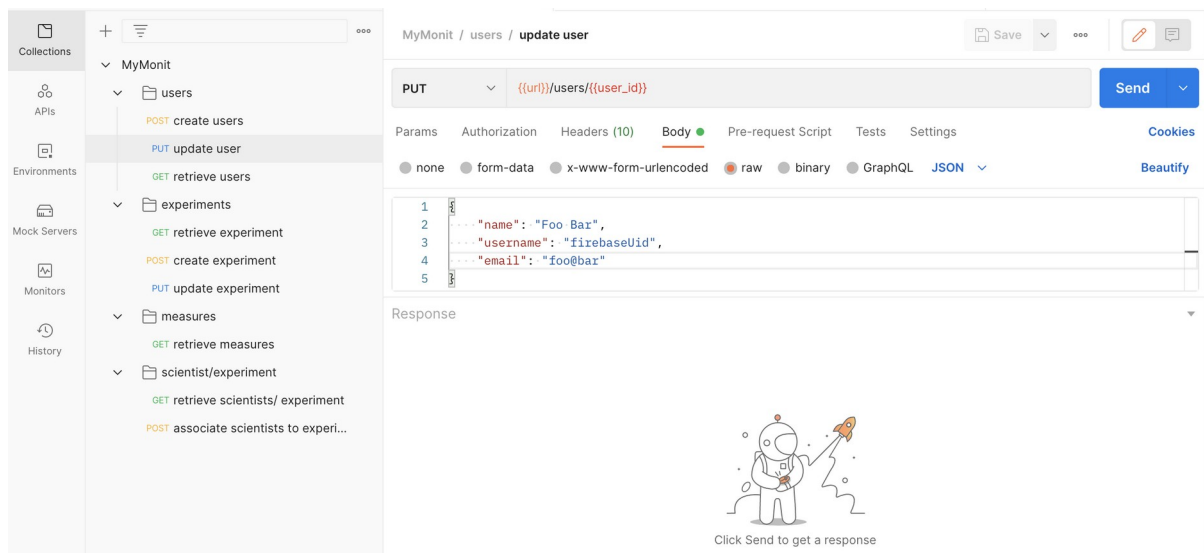


An administrator must create a User of type SCIENTIST. Subsequently, the Admin can retrieve and modify the User. Similarly, the Admin can create an Experiment.

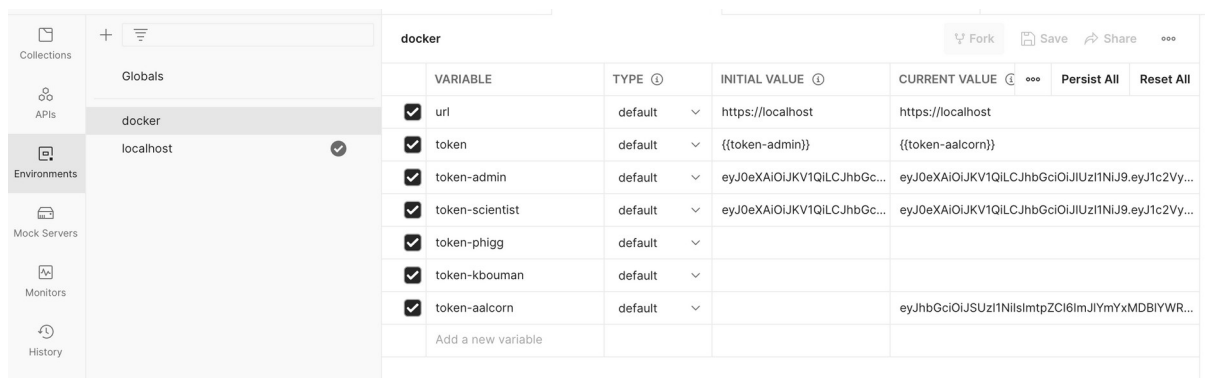
When associated, the User can retrieve the Measurements.

Calling the APIs

A collection of all the API calls is available for Postman (Postman, N.D.) between the attachments.



The collection uses variables that can be specified in the Environment tab.



API documentation

The APIs are documented in the attached api.html using Swagger (Swagger, N.D.).

users

POST /users/ create users

GET /users/ retrieve users

PUT /users/{user_id} update user

Parameters

Name	Description
Content-Type string (header)	<input type="text" value="application/json"/>
user_id * required string (path)	<input type="text" value="user_id"/>

Request body

application/json

Example Value

Schema

```
{
  "name": "Foo Bar",
  "username": "firebaseUid",
  "email": "foo@bar"
}
```

Responses

Code	Description	Links
200		No links

List of attachments

- Firebase: credentials for the Firebase console. This attachment contains credentials and should be removed when the documentation is shared with third parties.
- Postman: postman collection with examples of the APIs usage.
- Reports: reports generated with 'run-check.sh'
 - bandit.txt: report generate with bandit
 - cov: report generated with coverage
 - pylint.txt: report generated with pylint
- Swagger: description of the APIs

References

- Aimar, A., Corman, A. A., Andrade, P., Fernandez, J. D., Bear, B. G., Karavakis, E., ... & Magnoni, L. (2019). MONIT: monitoring the CERN data centres and the WLCG infrastructure. In EPJ Web of Conferences (Vol. 214, p. 08031). EDP Sciences. Available from https://www.epj-conferences.org/articles/epjconf/pdf/2019/19/epjconf_chep2018_08031.pdf [Accessed 14/May/2022]
- Coverage.py (N.D.) Coverage.py. Available from <https://coverage.readthedocs.io/en/6.3.3/> [Accessed 14/May/2022]
- Dependency Injector (N.D.) Dependency Injector — Dependency injection framework for Python. Available from <https://python-dependency-injector.ets-labs.org/> [Accessed 14/May/2022]
- Docker Compose (N.D.) Overview of Docker Compose. Available from <https://docs.docker.com/compose/> [Accessed 14/May/2022]
- Elastic (N.D.) Centralize, transform & stash your data. Available from <https://www.elastic.co/logstash/> [Accessed 14/May/2022]
- Elasticsearch (N.D.) What is Elasticsearch? Available from <https://www.elastic.co/what-is/elasticsearch> [Accessed 14/May/2022]
- Elastic (N.D.) Set up a cluster for high availability. Available from <https://www.elastic.co/guide/en/elasticsearch/reference/current/high-availability.html> [Accessed 14/May/2022]
- Firebase (N.D.) Firebase. Available from <https://firebase.google.com/> [Accessed on 14/May/2022]

- Firebase (N.D.) Add the Firebase Admin SDK to your server. Available from <https://firebase.google.com/docs/admin/setup> [Accessed 14/May/2022]
- Firebase (N.D.) Firebase Console. Available from <https://console.firebase.google.com/project/coding-output/authentication/users> [Accessed 14/May/2022]
- Flask (N.A.) Flask, web development one drop at a time. Available from <https://flask.palletsprojects.com/en/2.1.x/> [Accessed 14/May/2022]
- Hartley J. (2021) Colorama. <https://github.com/tartley/colorama> [Accessed 14/May/2022]
- Kibana (N.D.) Your window into the Elastic Stack. Available from <https://www.elastic.co/kibana/> [Accessed 14/May/2022]
- Kubernetes (N.D.) Secrets. Available from <https://kubernetes.io/docs/concepts/configuration/secret/> [Accessed 14/May/2022]
- MySQL Connector (N.D.) MySQL Connector/Python Developer Guide. Available from <https://dev.mysql.com/doc/connector-python/en/> [Accessed 14/May/2022]
- MySQL (N.D.) MySQL - The world's most popular open source database. Available from <https://www.mysql.com/> [Accessed 14/May/2022]
- MySQL (N.D.) MySQL Cluster: Getting Started. Available from <https://www.mysql.com/products/cluster/start.html> [Accessed 14/May/2022]
- Nginx (N.D.) Nginx. Available from <https://nginx.org/en/> [Accessed 14/May/2022]

- Nginx (N.D.) What Is Clustering?
<https://www.nginx.com/resources/glossary/clustering/> [Accessed 14/May/2022]
- Pika (N.D.) Introduction to Pika. Available from
<https://pika.readthedocs.io/en/stable/> [Accessed 14/May/2022]
- Pylint (N.D.) Pylint User Manual. Available from
<https://pylint.pycqa.org/en/latest/> [Accessed 14/May/2022]
- Python Code Quality Authority (2022) Bandit. Available from
<https://github.com/PyCQA/bandit> [Accessed 14/May/2022]
- Python Mock (N.D.) unittest.mock — mock object library. Available from
<https://docs.python.org/3/library/unittest.mock.html> [Accessed 14/May/2022]
- RabbitMQ (N.D.) RabbitMQ. Available from <https://www.rabbitmq.com/>
[Accessed 14/May/2022]
- RabbitMQ (N.D.) Clustering Guide. Available from
<https://www.rabbitmq.com/clustering.html> [Accessed 14/May/2022]
- Schmitz, T., Rhodes, D., Austin, T.H., Knowles, K., Flanagan, C. (2016)
Faceted Dynamic Information Flow via Control and Data Monads. In:
Piessens F., Viganò L. (eds) Principles of Security and Trust. POST 2016.
Lecture Notes in Computer Science, vol 9635. Springer, Berlin, Heidelberg.
- Rossotto, A, Ahmed, B, Padukka, S, Thompson, V, Wimalendran, P (2022)
MyMONIT - Collecting measurements to monitor CERN's experiments.