MyMONIT

Collecting measurements to monitor CERN’s experiments – README

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

Table of Contents

Application composition

Overview

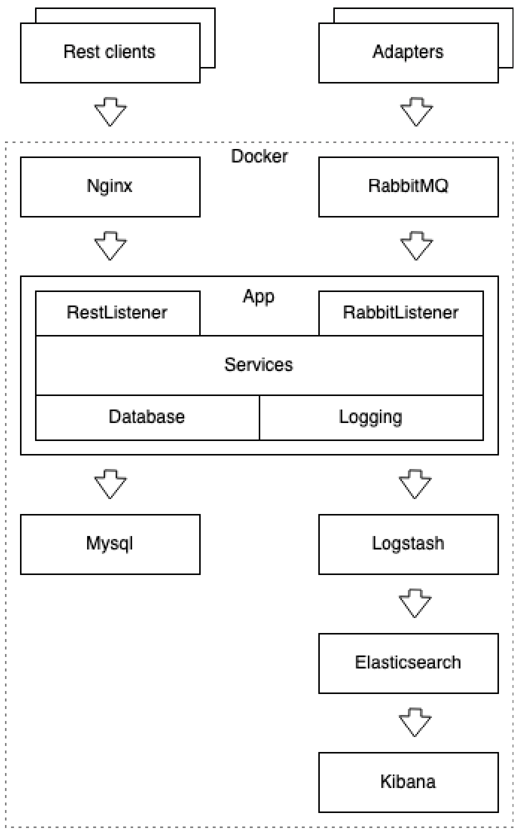
The solution is designed to deliver experimental measures via message broker and consume them via REST APIs. There is a separate interface for auditing. The solution is composed of different containers orchestrated by Docker Compose.

Components

The application is composed of seven containers running in Docker.

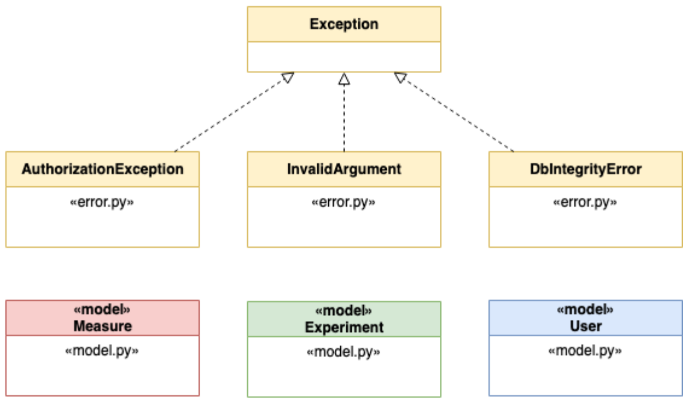
- App is the core component containing the business logic in Python. It exposes REST APIs and consumes AMQP messages.
- Mysql is the core database containing the users with access to App, and the data relative to the experiments.
- Rabbit is the AMQP message broker responsible for the collection of the measures and the delivery to App.
- Nginx is the HTTP/S proxy that exposes App's REST APIs to the external world.
- Logstash is responsible for log collection, formatting, and correct storage in Elasticsearch.
- Elasticseach is the database responsible for the collection of all the logs from all other containers.
- Kibana is the frontend application that visualizes the content of Elasticsearch.

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

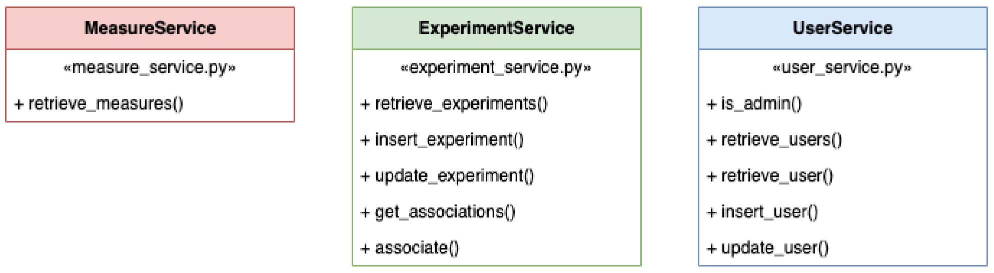


Class diagram of APP

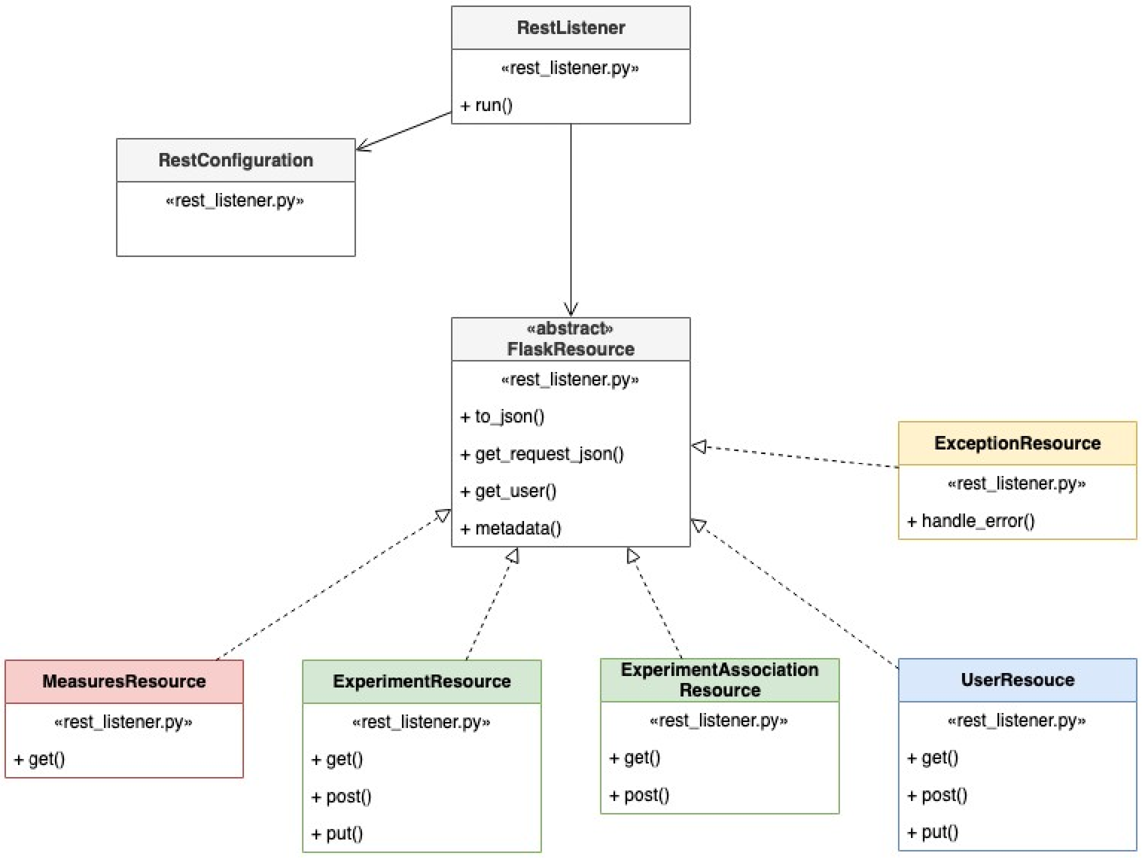
The model is composed of Measure, Experiment, User, and Exceptions.



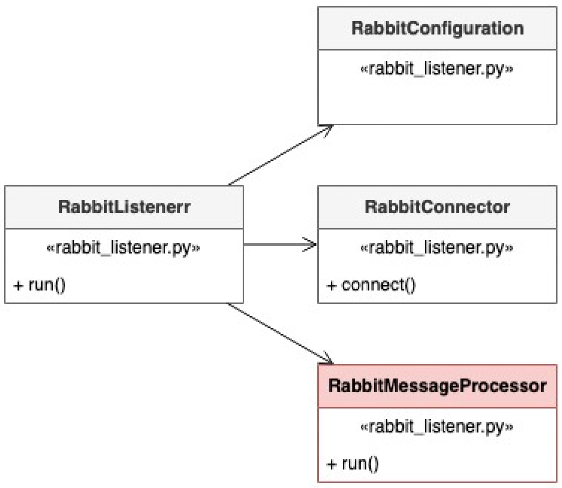
A specialised service is responsible for the three main elements of the model.



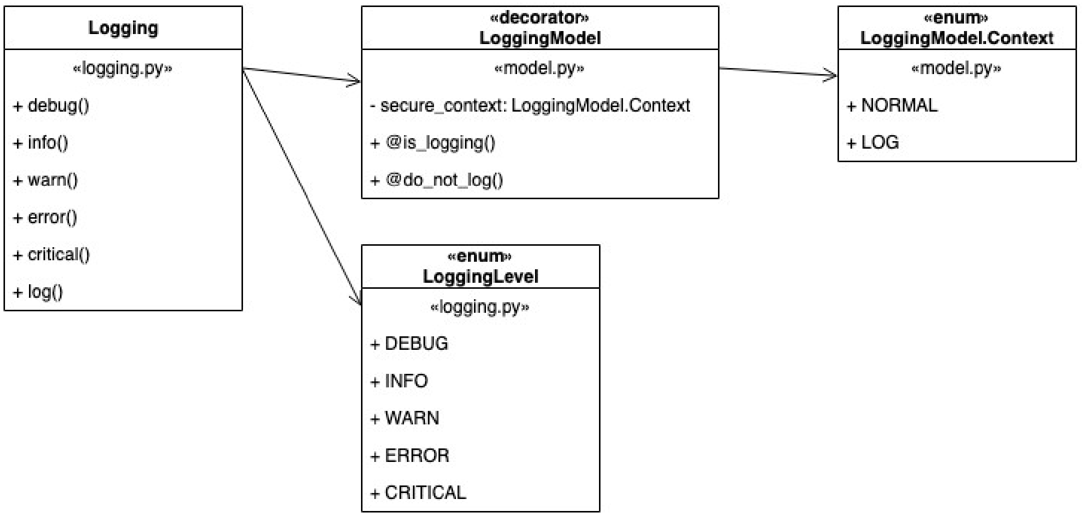
RestListener is responsible for the REST interface of the application. RestListener uses an object for the configuration, and delegates the handling of the resources to specialised components.



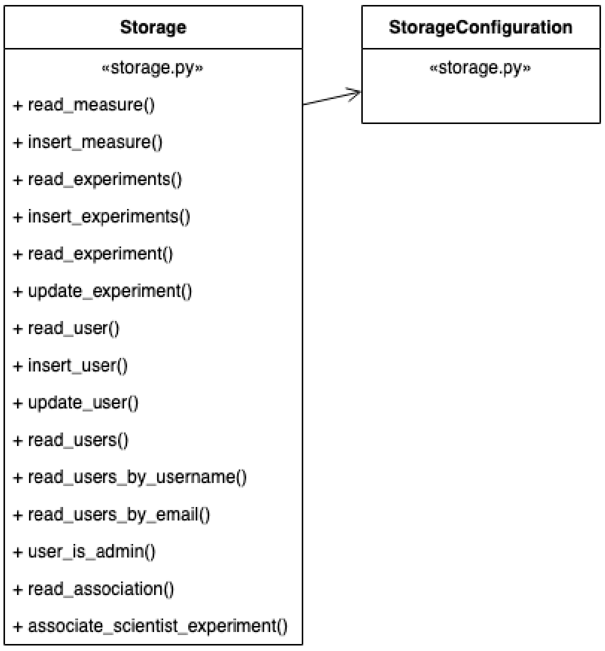
RabbitListener is the service responsible for the connection to RabbitMQ and it uses a RabbitConfiguration to group the configuration. RabbitConnector is a component of the listener used to extract the IO functions and simplify unit testing. RabbitMessageProcessor contains the logic to store the measures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



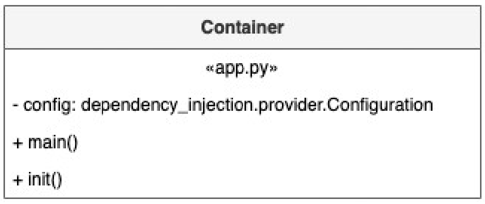
Logging is a cross-cutting service used in different components. It supports the two decorators @is_logging and @do_not_log to respectively mark functions that perform logging, and parts of the model that should not appear in the logs with their real value to prevent data leaks.



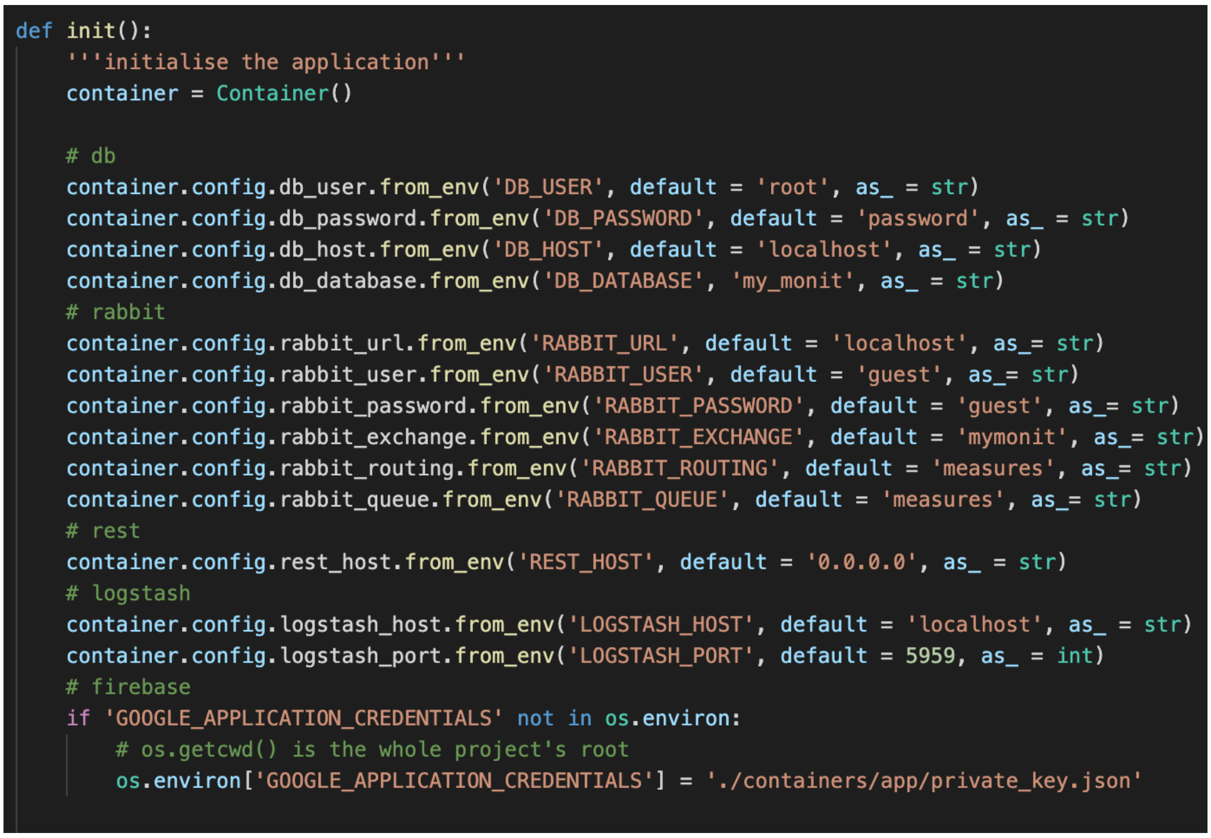
Storage is responsible for the IO with the database.



Container is responsible for the configuration and dependency injection. Thanks to Container, components do not instantiate one another simplifying unit testing.



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

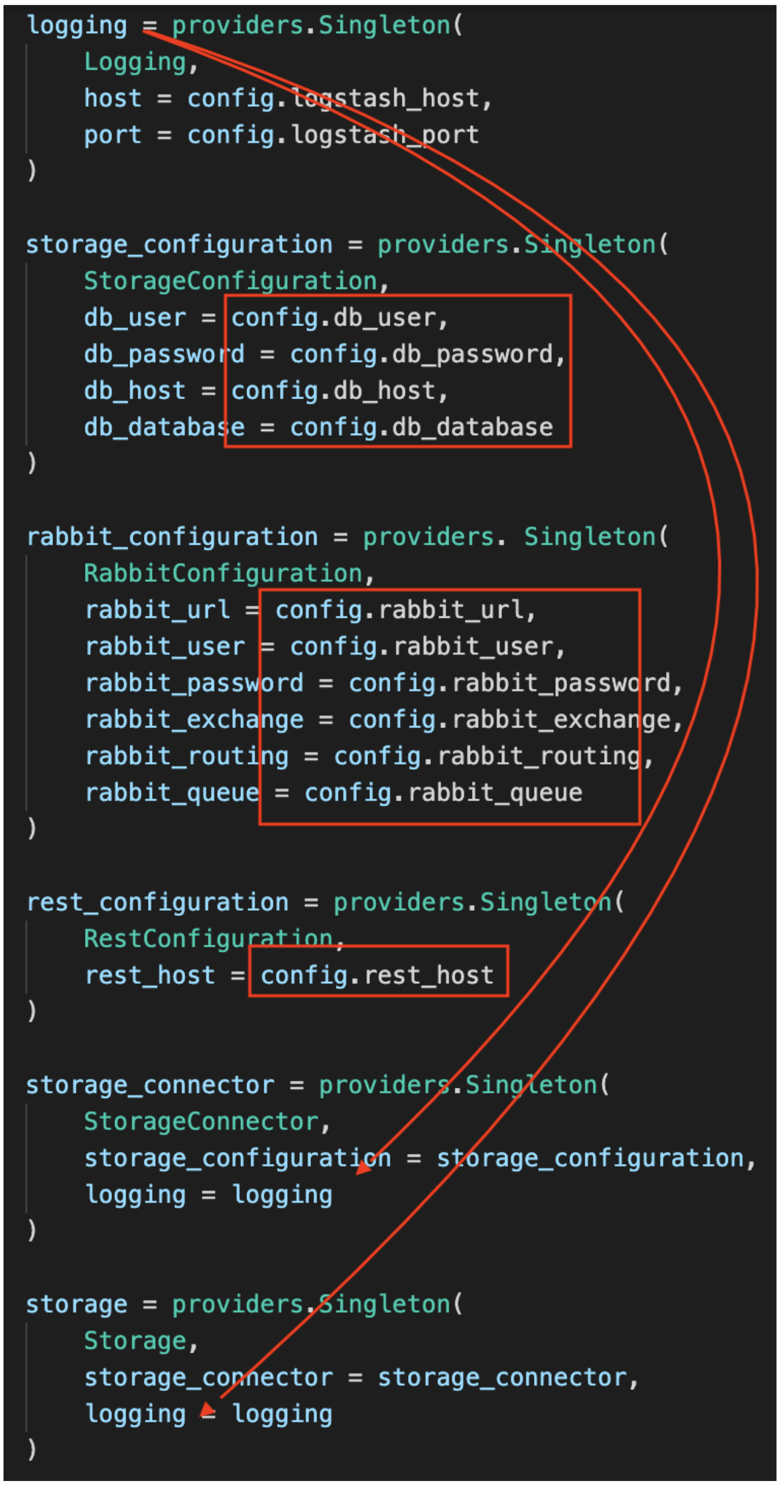


```python
def init():
    '''initialise the application'''
    container = Container()

    # 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.

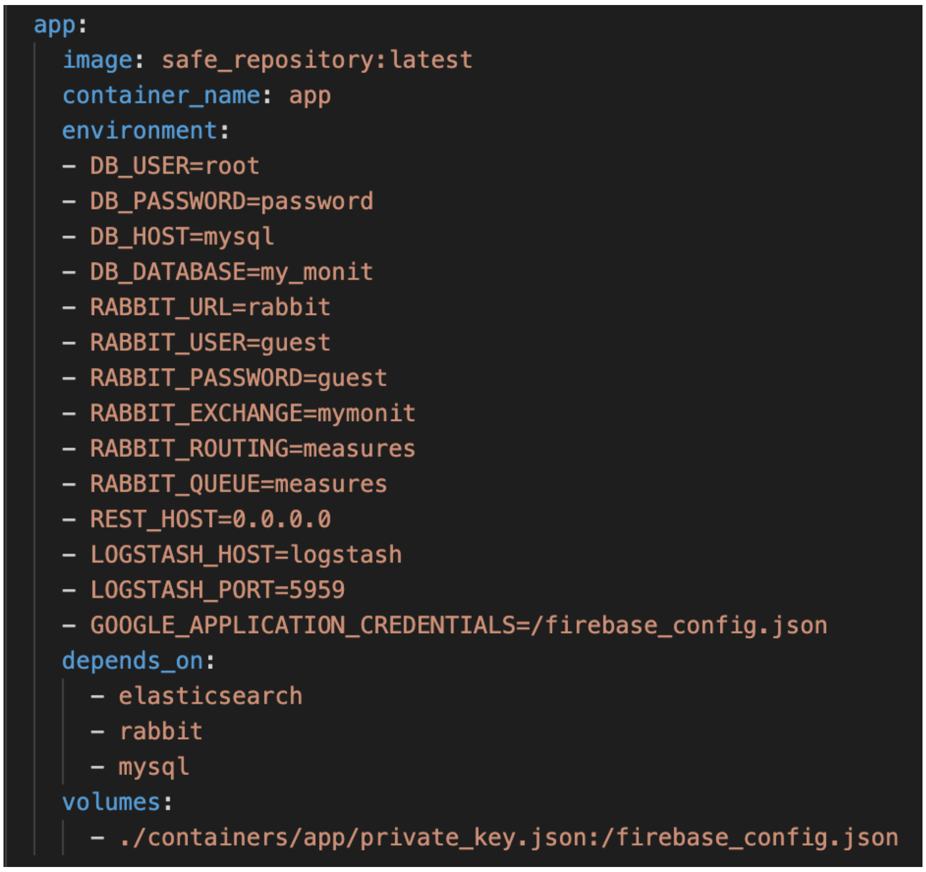
Dependency injection is essential to perform accurate unit testing because it makes object-mocking easy.



```
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
)
```

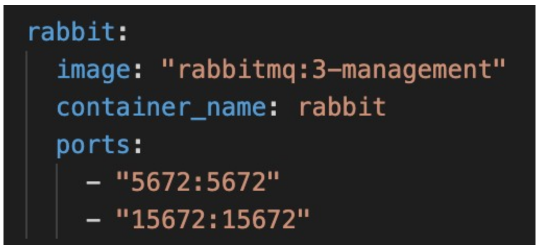
Orchestration

The orchestration is described in docker-compose.yml.



```yaml
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 recommended to change them 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.



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

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
Mysql	Mysql:5.7	None
Elasticsearch	Elasticsearch:6.5.4	None
Kibana	Kibaba:6.5.4	None

Container’s name	Base image	Customisation
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. The core Python application is stateless and it is possible to start multiple instances in parallel to improve performances. All the other components natively support clustering through configuration.

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

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

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 generates a signed Json Web Token with the user’s information.

Such token is verified in every API call to verify authorizations.

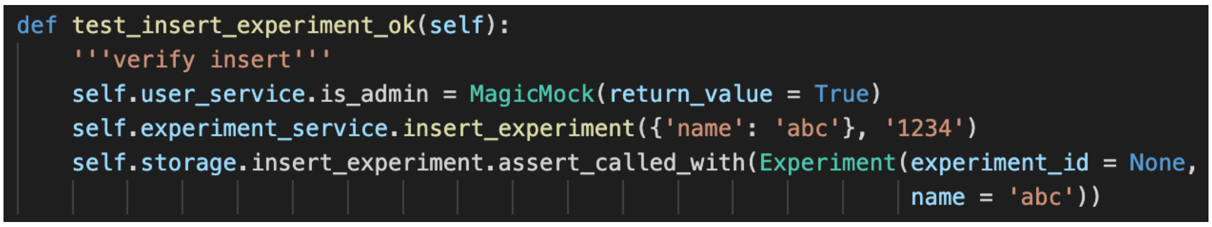
Testing

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 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:



```python
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

Pylint and Bandit spot unused variables, formatting issues, and potential vulnerabilities. Coverage 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

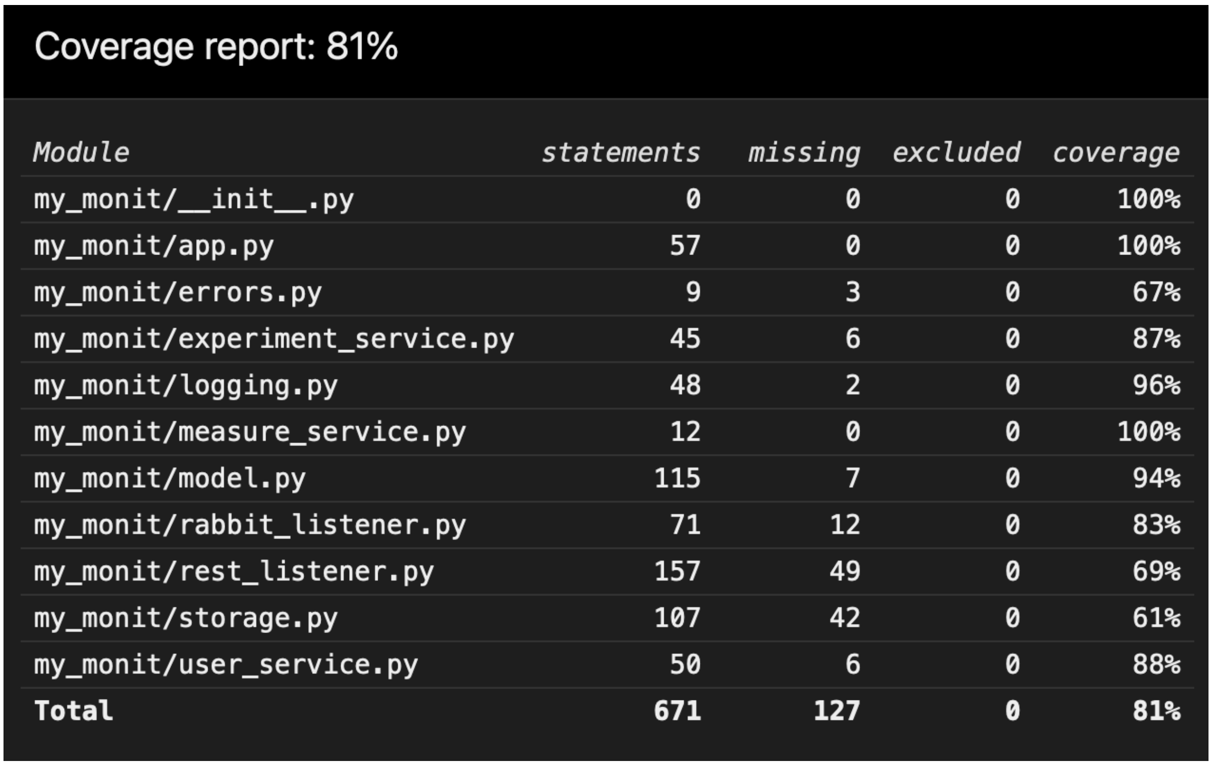
Bandit reports a possible vulnerability:

```
>> 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%

Module	statements	missing	excluded	coverage
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

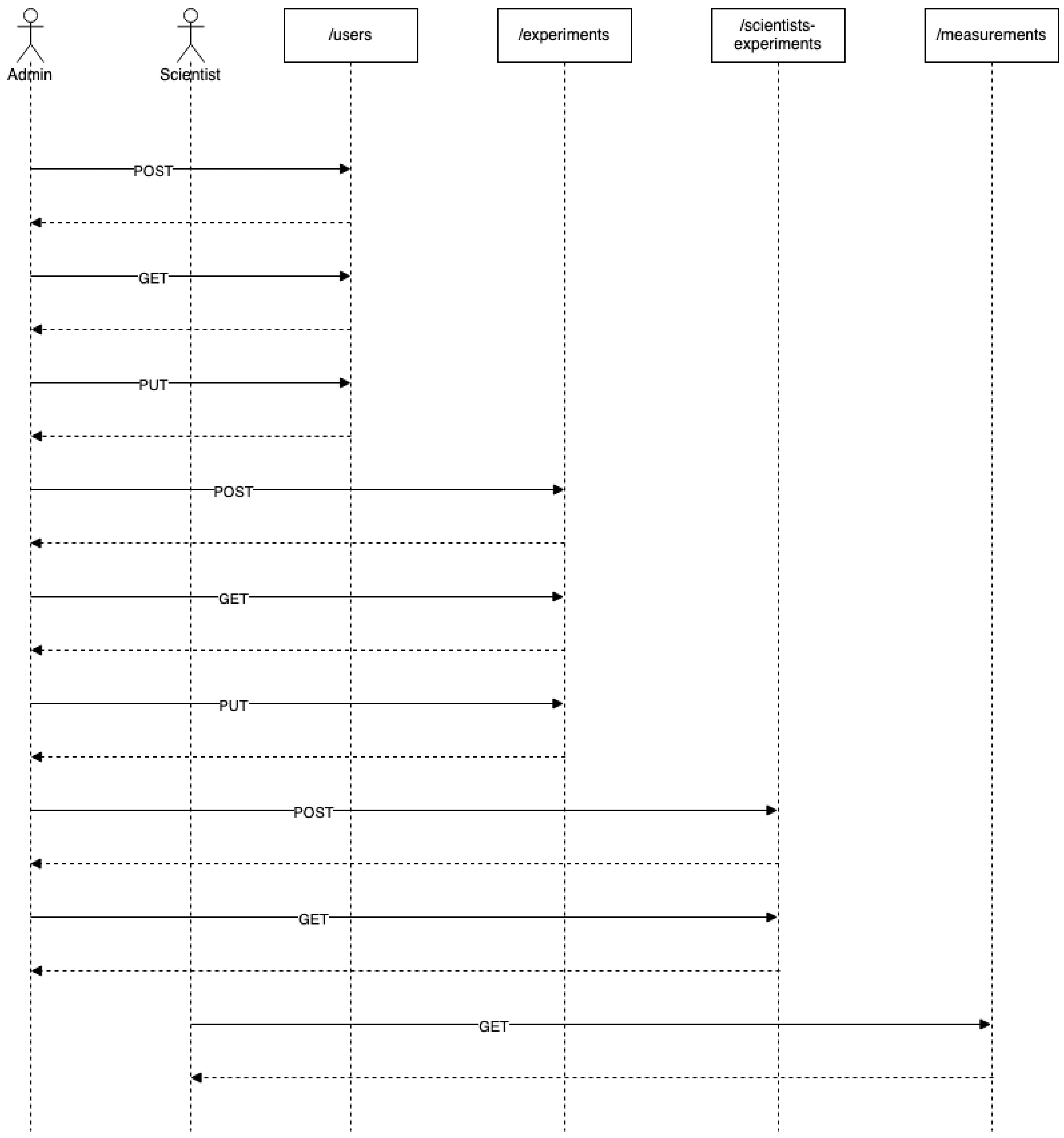
Manual testing detected the differences with the initial specifications and ensured the overall quality of the application.

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.



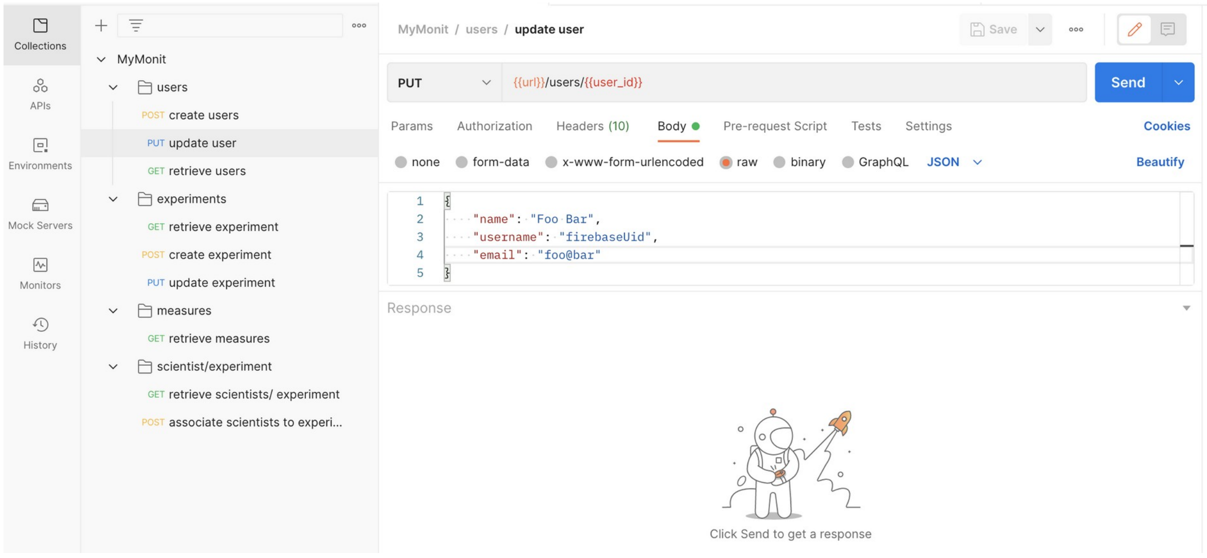
An administrator must create a User of type SCIENTIST. Subsequently, the Admin can retrieve and modify the User. Similarly, the Admin can create an Experiment.

With at least one User and one Experiment, it is possible to associate them to give access to the Experiment.

When associated, the User can retrieve the Measurements.

Calling the APIs

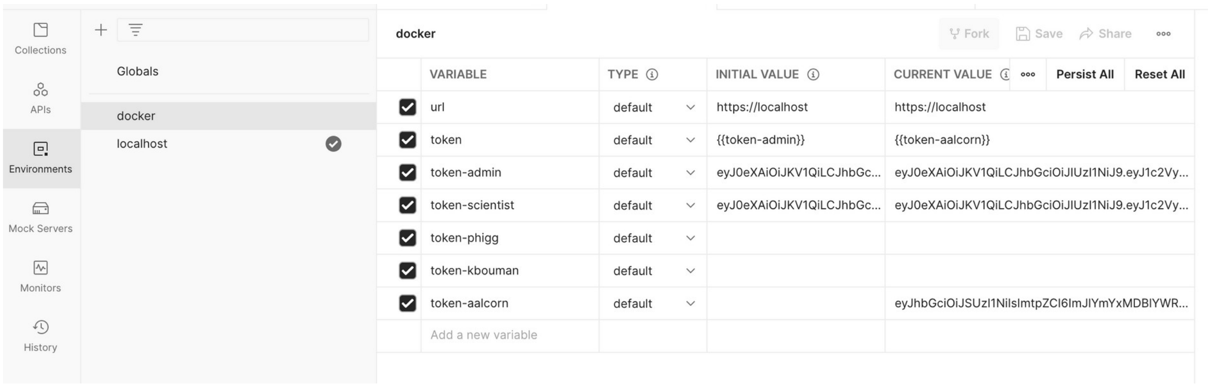
A collection of all the API calls is available for Postman in the postman folder.



```
{{url}}/users/{{user_id}}
```

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

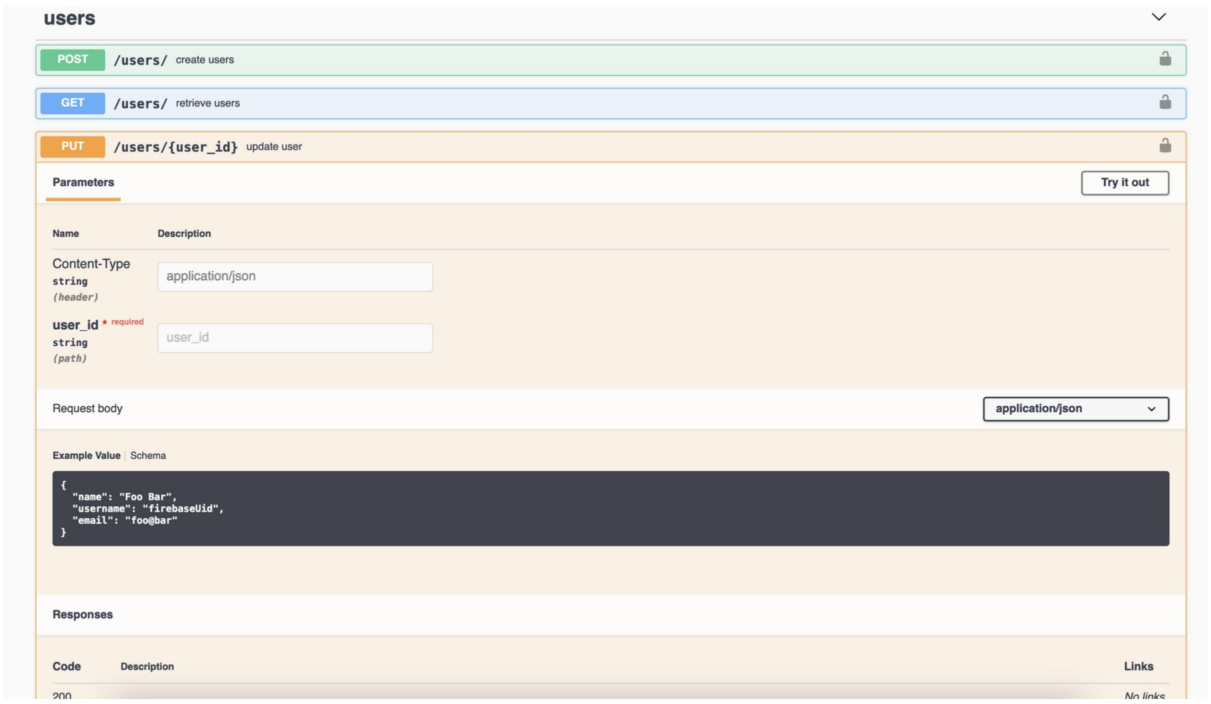
The collection uses variables that can be specified in the Environment tab.



	VARIABLE	TYPE	INITIAL VALUE	CURRENT VALUE
✓	url	default	https://localhost	https://localhost
✓	token	default	{{token-admin}}	{{token-aalcorn}}
✓	token-admin	default	eyJ0eXAiOiJKV1QiLCJhbGc...	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2Vy...
✓	token-scientist	default	eyJ0eXAiOiJKV1QiLCJhbGc...	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2Vy...
✓	token-phigg	default		
✓	token-kbouman	default		
✓	token-aalcorn	default		eyJhbGciOiJSUzI1NiIsImtpZCI6ImJlYmYxMDBlYWR...
	Add a new variable			

API documentation

The APIs are documented in the attached api.html using Swagger.


users

POST /users/ create users

GET /users/ retrieve users

PUT /users/{user_id} update user

Parameters

Try it out

Name	Description
Content-Type string (header)	application/json
user_id * required string (path)	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

- 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