# **Unit Testing**

Contents

[**Unit Testing** 1](#_Toc195228499)

[DatabaseManger Class Testing 2](#_Toc195228500)

[test\_get\_connection\_success 2](#_Toc195228501)

[test\_get\_connection\_failure 2](#_Toc195228502)

[FlaskEndpoints Class Testing 2](#_Toc195228503)

[test\_successful\_login\_flow 2](#_Toc195228504)

[AuthService Class Testing 2](#_Toc195228505)

[Registration Tests 2](#_Toc195228506)

[Login Tests 2](#_Toc195228507)

[User Preferences 2](#_Toc195228508)

[AdminService Class Testing 2](#_Toc195228509)

[User Management Tests 2](#_Toc195228510)

[System Configuration Tests 2](#_Toc195228511)

[User Preferences 2](#_Toc195228512)

[DataService Class Testing 2](#_Toc195228513)

[Data Retrieval Tests 2](#_Toc195228514)

[Data Filtering and Formatting Tests 2](#_Toc195228515)

[SimulationService Class Testing 2](#_Toc195228516)

[Sensor Reading Generation Tests 2](#_Toc195228517)

[Timezone and Simulation Tests 2](#_Toc195228518)

[Database Interaction Tests 2](#_Toc195228519)

[Mocking and Threading Tests 2](#_Toc195228520)

[LogService Class Testing 2](#_Toc195228521)

[Log Level Tests 2](#_Toc195228522)

[Log Persistence Tests 3](#_Toc195228523)

Our team has implemented comprehensive unit testing for our flask-based dashboard API. We have focussed our testing on each class component independently as their own python file but still called and executed from one central run\_all\_app.py

We used Python’s unittest framework which has the benefit of being able to mock the database and use separate instances during testing without relying on/ risking using the original database.

We also use the real database for testing components to accurately test the Flask API

The unit testing was separated into 7 files each for the 7-unit testing files which were created to test each of the 7 main classes for our Flask API setup.

All of the unit testing classes setup by using a mock configuration when testing the database.

## DatabaseManger Class Testing

### test\_get\_connection\_success

This test checks if the database connects successfully when the config credentials are all correct. We verify the connection by making sure all the important connection details are correct:

The right host, username, password, and database name are being used

This ensures that when our system tries to connect to the database, it's using exactly the configuration we expect it to use.

### test\_get\_connection\_failure

This test checks what happens if the connection fails and makes sure it returns a correct "None" rather than crashing. This is very important as we need our system to handle database problems gracefully. When the database isn't available, our system should:

Return `None` instead of crashing

This way, other parts of our system can handle the situation appropriately.

## FlaskEndpoints Class Testing

This class test particularly focuses on verifying our API endpoints are working as intended with a focus on the login functionality.

This setup includes the usual mock database but also a test client to mock requests

### test\_successful\_login\_flow

This test checks if users can log in successfully with valid credentials. The test creates a mock user with specific details like email, password, and admin status. It then attempts to log in with matching credentials and verifies that:

* The login request returns a successful status code (200)
* The system accepts valid login credentials

The test uses a mock user with these credentials:

* Email: [test@example.com](mailto:test@example.com)
* Password: ValidPass123!
* Admin status: Active
* Account status: Active

These mock user details are just that as they do not conform to our “rakusens” validation. This is because our validation is done in JavaScript and therefor not applicable in backend unit testing.

## AuthService Class Testing

This class test verifies user authentication and account management functions, including registration login and changing preferences.

This setup includes the usual mock database but also two types of users, inactive and active with various test credentials

### Registration Tests

These tests verify the registration capabilities of the API

#### test\_successful\_registration

This test method verifies new users can create an with valid details in which passwords are properly hashed, the correct success response is asserted and the event is logged accurately

#### test\_register\_with\_existing\_email

This test method ensures duplicate email addresses are handles properly by starting with an attempt to register with an existing email address then verifying the correct error message is returned and that the event is not logged

### Login Tests

#### test\_login\_with\_invalid\_password

This test method checks if a user can login with incorrect passwords and subsequently the correct error messages are returned, as well as the correct logging

#### test\_login\_with\_inactive\_account

This test method verifies the systems handling of deactivated accounts by attempting a log in with an active:0 account and checking if appropriate error messages and logging are returned

### User Preferences

#### test\_update\_dark\_mode\_toggle

This test method checks the dark mode preference functionality by first updating the users dark mode setting. Verifying query construction for the TestAuthService class method where the query is constructed with variables to execute. The test also confirms changes are logged and a successful response is returned

## AdminService Class Testing

This class test verifies administrative functionality focussing on user management and system configuration, ensuring administrators can effectively manage users and sensors

### User Management Tests

#### test\_toggle\_admin\_status

This method tests the ability to change the user’s admin privileges confirming the database updates and the correct logging of the action

#### test\_delete\_user\_success

This test method validates user deletion by testing proper database query execution, correct logging and return of the correct response for the action

#### test\_update\_user\_details

This test method tests the user information updates on the system by validating email and name updates. The method checks the query construction, logging and also ensures no duplicate emails are updated

### System Configuration Tests

#### test\_login\_with\_invalid\_password

This test method checks if a user can login with incorrect passwords and subsequently the correct error messages are returned, as well as the correct logging

#### test\_login\_with\_inactive\_account

This test method verifies the systems handling of deactivated accounts by attempting a log in with an active:0 account and checking if appropriate error messages and logging are returned

### User Preferences

#### test\_update\_dark\_mode\_toggle

This test method checks the dark mode preference functionality by first updating the users dark mode setting. Verifying query construction for the TestAuthService class method where the query is constructed with variables to execute. The test also confirms changes are logged and a successful response is returned

## DataService Class Testing

This test class primarily focuses on the different types of data retrieval, filtering and sensor data management to ensure interactions with the database and supplying to the front end is correctly done in the expected format and structure.

### Data Retrieval Tests

#### test\_historical\_data\_retrieval

this test method tests the retrieval of historical data for a chosen specified line and makes sure it is fetched and returns the right data from the database

#### test\_get\_logs

this test method checks if the correct system logs are retrieved. It also confirms the correct sql query was executed

#### test\_get\_forecasted\_data

this test method checks the confirmation of the retrieving forecasted data for a given line and also if it falls within the expected range

#### test\_get\_sensor\_data

this test method checks the retrieval of data for a single sensor e.g r01 from a certain line while also making sure its in the correct format

### Data Filtering and Formatting Tests

#### Test\_historical\_data\_filtering

This test method checks the that historical data that is retrieved can be filtered by date and sensor values. Also making sure the correct SQL statement is generated for filtering the data and assuring the correct structure

#### Test\_live\_data\_format

This test method validates the live data format and structure making sure the retrieved live data is has the right timestamps and sensor values(making sure they are valid).

## SimulationService Class Testing

This class test ensures that simulated sensor data are generated correctly and also the management of simulation functionalities like threading and data insertion

### Sensor Reading Generation Tests

#### Test\_sensor\_reading\_generation

This method test validates the generated data as correct within the predefined sensor ranges. Making sure the reading is a floating point number which stays within the defined min and max values.

#### Test\_multiple\_sensor\_readings

This test ensures multiple sensor readings can be generated for multiple sensors e.g r01,r02,r03 at the same time. This again make sure they are within the min and max values and also the data format are consistent and it behaves normally across multiple sensors without it going off course.

#### Test\_reading\_range\_validation

This test method once again like in other tests ensures all generated sensor reading are within the expected ranges(between the min and max limits). This is done for all sensors as none should be out of the limits.

### Timezone and Simulation Tests

#### Test\_timezone\_offset

This test method depending on the month calculates the correct timezone offsets. For months April to Oct it should be +01 hour and for Nov to March should be +00 hour.

#### Test\_simulation\_start\_stop

This method checks if the start and stop of the simulation thread is working as intended by starting with a new thread when start is called making sure the expected parameters are there. The stop method is tested just to check if there is no errors as it stops itself automatically due to being a daemon thread

### Database Interaction Tests

#### Test\_insert\_line\_readings

This test method verifies the correct insertion of the generated sensor readings. Also verifying the correct SQL INSERT query is built with the correct structure. This is validated by comparing the actual query with the expected query.

### Mocking and Threading Tests

Throughout the tests methods such as threading.thread and time.localtime are patched at the top of the methods to simulate the different conditions and everything is correctly handled and behaves as expected

## LogService Class Testing

This test class ensures that logging functionality is added correctly. Specifically on how messages are logged, insertion of logs into the database and the formatting

### Log Level Tests

#### Test\_logs\_levels

This test method checks the validates the handling of different log levels(e.g INFO, WARNING, ERROR etc).it checks that different log messages are logged with the correct level and type and also the database query for logging is formatted correctly using normalized spaces. Then finally a correct insertion of log entry into the database pairing with a commit() on the database connection.

#### Test\_log\_formatting

This test method simply ensures that log messages are formatted correctly by checking the INSERT query and if the correct parameters are passed to it (log level,type and message). And if all that is formatted correctly so it is persisted into the database.

### Log Persistence Tests

#### test\_log\_persitence

this test method ensures the log message is stored in the database. it checks the correct query is executed for each log message. Correct parameters are passed through to the query and they match the expected values. Also making sure the log messages have the appropriate log level, type, and message paired with a commit() on the database connection when committed to the database