# **Unit Testing**

Contents

[**Unit Testing** 1](#_Toc195220410)

[DatabaseManger Class Testing 2](#_Toc195220411)

[test\_get\_connection\_success 2](#_Toc195220412)

[test\_get\_connection\_failure 2](#_Toc195220413)

[FlaskEndpoints Class Testing 3](#_Toc195220414)

[test\_successful\_login\_flow 3](#_Toc195220415)

[AuthService Class Testing 3](#_Toc195220416)

[Registration Tests 3](#_Toc195220417)

[Login Tests 3](#_Toc195220418)

[User Preferences 4](#_Toc195220419)

[AdminService Class Testing 4](#_Toc195220420)

[User Management Tests 4](#_Toc195220421)

[System Configuration Tests 4](#_Toc195220422)

[User Preferences 5](#_Toc195220423)

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