CovaCare Code Testing

Brydon Herauf (200454546) & David Kim (200405213)

Faculty of Engineering & Applied Science, University of Regina

March 13, 2024

Test Plan

The goal of CovaCare's code testing is to develop confidence that our solution works as expected. We plan to achieve this by covering all functionality that our system provides. We will do this by writing unit and integration tests for the real-time processing module, mobile application, alerting service, and API. Additionally, we will test our fall detection model against a test set with 1,100 samples. To confirm that the system is working as a whole, we will conduct a few high level manual system tests. Finally, user acceptance testing will be performed against our complete product to verify that it is meeting our user's needs. This document will outline our testing for each part of the system. For a more detailed look, see the repository's test code.

Mobile Application

Procedure: Our UI testing was conducted using Jest and React Testing Library for both unit and integration tests. We wrote cases for each component, and then the app as a whole.

Test Cases:

Unit Tests:

- CameraListItem: Tests rendering, ensuring it displays camera information correctly and handles selection and deletion actions.
- ContactListItem: Tests rendering, ensuring it displays contact information correctly and handles selection and deletion actions.
- CameraForm: Ensures correct rendering for new cameras, validating required fields, and calling the onSave function with the correct data when the form is valid.
- ContactForm: Ensures rendering of the necessary fields, validating input, and that it correctly handles saving and editing existing contacts.
- ToggleField: Tests rendering, checks if it handles value changes correctly, and verifies the initial value display.

- DetectionSection: Validates rendering, showing time inputs when enabled, and displaying sensitivity and duration fields when specified.
- FormField: Tests rendering, checking for text changes, error state display, and secure text entry functionality.
- Card: Verifies that it correctly renders the title, description, and child content.
- SliderField: Tests rendering, checks if it handles value changes correctly, and verifies that the current value is displayed as a percentage.

Integration Tests:

- Tests the integration of the CameraScreen component, ensuring that it renders correctly and handles camera operations including adding, updating, and deleting entries.
- Tests the integration of the ContactScreen component, ensuring that it renders correctly and handles contact operations including adding, updating, and deleting entries.

Results:

```
PASS __tests__/ContactForm.test.tsx
PASS __tests__/DetectionSection.test.tsx
PASS __tests__/CameraForm.test.tsx
PASS __tests__/TimeInputField.test.tsx
PASS __tests__/ContactListItem.test.tsx
PASS __tests__/FormField.test.tsx
PASS __tests__/CameraListItem.test.tsx
PASS __tests__/SliderField.test.tsx
PASS __tests__/helpers/setupIntegrationTests.ts
PASS __tests__/Card.test.tsx
PASS __tests__/ToggleField.test.tsx
PASS __tests__/App.integration.test.tsx

Test Suites: 12 passed, 12 total
Tests:       40 passed, 40 total
Snapshots:   0 total
Time:        2.928 s, estimated 4 s
Ran all test suites.
```

Testing these components led to minor bug discovery.

API Testing

Procedure: Our API testing was conducted using pytest. We wrote unit tests for individual routes, and integration tests to verify database interaction.

Test Cases:

Unit Tests:

- Get Cameras (GET /cameras)
    - Verifies that the endpoint correctly retrieves a list of cameras.
    - Mocks the database query to return predefined camera data.
- Get Camera by ID (GET /cameras/<id>)
    - Ensures that retrieving a camera by its ID returns the correct details.
    - Mocks the database query to return a specific camera.
- Add Camera (POST /cameras)
    - Tests the ability to add a new camera and verifies the response status and data.
    - Mocks the database interaction to simulate adding a camera.
- Update Camera (PUT /cameras/<id>)
    - Ensures that updating an existing camera correctly modifies its details and returns the expected status.
    - Mocks the database query to simulate the update process.
- Delete Camera (DELETE /cameras/<id>)
    - Verifies that deleting a camera removes it from the system and returns a success response.
    - Mocks the database interaction to simulate the deletion.
- Get Contacts (GET /contacts)
    - Verifies that the endpoint correctly retrieves a list of contacts.
    - Mocks the database query to return predefined contact data.
- Get Contact by ID (GET /contacts/<id>)

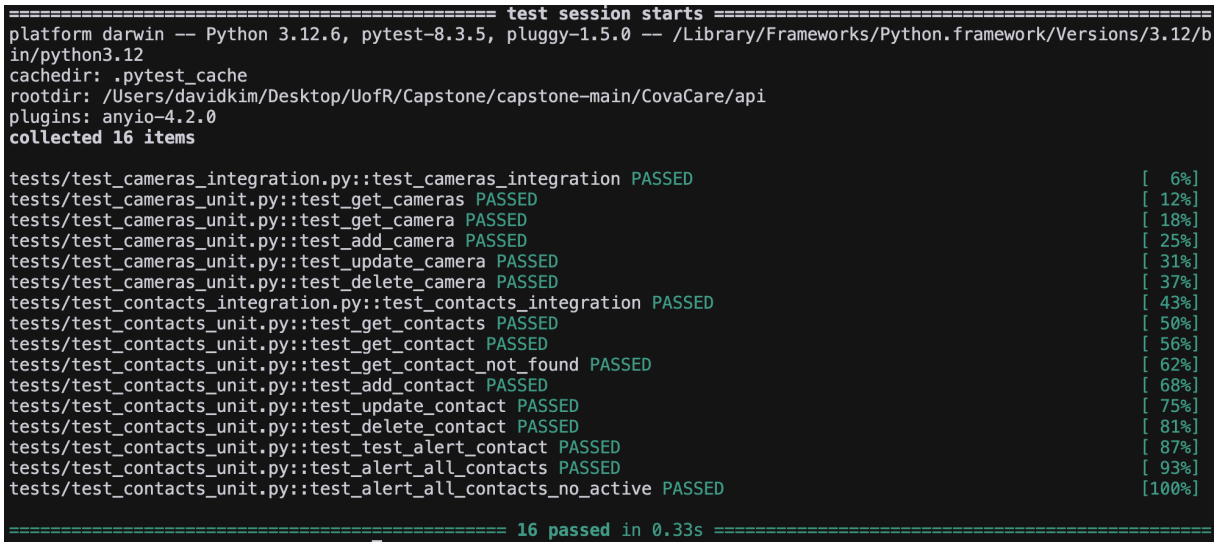
    - Ensures that retrieving a contact by ID returns the correct details.
    - Mocks the database query to return a specific contact.
- Add Contact (POST /contacts)
    - Tests the ability to add a new contact and verifies the response status and data.
    - Mocks the database interaction to simulate adding a contact.
- Update Contact (PUT /contacts/<id>)
    - Ensures that updating an existing contact modifies its details correctly and returns the expected status.
    - Mocks the database query to simulate the update process.
- Delete Contact (DELETE /contacts/<id>)
    - Verifies that deleting a contact removes it from the system and returns a success response.
    - Mocks the database interaction to simulate the deletion.
- Test Alert Contact (POST /contacts/<id>/test-alert)
    - Ensures that the alert functionality sends a notification to a specific contact and returns a success message.
    - Mocks the alert service to simulate sending an alert.
- Alert All Contacts (POST /contacts/alert-all)
    - Verifies that alerts are sent to all active contacts and a success response is returned.
    - Mocks the alert service to simulate sending alerts.

Integration Tests:

- Cameras Integration: Tests the full lifecycle of camera operations, including adding, retrieving, updating, and deleting cameras. It ensures that the database reflects the changes made through the API.

- Contacts Integration: Tests the full lifecycle of contact operations, including adding, retrieving, updating, and deleting contacts. It ensures that the database reflects the changes made through the API and verifies alert functionality.

Results:



No bugs were discovered during API testing, but it verified that everything was working as expected.

Alerting Service

Procedure: Our alerting service testing was conducted using pytest to ensure that the alert functionality works as expected. The service is quite small, so we only wrote a few unit tests, and our system testing covers its integration with other parts of the system.

Test Cases:

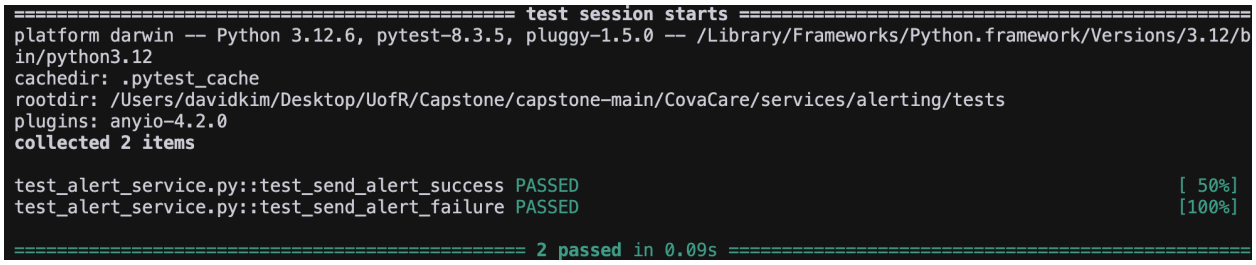
Unit Tests:

- Send Alert Success: Tests the send_alert method of the AlertService class to ensure it successfully sends an alert message using the Twilio client. It mocks the Twilio client to

verify that the correct parameters are passed and that the expected message SID is returned.

- Send Alert Failure: Tests the send_alert method of the AlertService class to ensure it raises an exception when the Twilio client fails to send a message. It mocks the Twilio client to simulate an error scenario.

Results:



No bugs were discovered during this testing.

Real-time Processing Module

Procedure: Our real-time processing module was tested using pytest. We wrote a series of unit tests and integration tests.

Test Cases:

Units Tests:

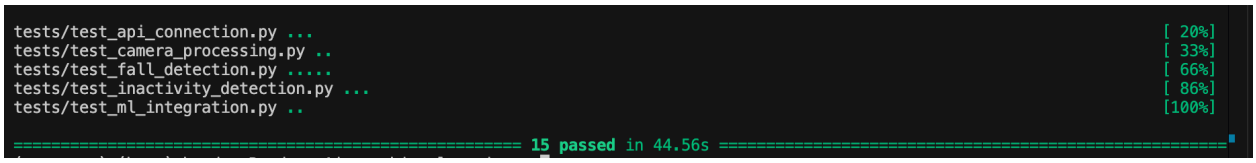
- Test Format Stream URL: Tests the format_stream_url function to ensure it correctly formats the stream URL using the provided camera credentials. It verifies that the output matches the expected RTSP URL format.
- Test Is Within Time Range: Tests the is_within_time_range function to ensure it accurately determines whether the current time falls within a specified start and end time range. It checks the function's output against expected boolean values.
- Test Api Polling: Tests the poll_cameras function with a mocked API response.

- Test Alert Active Contacts: Tests the alert_active_contacts function with both success and failure mocked API responses.
- Test Evaluate Window: Tests the evaluate_window method of the FallDetector class to ensure it correctly evaluates a window of keypoints data and returns the expected results based on the input.
- Test Check History for Falls: Tests the check_history_for_falls method of the FallDetector class with various patterns in the prediction history to verify that it correctly identifies fall patterns.
- Test Inactivity Detection No Movement: Tests the check_inactivity method of the InactivityMonitor class to ensure it returns False when there is no movement between frames.
- Test Inactivity Detection Movement: Tests the check_inactivity method to ensure it returns False when there is movement between frames, but not enough time has passed to flag it.
- Test Inactivity Detection After Duration: Tests the check_inactivity method to ensure it returns True after the inactivity duration has passed without movement.

Integration Tests:

- Fall Detected From Video: This test passes a video into the processing script, and verifies if a fall is appropriately detected.
- Inactivity Detected From Video: This test passes a video into the processing script, and verifies if inactivity is appropriately detected.

Results:

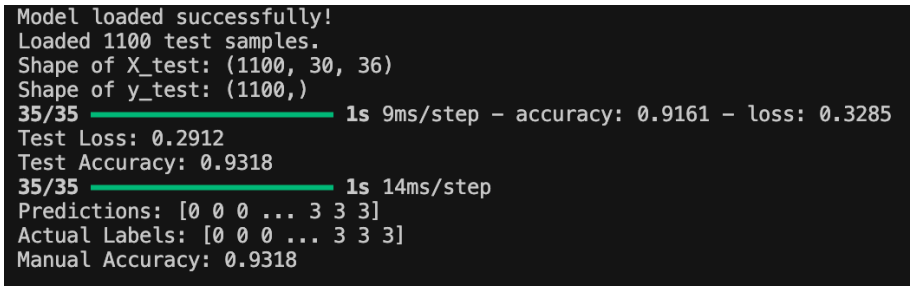


A bug with the inactivity detection algorithm was discovered and fixed with this testing.

Fall Detection Model Testing

Procedure: To test the fall detection model itself, we used a test set of 1100 samples, containing 275 30-frame segments of each class. The data is from the same five datasets we used to train our model, but these clips were not used during that process.

Results:



The model achieved an accuracy of approximately 93%. This was just on classifying 30-frame clips as non-fall, start of fall, full fall, or end of fall. The full fall detection process combines this model and an algorithm that looks at the history of model predictions.

System Testing

To ensure that all parts of CovaCare are working together, we have conducted some high-level manual system tests. We accomplished this by running the API, UI, and real-time processing module, and setting up a wifi camera. Once everything was running, we entered the details for the camera in the application, and it started processing. We then performed the following tests:

- Change the active hours, inactivity time before alert, and other settings in the application and verify that they are received in the processing module.
- Input our phone numbers in the application as emergency contacts, and set one to active.
- Walk by the camera performing various motions, including sitting, kneeling, bending over, and falling. Verify if alerts are received, and which number(s) gets them.

- Repeat testing in various rooms and lighting conditions.

After numerous attempts, and bug fixes, all of these tests were a success. Bugs discovered and fixed during the process were:

- RTSP URL formatting was wrong
- Handling active hours incorrectly in real-time processing module
- Real-time processing module was not fit to handle streams at less than 30 FPS with falling behind
- Fall detection class was not set to handle input frames at less than 30 FPS
- Incorrect use of datetime and time libraries
- Inaccuracies in the fall detection algorithm were identified and improved.

As we polish the project, we will continue to perform system testing. This will include running everything on each of our laptops, with different hardware. We have built-in performance scaling, and expect the system to run smoothly between different devices.

User Acceptance Testing

Our final phase of testing was user acceptance testing. The goal was to evaluate the system's usability and ensure it met their expectations. We conducted this casually by putting two individuals, roughly aged 55, in an imaginary scenario where they purchased our system. We framed it as if someone had just installed the hardware, including cameras and the server, and gave them access to the mobile application. We told them to set up CovaCare as if they were configuring it for their mother.

They successfully added themselves as emergency contacts, and set up the camera using the username, password, and IP address they were given. They used the health check feature to see that the system was online and working, and even faked a fall which successfully sent an alert to their phone. Overall, it went smoothly and the users confirmed that the system met their needs. During this process, we received the following feedback:

- It was difficult to know when everything was set up and working, even with the health check. We came to the conclusion that a short user manual would help.
- The alert messaging could be refined slightly.
- Some of the tooltips could be more clear.
- When trying to click a tooltip with the keyboard open, it just closed the keyboard and did not click the button.