Software Testing Document
People Counting Infrastructure
Version 1.0

Date: March 26th, 2021

Team Name: PiWatcher

Project Sponsor: Duane Booher, NAU ITS

Team's Faculty Mentor: Volodymyr Saruta

Team Members: Seth Burchfield, Champ Foronda, Joshua Holguin, Brigham Ray, Brandon Thomas

Table of Contents

1.0 Introduction

People counting has become an important aspect of keeping people safe during the pandemic. It can also be used to help an organization utilize space more effectively once enough footfall data have been collected over time. This project focuses on providing a solution to the problem of allowing authorized personnel to monitor the number of people in rooms and buildings of an organization. Various technologies were utilized in building the people counting component and the accompanying web application. Each component within the platform performs different functions that collect and provide people counting data. Several of the components within this system operate independently of one another, but in a few cases, there is cross-communication. The process of developing such a system is difficult not only in terms of planning and development but also in terms of testing.

Due to the complex nature of this project, various testing methods must be performed on each component specifically as well as how communication between one another occurs to ensure that the system will operate effectively. Due to this, dedicated tests will be created that verify proper communication between components. Within each component of the system: the backend, frontend, and IoT device; multiple modules will need to be tested to ensure accurate performance in a real-world situation.

Several programming languages were utilized in this project, so different testing methods and libraries will be used throughout the testing process. The frontend is written in React and will utilize two recommended testing frameworks, Jest, React Testing Library, and Nock, to ensure components within the web application are performant and accurate in their rendering. The backend of the solution, as well as the IoT device, utilizes Python. The popular testing library Unittest provides the ability to unit test modules within these components. All three major components of the project communicate with at least one other module, so integration testing will need to be performed on each of them to simulate accurate communication. Due to user interaction being required within the IoT device and frontend components, they will both need to be tested for issues with basic user interaction and user experience whereas the backend will need to be tested for issues with developer interaction. Unit tests will be performed on each of the modules within the different components. The majority of the testing focuses on the components where user interaction, the complexity of the modules, and cross-communication are involved since each of these factors must seamlessly integrate. Each of the following sections will address in detail the different testing methods and libraries used by the different components of this project.

2.0 Unit Testing

Within the unit testing section, we are going to be discussing the plan of testing individual units within their respective component for expected output. To ensure our system is operating and functioning correctly, the unit testing will encompass various inputs.

The frontend of the project will be tested for accurate data processing and storage within the web application. Testing will include modules where data is received and displayed, such as live usage percentages and graphs, as well as where data is sent, such as the user login page and administrator settings. The backend of the project will be testing the resources that the REST API provides. The IoT device of the project will be testing the different services that it supplies such as people detection, scheduling, and the various bash scripts for setting up the application.

2.1 Unit Testing for Frontend

Nearly all of the components within the web application are functional by nature. Due to this, unit testing the frontend will consist of passing in different information throughout the application and verifying it all displays correctly.

Authentication

- User Login
    - The initial page that is seen by the user. Any text input is recorded within the component and sent for verification on submit. Testing will ensure the user data is passed around and sent to the backend accordingly.
- User Creation
    - A secondary page within the login component. Any text input is recorded within the component and sent on submit. Testing will ensure the user data is passed around and sent to the backend accordingly.

Dashboard

- Room Listing
    - This component is displayed with a list of the rooms within a selected building. Testing will ensure the correct pulling of rooms and their usages to be displayed on an interval.
- Room Selection
    - This component is individualized on a per-room basis. On click, the room will be selected within the state of the dashboard. Testing will ensure the correct room information is passed on selection.

- Chart Creation
    - The chart component is displayed after a room is selected, a new chart will be displayed with the selected building and room within its state. Testing will include this selection being set correctly as well as the chart information being pulled and updated on an interval.

Administrator

- User Listing
    - This component pulls and displays user's information from the database for admin access. Testing will ensure the information is passed correctly for display.
- User Updating
    - This component takes input from the admin and sends it on submit. Testing will ensure the input information is set correctly before sending.
- Role Creation
    - This component takes input from the admin and sends it on submit. Testing will ensure the input information is set correctly before sending.

2.2 Unit Testing for Backend

The backend of the system is a REST API built with Flask-Rest. Our login authentication service provides a way for providing authentication to the user. Our mongo manager service provides a way for either the client or the IoT device to make queries with the MongoDB database.

Unit testing for the Flask backend requires testing all the resources that are provided to ensure that the expected status code and data are received from the responses. The modules/resources that are going to be tested are:

- Authentication resources:
    - /api/auth/signup (POST):
        - The input boundary values that have been identified for this endpoint are in the form of JSON through the request body. The input values in the JSON consist of an email that is of type string, a password that is of type string, the full name that is of type string, and the role of the user account that is of type string. All other erroneous inputs are ignored through this resource. Any inputs that do specify the correct input keys but incorrect input type will automatically return an unsuccessful response.

        - The output boundary values that have been identified for this endpoint are in the form of JSON through the response body. The response body for this endpoint contains the status code of the response that is of type integer and a description detailing the success or failure that is of type string.

- /api/auth/signin (POST):
    - The input boundary values that have been identified for this endpoint are in the form of JSON through the request body. The input values in the JSON consist of an email that is of type string and a password that is of type string. All other erroneous inputs are ignored through this resource. Any inputs that do specify the correct input keys but incorrect input type will automatically return an unsuccessful response.

    - The output boundary values that have been identified for this endpoint are in the form of JSON through the response body. The response body for this endpoint contains a status code of type integer, a username of type string, a role of type string, and a JWT token of type string.

- /api/auth/roles (GET, POST):
    - The input boundary values for the GET request of this endpoint have been identified as query parameters. The only recognized input for this endpoint is a JWT token of type string. All other erroneous inputs are ignored through this endpoint. The input boundary values for the POST request of this endpoint have been identified as both query parameters and a JSON body. The conditions are the same for the query parameters in this POST request. For the JSON body, the input values in the JSON consist of a role name of type string, an admin boolean, and a raw data boolean. All other erroneous inputs are ignored through this resource. Any inputs that do specify the correct input keys but incorrect input type will automatically return an unsuccessful response.

    - The output boundary values for the GET request have been identified as a JSON object through the response body. The response body for this endpoint contains a status code of type int, a list of roles each containing JSON objects that hold a role name of type string, an admin variable of type boolean, and a can view raw data of type boolean.

- /api/auth/users (GET):
    - The input boundary values for this endpoint have been identified as query parameters. The only recognized input for the endpoint is a JWT token of type string. All other erroneous inputs are ignored through this endpoint.

    - The output boundary values for this endpoint have been identified as a JSON object through the response body. The response body for this endpoint contains a status code of type integer, a list of users each containing JSON objects that hold a username of type string, an email of type string, and a role of type string.

- /api/auth/users/update (POST):
    - The input boundary values for this endpoint have been identified as query parameters and a JSON object that is sent through the request body. The only recognized query parameter is a JWT token of type string. All other erroneous query parameters are ignored through this endpoint. The JSON object that is sent through the request contains a user email of type string and the new role of type string.

    - The output boundary values for this endpoint have been identified as a JSON object sent through the response body. The response body for this endpoint contains a status code of type integer and a user that contains a JSON object that holds the username of type string, an email of type string, a role of type string.

- Data Resources:
    - /api/data/building/room/hour (GET):
        - The input boundary values for this endpoint have been identified as query parameters. These parameters are a JWT token of type string, a building name of type string, and a room name of type string. All other erroneous query parameters are ignored through this endpoint.
        - The output boundary values for this endpoint have been identified as a JSON object sent through the response body. The response body for this endpoint contains a status code of type integer and a list of JSON objects that contain the timestamp as a DateTime

object, a building name as a string, building id as a string, room name as a string, room capacity as an integer, endpoint id as a string, and count as an integer.

- /api/data/building/room/day (GET):
    - The input boundary is the same as /api/data/building/room/hour.
    - The output boundary is the same as /api/data/building/room/hour.

- /api/data/building/room/week (GET):
    - The input boundary is the same as /api/data/building/room/hour.
    - The output boundary is the same as /api/data/building/room/hour.

- /api/data/building/room/month (GET):
    - The input boundary is the same as /api/data/building/room/hour.
    - The output boundary is the same as /api/data/building/room/hour.

- /api/data/building/room/quarter (GET):
    - The input boundary is the same as /api/data/building/room/hour.
    - The output boundary is the same as /api/data/building/room/hour.

- /api/data/building/room/year (GET):
    - The input boundary is the same as /api/data/building/room/hour.
    - The output boundary is the same as /api/data/building/room/hour.

- /api/data/buildings (GET):
    - The input boundary is identified as a query parameter. The only recognized query parameter through this endpoint is the JWT token defined as a string. All other erroneous inputs are ignored through this endpoint.
    - The output boundary is identified as a JSON object sent through the response body. The identified keys of the JSON object area status code of type integer and a list of JSON objects that each store the building name as a string and the building id as a string.

- /api/data/building/rooms (GET):
    - The input boundary is identified as a query parameter. The recognized query parameters through this endpoint are a JWT token defined as a string and the building name that is defined as a string. All other erroneous inputs are ignored through this endpoint.

        - The output boundary is identified as a JSON object that is sent through the response body. The identified keys of the JSON object is a status code of type integer and a list of JSON objects that contain a room as a string, current count as an integer, and room capacity as an integer

    - /data/iot/update (POST):
        - The input boundary is identified as a JSON object sent to the request body. The identified keys of the JSON object is a timestamp represented as a DateTime object, the building name presented as string, the building is represented as a string, the room name represented as a string, the room capacity represented as an integer, the endpoint id is represented as a string, and the count represented as an integer. All erroneous inputs are ignored through this endpoint
        - The output boundary is identified as a JSON object that is sent through the response body. The identified keys of the JSON object are a status code of type integer and a description of type string.

2.3 Unit Testing for IoT Device

The IoT Device of the system is set up on a Raspberry Pi that utilizes a camera that takes pictures of the classroom and analyzes it with multiple modules to identify a person in the picture then count the number of people before sending it off to the backend.

Unit testing for the Raspberry Pi will consist of ensuring each python file works as intended. This will be done by utilizing pytest, a python library designed for testing python code. The modules we will be testing are as follows: pi_connection.py, detect.py, loaded_model.py, scheduler.py, setup.sh, and set_env.sh.

- *Detect.py:*
    - This module will detect the number of people in the photo that is taken and call loaded_model and the pi_connection modules to load the yolo4 model we use for detection and send the count it outputs to the database. We will test this by ensuring the modules called here return correctly formatted data and ensuring all inputs are correct and no photo exists after complete execution.

- *Loaded_model.py:*

    - This module is very simple; it takes in a path to the model and loads it into memory. We will test this by ensuring the model is loaded using tensor flow commands and checking one of the returns we get from loading it that is a true-false based on if it was successful.

- *scheduler.py:*
    - This module kicks off the detect module and passes in a photo that is taken before the detect module. This will be tested by ensuring a photo is there to be passed in and the jobs for the scheduler are correct and waiting to be kicked off.

- *setup.sh:*
    - This script will set up python3 and call the set_env module. We will check to ensure the correct version of python is installed for this test.

- *set_env.sh:*
    - This script will set the environment variables needed to send data to our backend. This will be tested by using regexes to ensure the correct information is there for the variables as well as ensure they are set as intended.

3.0 Integration Testing

Within the integration testing section, we are addressing testing the communication between different components of the solution.

The frontend of the project will be tested in how connections with the backend component are handled. The testing is ensuring that the frontend can handle successes and failures correctly during connection. A deeper explanation is within section 3.1. The IoT testing is done similarly to the frontend but only needs to test the single update endpoint. More description of the IoT testing is within section 3.3

3.1 Integration Testing for Frontend

The frontend of this project is the web application that graphically displays the number of people in a room over time. It will be protected by a sign-in page that will be located on Northern Arizona University's network, where students and staff will need to be located physically or connected through the NAU VPN to access. User accounts and their assigned roles dictate the viewable information for that account.

After signing into the frontend sign-up page, the dashboard will fetch any buildings with people counting in action. Then, after the desired room is selected, the room's data will be fetched and displayed. An account with administrator privileges will have the option of viewing more specific data than non-admin users. Administrators also can manage other users and their roles.

Integration testing the frontend of our project will consist of utilizing the React Testing Library and Nock for mocking API calls. The frontend will need to handle successes and failures returned from the backend. If the connection is broken, the frontend must be able to gracefully handle these occurrences. Each endpoint will be tested separately from one another, with a total of two tests each. This will ensure that the connection between the frontend and backend is working as intended when both successful and unsuccessful.

3.2 Integration Testing for Backend

Since the backend does not communicate to external services, integration testing is not needed for the backend.

3.3 Integration Testing for IoT Device

The Raspberry Pi connects specifically to the backend of the solution so that the backend can send its counts to the database. This involves communication with the backend's IP. This needs to be correct, otherwise, no connection can be made. Other than the HTTP response string sent back from the database, it is a one-way communication from the raspberry pi to the backend.

A pi will have to make a post request to the backend each time it sends a count. This post request contains all the information regarding the endpoint and must be entered into the database in a specific format for the web application to display properly. A function can be created that asserts the data is formatted correctly before submission to the backend. This function can be run 1000 times or more with an error rate of less than .5% to verify the data flow is not corrupted throughout the application.

Once the string is verified, it can then be sent to the backend. The post request should return a status 200 verifying that the data was in the proper format and that the data was sent correctly. Whenever a status 400 is returned the string may be in an improper format or the connection to the backend is configured improperly. The message will return valuable information to help a developer determine if it is a connection issue or the JSON string was built in an improper format. A function harness can be created asserting the post request string contains a status 200 each time data is sent to the backend. This function can be run 1000 times to verify the connection to the backend is valid, and that the formatting of the JSON string is correct. Any error rate > .5% will need to be investigated.

4.0 Usability Testing

In this section of the usability testing, there are three parts to this section. The usability testing is the frontend, the backend, and the IoT part of the project. In section 4.1 section, there will be explaining the usability testing of the frontend and what it will be testing. In section 4.2, there will be explaining the usability testing for the backend of our application. In section 4.3, there will be explaining the usability testing of the IoT part of the project.

4.1 Usability Testing for Frontend

Usability testing will focus on confirming that user interaction and experience with the frontend are at an acceptable level. Testing will highlight if there is any trouble or misunderstandings about the layout of the web application as a whole, whether that is on desktop or mobile devices. Each page will be tested for visual and functional usability, on multiple different screen sizes.

4.2 Usability Testing for Backend

The backend of this solution is not built for basic user interaction. Instead, only a developer will ever be interacting with the core components of the backend. To test this portion of the backend, documentation will be provided that details the setup and all the resources that are provided through the REST API. Gathering results for this requires a developer to step through and follow the instructions on the documentation and ensure that the setup and the usage of the backend are correct and appropriate. Any issues or misguidance that is not provided or misinterpreted from the documentation will be recorded and adjusted accordingly.

4.3 Usability Testing for IoT

Due to the autonomous nature of the IoT device, Usability testing will not be available.

5.0 Conclusion

Due to the complex nature of this people-counting infrastructure, a multilayered testing approach is necessary for the majority of its components. These components are wired to provide connectivity between themselves and other components, along with error-free user interaction and modules. Each of the components' main functions will be unit tested to ensure reliable operation. Any communications between components will be subject to iterative testing that proves their reliability in a production environment. Lastly, modules within components that are subject to user interaction will be tested so that nobody interacting with the final product encounters any bugs. We believe that our testing suites will provide our client confidence in our product's success.