DETAILED TEST PLAN

1. **How test cases of different levels (frontend backend units, integration) are organized.**

We will simply keep our test cases of the different levels grouped feature-wise. This will allow us to keep the structure of the project easily traversable, and readable. The frontend and backend will be organized into its unit tests to address verification, and then integration testing, and finally the system tests and acceptance tests will be organized in separate folders. Each requirement will contain similar test cases that have minor changes based on the sub-requirement

Organizing test cases for R1:

GET commands:

In this section of R1 we will focus on how the action buttons responds to user input. We do not necessarily need a test_user instance yet and can just run a series of commands through and validate the correct. output is displayed

POST commands:

General test case:

```
test_user = User(
    email='test_frontend@test.com',
    name='test_frontend',
    password=generate_password_hash('test_frontend')
)
```

Test case modifications:

- Assign both email and password to empty strings/values (negative test case)
- Assign email to a non addr-spec defined in RFC 5322 (negative test case)
- Password does not reach required complexity (negative test case)
- Format errors in email or password (negative case)

Organizing test cases for R2:

GET commands:

Like R1, in this section of R2 we will focus on how the action buttons responds to user input. We do not necessarily need a test_user instance yet and can just run a series of commands through and validate the correct output is displayed.

POST commands:

General test case:

```
test_ticket = Ticket(
    owner='test_frontend@test.com',
    name='test_ticket_yo',
    password1 = generate_password_hash('test_frontend')
  password2 = generate_password_hash('test_frontend')
)
```

Test case modifications:

- Email and passwords do not satisfy general requirements (negative test)
- Passwords are not the same (negative test)
- Username is 1) empty, 2) contains special characters, 3) has space as first character, 4) has space as last character, 5) has space as both first and last character (negative test cases)
- Username is 1) shorter than 2 characters and 2) longer than 20 characters (negative test cases)
- Any of the attributes contain formatting errors (negative test)
- An email is used that is already registered in the system (negative test)

With successful test cases: ensure that user balance is set to 5000 and login page is shown.

Organizing test cases for R3:
GET commands:

This requirements section depends on what the page is displaying to the user. Here we will send in sequences of click (action) commands and then visibly validate the page is showing. The modifications for each test case in this requirement will be what actions are clicked to get to each different page.

Organizing test cases for R7:

This requirement section also depends on the page shown to the user, which we will have to validate that /logout is showing. Then, we will have to test that a user instance is not being held on to by the page after logging out. We can use a general login test case, log out, and then verify what needs to be verified (stated above).

Organizing test cases for R8:

General test case:
/*
We are just validating that there are no other pages tied to our website.
Modifications:

- Typing in paths that may seem like they are a “part of the website”, ex. /tickets
- Typing in paths that have small typos, ex. /regisrer, /lognn

We will need to validate that the 404 error is showing for this case

2. **The order of the test cases (which level first which level second).**

We will begin the order of test cases with testing the front end. When the font end testing is completed, we can move on to back end testing. Finally, after backend testing, we can do system testing and integration testing.

In regards to the order of frontend tests, we will begin by testing elements of the frontend that will be relied on for later testing. For example, we will begin by testing the /login page, and after that test the /register page since login tests will be relied on in the testing of registration. The /buy and /sell also rely on a successful login, so they will be done last in the frontend testing.

We will do the same in regards to the backend testing, by testing elements first that will be relied on for testing other elements.

We will begin by using unit tests to test these elements, and address the verification of each individual component. After the unit testing for front end and back end is complete, we will do integration testing, to verify that the groups of units can be integrated and work as a whole. After this, we will do system testing to verify that the product meets the functional specifications. Finally, we will do acceptance testing to validate that the software meets real intentions, and accept the result.

3. **Techniques and tools used for testing.**

   Techniques:

   Tools:

   - Selenium is a testing tool we can use for the front end and browser testing.
   - Pytest is a testing tool to be used on the python backend of the application
   - Actions is a testing tool to test input parameters within the application to ensure the app is performing the correct functions given meaningful input.

4. **Environments (all the local environment and the cloud environment) for the testing.**

The environment we will be testing this project on is the chrome browser, on a local home sever, since we are creating a web application. We will want to ideally test it on multiple devices and operating systems, to ensure that it works properly on all.

5. **Responsibility (who is responsible for which test case, and in case of failure, who should you contact)**

   R1: Michael

   R2: Anna

R3: Sarah

R7: Gabby

R8: Michael

Failure:

If failure is encountered, contact the members within the group. Each member of the group is responsible for X Y Z,

6. **Budget Management (you have limited CI action minutes, how to monitor, keep track and minimize unnecessary cost)**

With GitHub Actions, we have limited CI action minutes of 2000/month. Therefore, it is important to know how to effectively manage and minimize the minutes used. As for monitoring, in GitHub go to settings -> billing -> github actions, there you can see the details of the minutes used. Additionally, under “storage for actions and packages”, you can see the details of storage usage that syncs every hour. In terms of minimizing the action minutes, ensure that as few dependencies as possible are being downloaded and installed, since this reoccurs at every iteration of testing. Don’t install dependencies unnecessarily and limit environment variables to the narrowest possible scope.

How to minimize:

- Keep actions minimal:
    - Install as little as possible since all dependencies are downloaded and installed every time
- Don’t install dependencies unnecessarily
    - If publishing standalone Action, publish entire node_modules folder
    - Use caching mechanism
- Limit environment variables to the narrowest possible scope
    -
- Testing critical sections, multiple cases at once? Functions instead of lines

TA NOTES:

- 1 or a few sentences, check the time, not too many PRs, frequent checks
- No diagram?
- Doesn't expect details for each test case, how you organize the test cases, how you're gonna write unit test, increment unit test for each scenario. Q1
- Selenium, github actions ok. CI-python, prof put testing frames so we can mention that, black box testing techniques
- Environments OK.
- Failure and responsibility - contact group members OK, should only contact team members (no prof no TA). delegate roles to each team member, establish lead & scrum/exP positions
- Code frontend and back end, not unit test
- Front end not fully implemented, finish the first version of the server so we can do testing for next assignment. Don't write too much html, just focus on python code
- Backend.py - template for backend server flask what backend uses, flask SQL alchemy
- https://flask-sqlalchemy.palletsprojects.com/
- https://flask.palletsprojects.com/
- Might not need to write new models, just add verification for password and user check. Not a long solution
- Can use template, no renaming avoid hard coding, write to database itself.
- Flow diagram of architecture, diagram given is more like general structure, we need to give a structure for each class - how it is organized, what it is doing. We need to look at how classes are written and put them in diagram
-

Garbage but may be important later:

The levels of test cases are : unit, integration, system, and acceptance.

We start at unit testing because it consists of the most specific cases of testing. We can target singular functions and break down the program to find and isolate initial bugs easily. This will allow us to go through the program step by step and fix bugs incrementally. It is important that we test the front end first in unit testing over the backend. We will be creating separate test cases for front end functions and back end functions.

After unit testing where our singular functions have been accounted for, we move on to integration testing. Here, we make test cases that combine frontend and backend modules and then test them as a group. This will be important in seeing how the modules communicate with one another and if there are any bugs in passing information or calling on functions from other modules. This will be

important in testing cases that involve logging in and retrieving information from the back end, or registering a user and creating their account as an instance in the back end.

System testing will be our first instance of testing the program as a whole.Our goal here will be to ensure that the program has compiled correctly, meets all of our desired requirements, and passes quality standards. This will need more complex test cases that have multiple actions, such as a full simulation of someone registering, buying a ticket, selling another ticket, and then checking their account balance. At this point, other programmers outside of our team should be testing the program for functionality, usability, and other quality assurance indicators to make sure the program satisfies all requirements that were set by the customer.

Finally, we finish with acceptance testing. This will determine whether our program is ready for release/ regular use. We will use this final testing phase to make sure we have satisfied every possible known requirement. This will also involve making sure the final product meets ours/the customer’s needs. After the program passes this test, it will then be released for general use.