HSBC

Digital Walk-Around Project - Fat 6 Test Plan

Project Manager: Aidan Clarke Scott

Team Members: John Chapman, Willem van Doorn, Christopher Cliff, Matt Wilson, Sasha Maximovitch, Shane Labelle

March 30, 2022
Version 1.1

Sponsor Sign Off

Name: Branden K. Siegle	**Signature:**	**Date:**
Remarks:		
Name: Clifford Lee	**Signature:**	**Date:**
Remarks:		

Revision History

Date	Version	Description
03/06/2022	1.0	Creation of Document
03/30/2022	1.1	Added blurb about how many tests we implemented in summary

1 Summary

1.0 Testing Approach

Our testing approach will use a variety of manually performed and automated tested techniques including unit, SAT and UAT tests each of which employs white or black box testing where appropriate. We use a variety of tools to accomplish these different testing techniques, to perform the tests and to interface with (and isolate) certain components in our system as well as mock data appropriately. By identifying the risks associated with each testing technique, we were able to determine which other tests were necessary to adequately test a given component. This is incorporated into our test cycle plan and how we plan to test our components going forward. Starting off with testing individual components with unit tests then transitioning to SAT and UAT tests in future test cycles as the application development gets closer to completion. Throughout each test cycle period, we will report and fix bugs which appear due to showing up in testing. In total we expect to run four test cycles by the end of development.

We have not been able to implement all of the tests written in this document due to time constraints. However, we have completed the majority of the SAT, UAT, Manual and Performance tests that are outlined in this document.

1.1 Tools

Unit Testing: Jest (27.5.1), Mocha/Chai (9.2.1/4.3.6), lambda-tester (4.0.1), proxyquire (2.1.3)

Frontend Unit Testing: React Testing Library (12.1.2)

End to End Testing: Cypress (9.4.1)

API Testing: wscat (WebSocket cat - 5.1.0)

Static Code Analysis: Eslint (8.10.0)

Performance Testing: Apache JMeter (5.4.3)

2 Functional Test Plan

2.0 Intro

Below we describe all the tests for each of the core components involved in the MVP deliverable of the application. The tests are organised by testing technique, namely, Unit Tests, SAT, and UAT. Certain components do not use all forms of testing as they aren't necessary for fully testing all the functionality of that given component. For each testing technique, whitebox/blackbox testing, risks, and the number of test scripts are identified. For each technique, the happy-path and negative path scripts are enumerated and briefly described. For more detail about all the scripts, see the appendix (Section 7).

2.1 Login

Description:

Users will have to sign in to the application each time they wish to use it by filling out a form with their username and password and pressing a submit button.

Front End Unit Tests:

Technique: Automated black box testing with Jest and React Testing Library

Risks: There runs the risk that some of the libraries we use are not supported and can't be tested with Jest

Explanation: Jest is easy to use and well integrated with React making it a no brainer to use for Unit testing our components. It also supports asynchronous code which will be needed for testing authentication with Firebase. These tests will be necessary to ensure our login form is working and correctly authenticating with Firebase.

Number of scripts: 2 happy-path tests (UnitTestRenderForm, UnitTestValidInput) and 4 negative path tests (UnitTestIncorrectUsername, UnitTestIncorrectPassword, UnitTestInvalidInput, UnitTestMissingInput)

Back End Unit Tests:

Technique: Automated black box testing with Mocha/Chai

Risks: Testing is done in an offline serverless environment simulated outside of AWS, which may not ensure functions will work within AWS itself.

Explanation: Mocha provides an easy to use automated test framework for writing unit tests for our backend. It provides support for asynchronous code, and integrates well with Serverless which we'll be using to help run and test our Lambda functions outside the AWS dashboard. These tests will be necessary to ensure Firebase user tokens are properly authenticated in the backend.

Number of scripts: 1 happy-path script (UnitTestValidToken) and 2 negative path scripts (UnitTestInvalidToken, UnitTestMissingToken)

SAT Tests:

Technique: Manual blackbox testing of Lambda’s with wscat (WebSocket cat) in AWS

Risks: This does not exactly simulate a connection from the frontend of our application, there may still be unknown issues we could face when performing our UAT tests.

Explanation: Wscat provides an easy way to manually test our backend API and simulate a connection with our frontend. We need to test sending a user token to ensure authentication properly receives and handles valid and invalid tokens.

Number of scripts: 1 happy-path script (SATValidTokenTest) and 2 negative path script (SATInvalidTokenTest, SATInvalidFormatTest)

UAT Tests:

Technique: Automated blackbox testing with Jest and React Testing Library, backend deployed on AWS

Risks: Hard to detect and debug errors that occur on the backend

Explanation: Need to simulate user interaction through automated and manual form completion to test communication with the backend deployed on AWS. Jest and React Testing Library provide a means for performing frontend automation to login a user and request authentication with the backend.

Number of scripts: 1 happy-path script (UATValidLogin) and one negative path script (UATInvalidLogin)

2.2 Showing Current Active Meetings

Description:

When a user goes to the office page they will get a list of current active meetings for which they are allowed to see.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire. Automated whitebox frontend testing with React Testing Library.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. For getting active meetings we will test the lambda function to see if it returns the correct active meetings for that function. After this is done, the test can take the results given and then check a test database to see if the users in the given meetings are one’s that the test user should see.

For the frontend, we will create unit tests using the React Testing Library to mock a response from the backend, to ensure that the active meetings are displayed correctly.

Number of scripts: 7 happy-path scripts (GetNoActiveMeetingBackend, Get1ActiveMeetingBackend, Get3ActiveMeetingBackend, GetNoValidActiveMeetingBackend, GetNoActiveMeetingFrontend, Get1ActiveMeetingFrontend, Get3ActiveMeetingFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of the websockets is working as expected.

Number of scripts: 1 happy-path scripts to ensure the websocket returns a list of active meetings to be displayed to the user (SATGetMeetingTest)

UAT Tests:

Technique: Manual black box testing

Risks: Need more than one person to actively test this

Number of scripts: 1 happy-path scripts (UATGetMeetingTest)

Explanation: Manual black box testing would be the easiest way to test active meetings. You will need to have at least two users to test this. One or more will join a meeting, and the user that is testing will check if the meeting shows up in the office page.

2.3 Joining a Meeting

Description:

When a user goes to the office page with active meetings they can click on a button to join a current active meeting.

Unit Tests:

Technique: Automated whitebox frontend testing with React Testing Library.

Risks: There is no realistic way to test the backend for this, the unit tests for getting active meetings will also test return the link for the zoom meetings. But, there is not really a way to test if the zoom links are valid in backend unit tests.

Explanation: For the frontend, we will create unit tests using the React Testing Library to mock a response from the backend, it will have an active meeting that would be available to be joined.

Number of scripts: 1 happy-path scripts (JoinActiveMeetingFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of the websockets is working as expected.

Number of scripts: 1 happy-path scripts to ensure the websocket returns a list of active meetings to be displayed to the user (SATJoinMeetingTest)

UAT Tests:

Technique: Manual black box testing

Risks: Browser and version may have varying effects on the UI display

Explanation: Manual black box testing would be the easiest way to test joining active meetings. You will need to have at least two users to test this. One or more will join a meeting, and the user that is testing will try to join the meeting that shows up in the office page.

Number of scripts: 1 happy-path scripts (UATJoinMeetingTest)

2.4 Creating Instant Meeting

Description:

In the office page the user clicks on a button to make a new office room which prompts for a selection of users which will then be called into a new meeting scheduled immediately.

Front End Unit Tests:

Technique: Automated black box testing with Jest and React Testing Library

Risks: There runs the risk that some of the libraries we use are not supported and can't be tested with Jest

Explanation: Jest makes automating unit tests within React an easy task, combined with the React Testing Library it is a good framework for us to use to test our frontend UI. On the front end we need to test to ensure a user selection form appears and is properly rendered when the New Office Room button is pressed. That users within the user list on the form that are currently in meetings are not able to be selected for an instant meeting. And that a proper backend request is formed when the form is submitted, which contains all the selected users.

Number of scripts: 4 happy-path tests (UnitTestRenderUserSelectionForm, UnitTestNoBusyUsers, UnitTestMultipleBusyUsers, UnitTestNoFreeUsers) and 1 negative path test (UnitTestNoUsersSelected)

Back End Unit Tests:

Technique: Automated black box testing with Mocha/Chai

Risks: Testing is done in an offline serverless environment simulated outside of AWS, which may not ensure functions will work within AWS itself.

Explanation: Mocha provides an easy to use automated test framework for writing unit tests for our backend. It provides support for asynchronous code, and integrates well with Serverless which we'll be using to help run and test our Lambda functions outside the AWS dashboard. These tests will be necessary to ensure that an immediate zoom meeting is properly created (through Zoom API) and all valid input users are invited.

Number of scripts: 1 happy-path scripts (UnitTestManyUsers) and 3 negative path script (UnitTestInvalidFormat, UnitTestInvalidUser, UnitTestNoUsers)

SAT Tests:

Technique: Manual blackbox testing of Lambda's with wscat (WebSocket cat) in AWS

Risks: This does not exactly simulate a connection from the frontend of our application, there may still be unknown issues we could face when performing our UAT tests.

Explanation: Wscat provides an easy way to manually test our backend API and simulate a connection with our frontend. Using wscat we can send websocket events to our api to trigger the creation of an instant meeting, and then we can manually ensure users immediately receive an invite to the zoom meeting. WIth multiple connected and authenticated wscat user instances we can test to ensure that invited users receive an invite to the meeting and non-invited users do not.

Number of scripts: 1 happy-path script (SATInstantMeetingCreated) and 1 negative path script (SATInstantMeetingNotCreated)

UAT Tests:

Technique: Manual blackbox testing through frontend client hosted on Heroku, with backend deployed on AWS

Risks: Hard to detect and debug errors that occur on the backend

Explanation: Need to simulate end-to-end user interaction for the creation of an instant meeting through testing communication with the backend deployed on AWS. Mock users will be added to the database, and tests will be done to create instant meetings with the mock users. We can have multiple clients open to test to see that users are notified when they are included in an instant meeting. We will also test to ensure a new Office Room is created for the instant meeting.

Number of scripts: 1 happy-path script (UATInstantMeeting)

2.5 View Your Groups & Members Locations

Description:

When a user is on the groups page, they should be able to see a list of all of their previously saved groups, and the current locations of each user in those groups.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire. Automated whitebox frontend testing with React Testing Library.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai to create unit tests directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. Specifically, for viewing groups and user locations in the backend, we will test that the lambda function correctly obtains each group user's current location from the zoom API, and the lambda successfully gets group information from the DynamoDB for each group.

In the frontend, the unit tests created with the React Testing Library will use mocked backend data, to ensure that each group is displayed with correct information and user locations.

Number of scripts:

- 4 happy-path scripts (ShowNoGroupsBackend, ShowMultipleGroupsBackend, ShowNoGroupsFrontend, ShowMultipleGroupsFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned by the websocket is valid and accurate.

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

Number of scripts:

- 2 happy-path scripts (ShowNoGroupsSAT, ShowMultipleGroupsSAT)

UAT Tests:

Technique: Manual black box testing through the user interface

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest to test viewing a user's groups and their member's locations from end to end. UAT testing with this would require at least 2 users. The first, would join an active meeting, and the second would then navigate to the groups page, create a group with that user, and ensure that the correct group information is displayed, and that the location of the first user is correct.

Number of scripts:

- 2 happy-path scripts (ShowNoGroupsUAT, ShowMultipleGroupsUAT)

2.6 Create New Group

Description:

When a user is on the groups page, there is an option to create a new group. Upon clicking this button, a dialogue will be presented to them, where they can set group information and select group members.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire. Automated whitebox frontend testing with React Testing Library.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai to create unit tests directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. Specifically, for creating new groups we will test that the lambda function correctly uses the data sent to it from the frontend to write the new group to the mocked database, and then ensure that the new group entry is present in the mocked database.

For the frontend, we will create unit tests using the React Testing Library to mock a response from the backend, to ensure that the new group is displayed with the correct info.

Number of scripts:

- 4 happy-path scripts (GroupIsCreated2MembersBackend, GroupIsCreated5MembersBackend, GroupIsCreated2MembersFrontend, GroupIsCreated5MembersFrontend)
- 2 negative-path scripts (GroupNameAlreadyExistsBackend, GroupNameAlreadyExistsFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned by the websocket is valid and accurate.

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

Number of scripts:

- 1 happy-path scripts (GroupIsCreatedSAT)
- 1 negative-path scripts (GroupNameAlreadyExistsSAT)

UAT Tests:

Technique: Manual black box testing through the user interface

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest to test the creation of a group from end to end. Simply navigating to the UI groups page and creating groups is the best way to test this feature through UAT.

Number of scripts:

- 1 happy-path scripts (GroupIsCreatedUAT)
- 1 negative-path scripts (GroupNameAlreadyExistsUAT)

2.7 Calling Your Group

Description:

User creates an immediate call with all the members of one of the groups they created, the members immediately receive a join call popup through Zoom.

Unit Tests:

Technique: Automated white box testing with Mocha/Chai, lambda-test and proxyquire.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: We will use mocha and chai to create the unit test framework and lambda-test to mock events for the lambda functions. Unit tests are appropriate for testing the backend responsibilities for this feature as we can use them to isolate the specific functionality and test its return value without interference from any other part of the system. We will isolate individual components and use proxyquire to mock responses/data for the external Zoom API and DynamoDB and compare the results to expected data. The unit tests will call with a mock group and ensure that the backend returns the correct Zoom IDs and the invite link.

Number of scripts:

- 3 happy-path scripts (UnitGroupsCall2Members, UnitGroupsCall5Members, UnitGroupsCall2MembersUserBusy)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned is valid and accurate.

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

Number of scripts: 2 happy-path script (SATGroupsCallTest, SATGroupsCall2MembersUserBusy)

UAT Tests:

Technique: Manual black box testing through UI

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest approach for testing the calling group functionality with the entire system. You will need to have at least two users to test this. This will test that the users all receive notifications that they are being called and that any error messages are shown to the caller appropriately.

Number of scripts: 3 happy-path scripts (UATGroupsCallTest2Members, UATGroupsCallTest5Members, UATGroupsCallUserBusy)

2.8 Add User to Favourites

Description:

User searches through all employees in the organisation and chooses one to add to their favourites list for easier calling and viewing of their location.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai to create unit tests directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. For adding users to your favourites list, we will test that the lambda function correctly uses the user ID sent to it from the frontend to write the new favourite user to the mocked database, and then ensure that the new favourite user is present in the mocked database.

Number of scripts:

- 1 happy-path scripts (UnitAddUserFavourite)
- 1 negative-path scripts (UnitUserAlreadyFavourite)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned by the websocket is valid and accurate.

Number of scripts:

- 1 happy-path scripts (SATAddUserFavourite)
- 1 negative-path scripts (SATUserAlreadyFavourite)

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

UAT Tests:

Technique: Manual black box testing through UI

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest approach for testing the adding favourites functionality with the entire system. This will test that a user can be added as a favourite and that this persists through sessions in the database. Based on frontend implementation it should not be possible to add a user who is already a favourite.

Number of scripts:

- 1 happy-path scripts (UATAddUserFavourite)

2.9 Viewing Your Favourite Members Locations

Description:

A modal containing the list of all of the user's favourite users also contains information as to the current zoom whereabouts of those users.

Unit Tests:

Technique: Automated white box testing with Mocha/Chai

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: We will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. We will mock a caller user for each test

Number of scripts:

- 3 happy-path scripts (UnitNoFavouritesToView, UnitAllFavouritesAvailable, UnitSomeFavouritesInCall)
- 1 sad-path script (UnitInvalidUserID)

SAT Tests:

Technique: Manual black-box testing with wscat (WebSocket cat) and test-endpoints that create fake data

Risks: Sending requests using wscat does not directly mirror the application making requests to our websocket.

Explanation: First we create a test endpoint in our backend that creates a zoom meeting with a few participants. Use wscat to send our request to our get favourites locations endpoint with a test userID. Manually compare wscat response to check that data is expected.

Number of scripts:

- 3 happy-path scripts (SATNoFavouritesToView, SATAllFavouritesAvailable, SATSomeFavouritesInCall)

- 1 negative-path script(SATInvalidUserID)

UAT Tests:

Technique: Manual black box testing using the UI and calling test endpoints with wscat.

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing will allow us to replicate common user stories to ensure the frontend is working correctly. Automation would be difficult and not as effective for gauging user experience. Begin by logging in as a test user with no favourites. Create a test endpoint that adds several users to a short zoom meeting. Perform the following tests in order.

Number of scripts:

- 3 happy-path scripts (UATNoFavouritesToView, UATAllFavouritesAvailable, UATSomeFavouritesInCall)

2.10 Viewing Your Past/Present/Future Meetings

Description:

A user views their own meetings in their calendar view. They click on the button to open their calendar view, and on page load are met with their calendar, populated with their past, ongoing and future meetings. They can view meetings with topics, names of participants, time and length of the meetings, and for past meetings, they'll also be provided with a link to a recording if it exists.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire. Automated whitebox frontend testing with React Testing Library.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai to create unit tests directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. Specifically, for viewing groups and user locations in the backend, we will test that the lambda function correctly obtains each group user's current location from the zoom API, and the lambda successfully gets group information from the DynamoDB for each group.

In the frontend, the unit tests created with the React Testing Library will use mocked backend data, to ensure that each group is displayed with correct information and user locations.

Unit tests are useful for viewing meetings as there are a series of individual components which can all be tested in an automated manner,

Number of scripts:

- 3 happy-path scripts (MeetingsBackend, RecordingsBackend, DisplayMeetingsFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned by the websocket is valid and accurate.

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

Number of scripts:
2 happy-path scripts (FetchMeetingsSAT, FetchRecordingsSAT)

UAT Tests:

Technique: Manual black box testing through the user interface

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest to test the creation of a group from end to end. Simply navigating to the UI groups page and creating groups is the best way to test this feature through UAT.

Number of scripts:
1 happy-path script (FetchMeetingsUAT)

2.11 Viewing Other Users Availability

Description:

A user is able to view the availability of another user in the calendar. They can see what times are blocked out, and what times are available, but cannot see anything about blocked out times.

Unit Tests:

Technique: Automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire. Automated whitebox frontend testing with React Testing Library.

Risks: Since we will have to run the lambda functions locally, as we can not use mocha/chai to create unit tests directly with AWS, we can not test situations specific to running the lambda functions through AWS itself, or completely ensure that the functions will work within the AWS ecosystem.

Explanation: For the backend, we will create a set of unit tests locally using mocha and chai to create the unit test framework, lambda-test to mock events for the lambda function, and proxyquire to mock responses/ data for the external Zoom API and DynamoDB. Specifically, for viewing groups and user locations in the backend, we will test that the lambda function correctly obtains each group user's current location from the zoom API, and the lambda successfully gets group information from the DynamoDB for each group.

In the frontend, the unit tests created with the React Testing Library will use mocked backend data, to ensure that each group is displayed with correct information and user locations.

Number of scripts:

- 2 happy-path scripts
  (AvailabilitiesBackend, DisplayAvailabilitiesFrontend)

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React, only checks that the data returned by the websocket is valid and accurate.

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of this websocket is working as expected.

Number of scripts:
1 happy-path script (FetchAvailabilitySAT)

UAT Tests:

Technique: Manual black box testing through the user interface

Risks: Browser and version may have varying effects on the UI display.

Explanation: Manual black box testing would be the easiest to test the creation of a group from end to end. Simply navigating to the UI groups page and creating groups is the best way to test this feature through UAT.

Number of scripts:
1 happy-path script (FetchAvailabilityUAT)

2.12 Browser Notifications Upcoming Meeting Reminders

Description:

When a meeting a user is a participant of is 30 and 15 minutes away from starting a browser notification pops up to remind them of the upcoming meeting.

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: Does not ensure data is correctly formatted and displayed by React

Number of scripts: 2 happy-path scripts to ensure the websocket notifies connected client that the meeting is 30 or 15 minutes away (SAT30MinWebsocketNotificationTest, SAT15MinWebsocketNotificationTest)

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of the websockets is working as expected.

UAT Tests:

Technique: Manual black box testing

Risks: Manual testing makes accurate timing difficult

Number of scripts: 2 happy-path scripts (UAT30MinuteNotificationTest, UAT15MinuteNotificationTest)

Explanation: Manual black box testing makes the most sense for testing browser notifications, as automating this would be difficult and manual testing is just as effective to check if they are appearing at the right times.

Number of scripts: 2 happy-path scripts (UAT15MinuteNotificationTest, UAT30MinuteNotificationTest)

2.13 Popup Modal to Join Scheduled Call

Description:

When a user's scheduled meeting begins, a modal appears on the application with a button to join the meeting.

SAT Tests:

Technique: Manual black box testing with wscat (WebSocket cat)

Risks: This test does not ensure that the websocket properly sends the message to the front-end at the right time (namely when a user meeting starts).

Number of scripts: 1 happy-path script (SATJoinMeetingPopup)

Explanation: Manual black box testing with wscat is good in this case because we do not want to have to worry about any UI (user interface) interactions for SAT tests or backend trigger timings. Using wscat allows websocket events to be triggered and received directly in individual terminals without the need for any user interaction, so it is the perfect tool to ensure the base functionality of the websockets is working as expected.

UAT Tests:

Technique: Manual black box testing.

Risks: Different browsers and browser versions may create different results.

Number of scripts: 2 happy-path scripts (UATJoinMeetingPopupJoin, UATJoinMeetingPopupIgnore)

Explanation: Manual black box testing makes the most sense for testing this timed notification, as the popup would only appear if the user is logged in and viewing the page right as a meeting begins. This is best simulated manually, by scheduling a meeting in the near future and checking the application for the popup modal close to the scheduled time.

3 Non-Functional Testing

3.0 Intro

This section describes our testing strategy for our application's non-functional requirements. Because of the nature of non-functional requirements, these will be harder to test than our functional components. The non-functional tests will most likely be performed when most of the development for our application has finished, as that is when the required features to actually test the non-functional requirements will all be put together and working.

3.1 Support for at least 20 concurrent users

Description:

The web application should support full operation for at least 20 concurrent users who are logged in and executing tasks within the application.

Unit Tests:

N/A

SAT Tests:

Technique: Automated performance testing using JMeter

Risks: JMeter will simulate all virtual users using the same cookie

Explanation: JMeter is a 3rd party testing software that lets you run performance tests on a web application by simulating the in-order calls to various back-end API's. We will be able to simulate any number of users and load using this system but since there are many actions users can take on the site, we will limit the usage paths to a simple linear use case. This use case will involve a user logging in and creating a meeting.

Number of scripts: 1 happy-path(ManyConcurentUsersSAT)

UAT Tests:

Technique: Manual usage testing

Risks: Difficult to get enough users to manually test

Number of scripts: 1 happy-path(ManyConcurentUsersSAT)

Explanation: This will involve the team members signing into the application with multiple test accounts each and performing functions to ensure that speed and accuracy of the system is adequate.

3.2 Response times for an individual search be < 3 seconds

Description:

The web application should support individual search times that are less than 3 seconds total. This includes API calls and any front-end sorting or filtering.

Front End Unit Tests:

Technique: Automated black box testing with Jest and React Testing Library

Risks: There runs the risk that some of the libraries we use are not supported and can't be tested with Jest

Explanation: Jest is easy to use and well integrated with React making it a no brainer to use for Unit testing our components. It also supports asynchronous code which will be needed for testing authentication with Firebase. These tests will ensure that any front-end sorting/filtering of searches will be less than 1 seconds(assuming a 1 second max RTT for the back-end API call)

Number of scripts: 1 happy-path test (FilterAndDisplaySearchResults)

SAT Tests:

Technique: Automated performance testing using JMeter

Risks: JMeter will only give an approximate response time

Explanation: JMeter is a 3rd party testing software that lets you run performance tests on a web application by simulating the in-order calls to various back-end API's. We will be able to simulate any number of users and load using this system but since there are many actions users can take on the site, we will limit the usage paths to a simple linear use case. This use case will involve performing a search with an already logged in user X numbers of times.

Number of scripts: 1 happy-path(FilterAndDisplaySearchResultsSAT)

UAT Tests:

Technique: Manual usage testing

Risks: Difficult to get enough users to manually test

Number of scripts: 1 happy-path(FilterAndDisplaySearchResultsUAT)

Explanation: Using the chrome developer tools(network tab) we will perform searches and ensure that the total response time is less than 3 seconds.

3.3 Scale design capacity to accommodate up to zoom maximum

Description:

On a free zoom account you are able to have 100 participants in one meeting at a time. Our web application should be able to scale in the future to accommodate that number of people or more.

SAT Tests:

Technique: Automated performance testing using JMeter

Risks: JMeter will simulate all virtual users using the same cookie

Explanation: JMeter is a 3rd party testing software that lets you run performance tests on a web application by simulating the in-order calls to various back-end API's. We will be able to simulate a high number of users and load using this system but since there are many actions users can take on the site, we will limit the usage paths to a simple linear use case. This use case will involve a user logging in and trying to join a meeting with around 100 participants in it.

Number of scripts: 1 happy-path(JoinLargeParticipantMeetingTestSAT)

UAT Tests:

Technique: Manual Usage Testing

Risks: It will be difficult for us to get enough people to test this

Explanation: Manual testing makes the most sense for this so that we can actually test out having a large amount of users on our web application.

Number of scripts: 1 happy-path(JoinLargeParticipantMeetingTestUAT)

4 Security Testing & Test Data Approach

4.0 Intro

Our application will be dealing with sensitive information which we will need to ensure is protected from unauthorised viewers. Unauthorised viewers include non-registered users who have not been authorised through firebase, as well as registered employees who should not be able to see sensitive data, such as zoom meetings, of users higher than themselves in the employee hierarchy. We will be testing the security of our application to ensure that data is only sent to the intended, firebase authorised recipient.

4.1 Security Testing

We'll be performing security testing on a variety of different components of our system.

Within our main testing framework we'll be performing Unit, SAT and UAT testing on our frontend and backend authentication systems. This includes our frontend login form which will be tested to ensure users who try to connect with incorrect information are denied access to our application. Our backend will be tested to ensure that only users who connect with a valid authentication token from Firebase will be able to have access to our backend API.

Additionally we'll be making use of a number of open source tools to check our software and dependencies for known bugs and vulnerabilities. We'll be using OWASP Dependency-Check to scan the Node JS dependencies we use in both our front and back end systems to ensure we are not using any outdated dependencies, or ones with known security vulnerabilities. We'll be performing static application security testing (SAST) using SonarQube to check for bugs and vulnerabilities within our app source code. Lastly we'll be doing dynamic application security testing (DAST) using OWASP Zap to perform penetration testing to ensure our app will be secure from attackers.

4.2 Test Data Approach

We'll be manually creating a list of 10 users with fake usernames and randomly generated passwords. Each user will be registered with Firebase, and have it's own row in our DynamoDB table for storing group and favourites data. These users will be given mock groups containing various combinations/subsets of the other 10 users which will assist in testing multiple group related activities. The users will also be organised in an arbitrary hierarchy which will be used to simulate the visible active meetings in the Office page. Certain users will also arbitrarily be given subsets of the other users as favourites which will help simulate favourite related tests.

4.3 Data Masking Needs

The only sensitive data we handle are the users emails and passwords for authentication which is securely handled and stored with Firebase.

5 Regression Testing

5.0 Intro

Throughout the testing and development process we will use a subset of the tests listed above to act as our regression test suite to ensure that all prior functionality is preserved as we iterate the code base. Many of our unit tests will be used for regression testing to ensure that the core functionality of our components are continuing to work correctly.

5.1 Front End Regression Tests

Using the unit tests that we defined above which make use of the React Testing Library, we will run frontend unit tests every test cycle for our components. This will ensure that any updated code will still pass any previously written unit tests and preserve the state of functionality of our application. Each core frontend component will have its unit tests run on each major frontend build. We will primarily rely on unit tests made for the core components listed above (for example those outlined in 2.1, 2.5, and 2.6). Another primary method of regression testing for frontend will be done through ensuring that for each major frontend build, the UI components of the UAT tests outlined in section 2.2 - 2.9 (the core MVP components) are passing, and the components are working as expected. Through the thorough testing of the components outlined in 2.2-2.9 through UAT and unit testing on each major frontend build, we will ensure that any regressions on our core MVP components are caught quickly, and thus the impact of any regressions will be minimised.

5.2 Back End Regression Tests

While back end regression testing involves a more diverse set of tools, the fundamental concept is still the same. We will use a batch of automated whitebox backend testing with mocha/chai, lambda-tester and proxyquire to cover the regression testing needs for our back end code. Unit testing from components 2.2 - 2.9 will be continually tested, as the functionality tested is core to our product and any bugs will need to be addressed immediately. The automated nature of our unit tests will make it easy to quickly run to check that our core MVP (Minimum Viable Product) functionality continues to work even as we add more updates and layers on top of it. We will also be using the SAT tests from components 2.2 - 2.9 as they test both our backend lambda functions and the websocket connection for the core features of the application.

6 Test Cycles

6.0 Intro

This section outlines our test cycle plan for the remaining duration of the project. Each cycle (roughly aligning one-to-one with the remaining weeks we have remaining for the project) outlines the tasks that should be completed, the critical requirements that must be finished by the end of each cycle (otherwise the plan will be delayed), and the acceptable issues that could be relatively easily dealt with in future cycles (though they by no means should be delayed if otherwise avoidable). The high-level summary of this test plan cycle, is that we will start by writing and testing with unit tests, then move on to SAT testing once a majority of the MVP system is functional and unit tests are passing, and finally completing UAT testing, which will be completed when our product is almost entirely complete, with all bugs at each stage being reported, and fixed promptly and in a well-thought-out manner.

6.1 Cycle One

Tasks:

- Backend unit tests
  - The local lambda testing environment must be set-up, including the installation of mocha, chai, lambda-tester and proxyquire to create the initial test suite.
  - Test classes will be created for each primary lambda function.
  - Test plans will be outlined for each function, and unit tests will be written to cover each case mentioned in our functional test plan above (section 2).
- Frontend unit tests
  - The frontend testing environment must be set-up with the React Testing Library.
  - Test files will be created for each core React component.
- Develop a well-thought out system for reporting bugs, and notifying those responsible for fixing them. In our case, this will be done with a separate channel in our group chat, where all bugs are tagged and reported, and the user is directly notified, where they can read what bug was found, and the steps required to reproduce it.

Critical Requirements:

- Backend unit tests must be functionally complete for the primary lambda functions involved in the core MVP product, including:
  - getActiveZoomMeetings
  - getUserGroups
  - joinActiveZoomMeeting
  - createInstantZoomMeeting
  - scheduleZoomMeeting
  - getFavouriteLocations
  - loginHandler
- Frontend unit tests must be complete for the React components used to create the core MVP pages, including:
  - The Office page
  - The Groups page
  - Active meeting cards

    - Group cards
    - Group creation
    - Instant meeting creation
    - Scheduled meeting creation
- The bug reporting system should be working, and any bugs found must be reported in the correct format, with all parties involved in fixing them being promptly notified.

Acceptable Issues:

- Some backend unit tests are not essential for this cycle, these mainly include unit tests for lambda functions that will be more thoroughly developed at later points in the development phase, and are not core to the MVP, such as:
    - getFutureCalanderMeetings
    - getPastCalanderMeetings
    - createMeetingInOutlook
    - getMeetingRecording
    - notifyUpcomingMeeting
    - getAccountProfile
    - setAccountProfilePicture

- Similarly to the backend unit tests, for the frontend components not directly involved in the core MVP product, unit tests can be more thoroughly developed at a later date, these include React components involved with:
    - The Calendar page
    - The Profile page
    - Browser notifications
    - Displaying recorded meetings in browser

6.2 Cycle Two

Tasks:

- SAT tests:
    - Ensure the SAT test cases for core systems are outlined and clearly explained, so each of our team members can easily perform any of the tests, and we can smoothly collaborate to share test results or perform the manual tests in conjunction.
    - At the start of the cycle, every test case for the core systems will be run, and any bugs will be reported to, and fixed by the people working on the failing systems.
    - At the end of the cycle, all core SAT test cases will be run again, and any bugs that still remain should be prioritised, as all core MVP systems should be functional at this point in time.
- Unit tests:
    - Finish writing any unit tests for non-core systems that were missed in the previous cycle.
    - Continue to run unit tests for each system as their development is being finalised, make certain that any bugs found due to these tests are fixed quickly and reliably.

Critical Requirements:

- All unit tests classes are finished, and contain well-thought-out tests for the required systems, so we can ensure that there is a clear plan to follow for the remainder of the development period, and that each of our systems has reliable, functional, and thoroughly-tested features.
- All core MVP SAT tests should have been run multiple times, and they should be passing at this point, with all bugs discovered having been fixed.

Acceptable Issues:

- Though the majority of development should be finished by this point, there may be some stretch-goal systems that are not entirely finished, so it is acceptable for these systems to have some failing unit tests, and if any of the SAT tests were able to be run (if any of the WebSockets actions for these stretch-goal items are reachable), it is acceptable to have some issues, though all bugs should be recorded, and dealt with accordingly.

6.3 Cycle Three

Tasks:

- UAT tests:
    - Outline a variety of unit tests for each core component, and assign each test to a member of the team. All tests that require multiple people will be assigned as such, and all members will decide upon a plan for testing the system collaboratively.
    - Begin by performing UATs on the core MVP system, reporting any bugs found in the appropriate channels.
    - Once the core MVP systems are well-tested, and the tests are passing, move onto the stretch-goal system, and report/ fix any bugs that are found for these systems.
    - After both the stretch goal and MVP components have been thoroughly tested, begin the UAT edge-case testing on different environments, and at a variety of activity loads.
- Finish SAT testing, and ensure that any non-core features missed in the previous cycle have tests created for them, and are thoroughly tested.
- Ensure all core unit tests are passing. Having written the tests in previous cycles, and structured development to ensure these tests pass, at this point, with development nearly complete, all of the core unit tests written earlier should be passing, and the overall product should be close to working as intended.

Critical Requirements:

- All unit tests are passing for both the backend and frontend components.
- All SAT tests are passing for both the core and stretch-goal components.
- All of the core, and stretch-goal UAT tests must have been run on the system, at least with the basic configuration (testing on a single browser with a consistent activity level).
- The core MVP UAT tests must all be passing, as full end-to-end coverage is essential at this point in the testing cycle, and all MVP systems must be complete with no issues.

Acceptable Issues:

- Some UAT edge cases may have been missed, at this point in time, acceptable missed UAT cases involve those that require a large amount of initial setup, such as testing the product on a variety of different browsers, from different computers, and especially those that require a multitude of users to test. As most of these tests are ‘repeats’ of what was previously tested, just under different circumstances, it is acceptable to not have these entirely finished by the end of this cycle, though they must be planned to be run in the future.

6.4 Cycle Four

Tasks:

- Finish any UAT testing that was missed in the previous cycle. Ensure a variety of different cases are covered for both the core MVP and the stretch goal systems, such as running the product on different machines, different browsers, or at different activity loads if possible.
- Re-run all tests multiple times, and ensure that the product is working as intended, and that no test edge cases were missed at any point in the testing cycle. If any issues are found, address them immediately, and make certain that any issues with testing, or the development of the product are resolved. Re-test the product with all of the outlined tests after each issue is fixed.

Critical Requirements:

- The product is working as intended, and all tests have been run multiple times, with all issues fixed, and every test passing each time they are run.

Acceptable Issues:

- None, at this point, all of our features must have been thoroughly tested, and no tests of any kind should be failing.

7 Appendix

7.0 Intro

Here is a list of all of the test scripts for functional and non-functional components in detail and any other information that we deemed to be useful but too specific to be included above. Every component is listed in the same order as the components section (Section 2), with non-functional components after (Section 3).

7.1 Login Test Scripts

Unit Tests - Frontend

Login: UnitTestRenderForm	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	On rendering of the App, all elements of the login form including the email-input, password-input, and submit-button are checked to make sure they exist.
Input	React App Login Component
Output	Tests pass if all elements exist

Login: UnitTestValidInput	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	When the form is filled out properly with a valid email and password, and the submit button is pressed, Firebase returns the UserCredential matching the email
Input	Valid email and password - press the submit button
Output	Returns the UserCredential

Login: UnitTestIncorrectUsername	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	When an email is submitted in a properly filled out form that is not of a registered user Firebase throws an error
Input	Invalid email and any password - press the submit button
Output	Invalid email error thrown

Login: UnitTestIncorrectPassword	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	When the form is filled out completely with a valid email but incorrect password Firebase throws an error
Input	Valid email, invalid password - press the submit button
Output	Invalid password error thrown

Login: UnitTestInvalidInput	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	When the form is not filled out correctly (invalid email format) you are not able to submit the form
Input	Incorrectly formatted email and any password - press submit
Output	Invalid email error thrown

Login: UnitTestMissingInput	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	When the form is missing the email or password field you are not able to submit the form
Input	Correctly input email, no password
Output	Submit button is greyed out and inaccessible

Unit Tests - Backend

Login: UnitTestValidToken	
Type	Automated
Automation Method	Automated black box testing with Mocha/Chai
Details	When a request is received with a valid token, the token is parsed and validated by Firebase and then a IAM policy allowing access to the API is returned
Input	Valid token
Output	IAM policy - API access allowed

Login: UnitTestInvalidToken	
Type	Automated
Automation Method	Automated black box testing with Mocha/Chai
Details	When a request is received with an invalid token, the token is parsed and invalidated by Firebased, and an IAM policy denying access to the API is returned
Input	Invalid token
Output	IAM policy - API access denied

Login: UnitTestMissingToken	
Type	Automated
Automation Method	Automated black box testing with Mocha/Chai
Details	When a request is received with no token an IAM policy denying access to the API is returned
Input	Empty
Output	IAM policy - API access denied

SAT

Login: SATValidTokenTest	
Type	Manual
Details	Wscat sends an authentication request with a valid token, and is returned confirmation that the user was successfully authenticated
Test Conditions	Websocket Auth Request Message - with valid token
Output	Confirmation Message

Login: SATInvalidTokenTest	
Type	Manual
Details	Wscat sends an authentication request with an invalid token, and is returned an error message stating that the user was not successfully authenticated
Test Conditions	Websocket Auth Request Message - with invalid token
Output	Error Message

Login: SATInvalidFormatTest	
Type	Manual
Details	Wscat sends an authentication request with an invalid format, and is returned an error message stating that the user was not successfully authenticated
Test Conditions	Websocket Auth Request Message - with invalid format
Output	Error Message

UAT

Login: UATValidLogin	
Type	Automated
Automation Method	Black box with Jest and React Testing Library, backend deployed on AWS
Details	The login form is filled out with valid email and password, the submit button is pressed and the user is authenticated with Firebase which allows the user access to the rest of the application
Input	Valid email and password - submit button is pressed
Output	Authentication Success Message - Main App page opens

Login: UATTestRequest	
Type	Automated
Automation Method	Black box with Jest and React Testing Library, backend deployed on AWS
Details	After the user has successfully logged in, a test request is sent to the backend with the user's authentication token and a response is received confirming the user is authenticated.
Input	Valid email and password - submit button is pressed
Output	Authentication success message

Login: UATInvalidLogin	
Type	Automated
Automation Method	Black box with Jest and React Testing Library, backend deployed on AWS
Details	The login form is filled out with an invalid email and/or password, the submit button is pressed and the user is not authenticated with Firebase which prevents the user from accessing the rest of the application
Input	Valid email and Invalid password - submit button is pressed
Output	Authentication Failed Message

7.2 Current Active Meetings Test Scripts

Unit Tests

Current Active Meetings: GetNoActiveMeetingBackend	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that calls get meeting with the test user ID, it will also create mock zoom data as an input, this zoom mock data will not have any meetings. Assert that the lambda function successfully returns no active meetings.
Input	Mock zoom data with no current active meetings at all
Output	A list of size 0 for the current active meetings

Current Active Meetings: Get1ActiveMeetingBackend	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that calls get meeting with the test user ID, it will also create mock zoom data as an input, this zoom mock data will have one meeting of test users. It will also mock an instance of the DynamoDB database with test users that will be in these meetings. It will set the created test users to be at the same level in the organisation as the test user. Assert that the lambda function successfully returns one active meeting with those users.
Input	Mock zoom data with one current active meeting with participants which the test user should be able to see
Output	A list of size 1 for the current active meetings, with that meeting showing the correct valid participants

Current Active Meetings: Get3ActiveMeetingBackend	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that calls get meeting with the test user ID, it will also create mock zoom data as an input, this zoom mock data will have 3 meetings with test users. It will also mock an instance of the DynamoDB database with test users that will be in these meetings. It will set the created test users to be at the same level in the organisation as the test user. Assert that the lambda function successfully returns 3 active meetings with those users.
Input	Mock zoom data with one current active meeting with participants which the test user should be able to see
Output	A list of size 3 for the current active meetings, with that meeting showing the correct valid participants

Current Active Meetings: GetNoValidActiveMeetingBackend	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that calls get meeting with the test user ID, it will also create mock zoom data as an input, this zoom mock data will have a meeting of test users. It will also mock an instance of the DynamoDB database with test users that will be in these meetings. It will set the created test users to be at a higher level in the organisation than the test user. Assert that the lambda function successfully returns no active meetings.
Input	Mock zoom data with one current active meeting with participants which the test user should not be able to see
Output	A list of size 0 for the current active meetings

Current Active Meetings: GetNoActiveMeetingFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Mock a response from the backend, and ensure that no active meetings are shown when the response has no meetings
Input	Mock backend data that has no current active meetings in its list
Output	A list of size 0 for the current active meetings

Current Active Meetings: Get1ActiveMeetingFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Mock a response from the backend, and ensure that 1 active meeting shows up correctly on the office page
Input	Mock backend data that has one current active meeting in its list
Output	A list of size 1 for the current active meetings

Current Active Meetings: Get3ActiveMeetingFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Mock a response from the backend, and ensure that 3 active meeting shows up correctly on the office page
Input	Mock backend data that has three current active meetings in its list
Output	A list of size 1 for the current active meetings

SAT Tests

Current Active Meetings: SATGetMeetingTest	
Type	Manual
Details	Create a connection to the websocket with wscat in a terminal. Run a test websocket action to create a test meeting with test users in it. Run the test to get the current active meetings, and check to see if the returned meetings contain the test meeting, and that all of the test users are displayed in it. Send a websocket action to close that test meeting. Close the wscat connection.
Test Conditions	One meeting with a few participants is currently active.
Expected Output	A list with one meeting in it with the correct participants in it

UAT Tests

Current Active Meetings: UATGetMeetingTest	
Type	Manual
Details	Create a meeting with a few other users. Then have your user open the office page and check to see if that active meeting shows up on the office page.
Test Conditions	One meeting with a few real participants in it.
Expected Output	Test user is able to see that one current active meeting on the office page and be able to see the participants in it

7.3 Joining a Meeting Test Scripts

Unit Tests

Joining Active Meetings: JoinActiveMeetingFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Mock a response from the backend, the response will have one active meeting that will be able to be joined. It make sure that the zoom link is able to be clicked and you can join the meeting
Input	Mock backend data that has one current active meeting in its list
Output	Test user clicks the link for the one active zoom meeting and is able to join it

SAT Tests

Current Active Meetings: SATJoinMeetingTest	
Type	Manual
Details	Create a connection to the websocket with wscat in a terminal. Run a test websocket action to create a test meeting with test users in it. Run the test to get the current active meetings, then check to see if you can join the active meeting that was returned. Send a websocket action to close that test meeting. Close the wscat connection.
Test Conditions	One meeting with a few test participants in it.
Expected Output	Be able to take the given response and join the meeting with the given link for that one active meeting

UAT Tests

Current Active Meetings: UATJoinMeetingTest	
Type	Manual
Details	Manual black box testing would be the easiest way to test joining active meetings. You will need to have at least two users to test this. One or more will join a meeting, and the user that is testing will try to join the meeting that shows up in the office page.
Test Conditions	One active meeting with a few real participants in it.
Expected Output	User is able to go to the office tab and then view the current active meeting and then be able to join it

7.4 Creating Instant Meeting Test Scripts

Unit Tests - Frontend

Creating Instant Meeting: UnitTestRenderUserSelectionForm	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	Tests that the User Selection form is properly rendered including all users of the app in a checklist and a button to confirm the creation of the instant meeting
Input	React App User Selection Component
Output	Tests pass if all elements exist

Creating Instant Meeting: UnitTestNoBusyUsers	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	The user list should display everyone available, and should test the selection of multiple users for the instant meeting
Input	React App User Selection Component - multiple users selected
Output	Instant Meeting is created with the selected users - Meeting notification appears

Creating Instant Meeting: UnitTestMultipleBusyUsers	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	The user list should display everyone, where busy users have a busy indicator next to their name and are unselectable
Input	React App User Selection Component - multiple users selected
Output	Instant Meeting is created with the selected users - Meeting notification appears

Creating Instant Meeting: UnitTestNoFreeUsers	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	The user list should display all users, however no users should be selectable and as a result the Create Instant Meeting button should be unclickable
Input	React App User Selection Component - no users selected
Output	Instant Meeting can not be created - Creation button is greyed out and unclickable

Creating Instant Meeting: UnitTestNoUsersSelected	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	No users are selected in the User Selection form, so the Create Instant Meeting button should be greyed out and unclickable
Input	React App User Selection Component - no users selected
Output	Instant Meeting can not be created - Creation button is greyed out and unclickable

Unit Tests - Backend

Creating Instant Meeting: UnitTestInvalidFormat	
Type	Automated
Automation Method	Black box unit testing with mocha/chai
Details	Test that the proper error message is returned when the CreateInstantMeeting Lambda is called with an incorrectly formatted request
Input	Lambda Event Create Instant Meeting Request - Incorrectly formatted
Output	Error message

Creating Instant Meeting: UnitTestInvalidUser	
Type	Automated
Automation Method	Black box unit testing with mocha/chai
Details	Test that the proper error message is returned when the CreateInstantMeeting Lambda is called with an invalid user (one that is not registered with the app)
Input	Lambda Event Create Instant Meeting Request - Multiple valid users and one invalid user
Output	Error message

Creating Instant Meeting: UnitTestNoUsers	
Type	Automated
Automation Method	Black box unit testing with mocha/chai
Details	Test that the proper error message is returned when the CreateInstantMeeting Lambda is called with no users
Input	Lambda Event Create Instant Meeting Request - No Users
Output	Error message

Creating Instant Meeting: UnitTestManyUsers	
Type	Automated
Automation Method	Black box unit testing with mocha/chai
Details	Test that the CreateInstantMeeting Lambda returns a success message and that a zoom meeting is properly created with all listed users invited and scheduled instantly
Input	Lambda Event Create Instant Meeting Request - Many users
Output	Zoom meeting is created

SAT

Creating Instant Meeting: SATInstantMeetingCreated	
Type	Manual
Details	Use wscat to send a websocket event to our CreateInstantMeeting Lambda with a number of mock users, and manually check that a zoom meeting is launched and all users are invited/notified by simulating multiple clients
Test Conditions	Websocket Instant Meeting Request Message - Multiple Users
Output	Zoom meeting is launched on the correct user clients

Creating Instant Meeting: SATInstantMeetingNotCreated	
Type	Manual
Details	Use wscat to send a websocket event to our CreateInstantMeeting Lambda with no users, and manually check that a zoom meeting is not launched and no clients receive any meeting notifications
Test Conditions	Websocket Instant Meeting Request Message - No Users
Output	Zoom meeting not launched - no meeting notifications

UAT

Creating Instant Meeting: UATInstantMeeting	
Type	Manual
Details	From the frontend an instant meeting is created with a number of mock users, we will setup a number of clients and manually check that each invited client immediately received a notification for a new meeting, and ensure that non-invited clients do not receive a notification
Test Conditions	Instant Meeting requested through frontend Office Instant Meeting Form
Output	Zoom meeting is created and notifications are received on all invited clients

7.5 View Your Groups & Members Locations Scripts

Unit Tests

View Your Groups & Members Locations: ShowNoGroupsBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that includes the request to return all of the group information, and mock an instance of the DynamoDB database with a Groups table that has no groups in it using proxyquire. Assert that upon receiving the event to return the list of groups, the lambda function reads the mocked DynamoDB, gets an empty set of groups as the response, and thus, returns an empty list as its result.
Input	Mock lambda event and DynamoDB table with no group entries.
Output	An empty list of groups returned.

View Your Groups & Members Locations: ShowMultipleGroupsBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that includes the request to return all of the group information, and mock an instance of the DynamoDB database with a Groups table that has multiple groups using proxyquire. This test will also have to mock the Zoom API request that returns the user's current locations using proxyquire. Assert that the lambda function correctly receives the request from the event, sends a read to get the group information from the DynamoDB, and for each user the groups (accounting for duplicates), uses the mocked Zoom API to get the location for each of the users. Then assert that this data is bundled into the correct format, and returned as the result of the lambda function.
Input	Mock lambda event and DynamoDB table with a list of groups, and a mocked Zoom API response to get the user locations.
Output	A list of groups to be displayed in the correct format, with the locations of each user in every group added to the response object.

View Your Groups & Members Locations: ShowNoGroupsFrontend	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	Mock a response from the backend that contains an empty list of groups, and ensure that no groups are displayed in the frontend.
Input	Mocked backend response that contains an empty list of groups.
Output	The size of the groups list displayed in the frontend is zero.

View Your Groups & Members Locations: ShowMultipleGroupsFrontend	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library
Details	Mock a response from the backend that contains groups, and their user information. Ensure that the number of groups displayed is correct, and that the information displayed in the card for each group is the same as what the mocked backend response provides.
Input	Mocked backend response that contains multiple groups to display.
Output	The size of the groups list, and information on each card displayed in the frontend matches what is provided in the mocked backend response.

SAT

View Your Groups & Members Locations: ShowNoGroupsSAT	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, send a request all groups for a user that has not previously created any groups. Ensure that the response from the backend contains an empty list of groups.
Test Conditions	For a newly created test user with now groups, send the wscat action to get the list of groups for that user.
Expected Output	A WebSocket response that contains an empty list of groups.

View Your Groups & Members Locations: ShowMultipleGroupsSAT	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, first use our test websocket action, to create a new test meeting and add test users to it. Then, use wscat to send the WebSocket action to create a new group that contains those test users. Finally, send the WebSocket action to request the list of groups, and ensure that the newly created group is returned, and that the location for each of the test users is correct. After testing is complete, ensure that the WebSocket action to end the test meeting and delete the test users is sent as well.
Test Conditions	For a user that has previously saved groups, call the WebSocket action to get the list of groups using wscat.
Expected Output	All of the user's previously saved groups are returned, and for each of the members in the groups, their current location is accurate.

UAT

View Your Groups & Members Locations: ShowNoGroupsUAT	
Type	Manual
Details	Ensure that the account being used has not previously created any groups, or has removed all of their previously created groups. Navigate to the groups page, and ensure that the only item listed is the option to create a new group, with no formatting issues.
Test Conditions	When logged into the app with a user that has not previously created any groups, navigate to the groups page.
Expected Output	Ensure that there are no groups displayed and the page has no formatting issues.

View Your Groups & Members Locations: ShowMultipleGroupsUAT	
Type	Manual
Details	With multiple users, have a variety of users join different active meetings. Then, from the account of the user not in a meeting, Navigate to the groups page, and create groups with those users in them. Ensure that all of the groups are displayed in the correct format, and that all of the user locations are correct for those users that had previously joined active meetings (assuming they had been added to the groups).
Test Conditions	When logged into the app with a user that has previously created groups, have other users that are part of those groups join active meetings.
Expected Output	When the initial user navigates to the groups page, ensure that all of their previously created groups are displayed and all of the member locations are accurate.

7.6 Create New Group

Unit Tests

Create New Group: GroupIsCreated2MembersBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that includes the group information, and the 2 members to be added, and mock an instance of the DynamoDB database with a Groups table using proxyquire. Then assert that the lambda function successfully makes a write call to the mocked database with the group name and 2 members. After this, assert that the mocked database now contains the new group entry.
Input	A mocked event to the lambda function and a mocked instance of our DynamoDB
Output	The mocked DynamoDB has a new entry in its groups table, and that group contains the 2 members listed in the mocked lambda event.

Create New Group: GroupIsCreated5MembersBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Mock an event to the lambda function with the lambda-tester library that includes the group information, and the 5 members to be added, and mock an instance of the DynamoDB database with a Groups table using proxyquire. Then assert that the lambda function successfully makes a write call to the mocked database with the group name and 5 members. After this, assert that the mocked database now contains the new group entry.
Input	A mocked event to the lambda function and a mocked instance of our DynamoDB
Output	The mocked DynamoDB has a new entry in its groups table, and that group contains the 5 members listed in the mocked lambda event.

Create New Group: GroupNameAlreadyExistsBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Use proxquire to mock a database that already contains an entry with the same group name sent to the lambda from the mocked lambda-tester event. Assert that the lambda function returns an error response that informs the user that the group can not be created because an entry with the same name already exists. Assert that no new entry has been added to the mocked database.
Input	A mocked event to the lambda function and a mocked instance of our DynamoDB that contains a group entry with the same name provided in the mocked event.
Output	An error message is returned and no new entry has been added to the mocked DB.

Create New Group: GroupIsCreated2MembersFrontend	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library.
Details	Mock a response from the backend, and ensure that the total number of displayed groups is updated (increased by 1), and that the new displayed group has the correct information and 2 members based on the mocked backend data.
Input	Mocked backend response from the backend that includes the new group with 2 members to be listed.
Output	The new group list size has increased by 1, and the information for the new group information is correct based on the mocked backend response with 2 members.

Create New Group: GroupIsCreated5MembersFrontend	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library.
Details	Mock a response from the backend, and ensure that the total number of displayed groups is updated (increased by 1), and that the new displayed group has the correct information, and that the member overflow is handled properly (only the first 4 members should be shown in the group card).
Input	Mocked backend response from the backend that includes the new group with 5 members to be listed.
Output	The new group list size has increased by 1, and the information for the new group information is correct based on the mocked backend response. The group card must only show the first 4 members without being expanded.

Create New Group: GroupNameAlreadyExistsFrontend	
Type	Automated
Automation Method	Black box unit testing with Jest and React Testing Library.
Details	Mock an error response for the backend that explains that the group could not be created because one with the name already exists. Ensure that no new groups are displayed, and that the error message is correctly displayed when the user tries to save the new group.
Input	Mocked error response from the backend.
Output	The size of the list of groups displays remains the same, and the error message from the mocked backend is displayed.

SAT

Create New Group: GroupIsCreatedSAT	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, send a create group action with the group information and members. Ensure that the correct response is received with the group information that the frontend should display.
Test Conditions	Send a wscat WebSocket action to create a group with the group and members information.
Expected Output	The WebSocket responds with a new list of groups to show, including the new group that was just added.

Create New Group: GroupNameAlreadyExistsSAT	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, send a create group action with group information and members. Wait for the reply that says the group was successfully created. Send another create group action with the same name as the first group. Ensure an error-reply is sent from the backend that says that the group could not be created because a group with the same name already exists.
Test Conditions	Send a wscat WebSocket action to create a group with a group name that already exists as one of the user's previously created groups.
Expected Output	The WebSocket responds with an error saying that a group with the provided name already exists.

UAT

Create New Group: GroupIsCreatedUAT	
Type	Manual
Details	Navigate to the Groups page, and click create group. Enter the group information and select a list of users that should be added to it. Click save and ensure that the created group is correctly displayed in the list of all groups with the correct information.
Test Conditions	When logged into the app, a user clicks the create group button and enters the group information.
Expected Output	After the user clicks save, the new group is correctly displayed in the list of groups with the correct information.

Create New Group: GroupNameAlreadyExistsUAT	
Type	Manual
Details	Navigate to the Groups page, and click create group. Enter the group information and select a list of users that should be added to it. Click save and ensure that the created group is correctly displayed in the list of all groups with the correct information. Then, try to create another group with the same name as the first group. Ensure that the UI displays an error describing that 2 groups can not be created with the same name, and that no duplicate group is present in the list of all groups.
Test Conditions	When logged into the app a user clicks the create group button and enters the group name of a group that they have previously created.
Expected Output	After the user clicks save, an error pops up saying that a group can not be created with the same name, and the group is not displayed in the Groups page.

7.7 Calling Your Group Scripts

Unit Tests

Calling Your Group Scripts: UnitGroupsCall2Members	
Type	Automated
Automation Method	Automated white box testing with Mocha/Chai, lambda-test and proxyquire.
Details	Mock an event to the lambda function with the lambda-tester library that includes the ID of the group, and mock an instance of the DynamoDB database with a Groups table using proxyquire which includes the users Zoom IDs. Assert that the lambda function successfully retrieves the Zoom IDs from the database and creates a Zoom call with an invite link.
Input	Current user ID and the ID of the group to call
Output	Invite link to the call

Calling Your Group Scripts: UnitGroupsCall5Members	
Type	Automated
Automation Method	Automated white box testing with Mocha/Chai, lambda-test and proxyquire.
Details	Mock an event to the lambda function with the lambda-tester library that includes the 5 group participants, and mock an instance of the DynamoDB database with a Groups table using proxyquire which includes the users Zoom IDs. Assert that the lambda function successfully retrieves the Zoom IDs from the database and creates a Zoom call with an invite link.
Input	Current user ID and the ID of the group to call
Output	Invite link to the call

Calling Your Group Scripts: UnitGroupsCall2MembersUserBusy	
Type	Automated
Automation Method	Automated white box testing with Mocha/Chai, lambda-test and proxyquire.
Details	Mock an event to the lambda function with the lambda-tester library that includes the 2 group participants, and mock an instance of the DynamoDB database with a Groups table using proxyquire which includes the users Zoom IDs. Also have to mock the getActiveMeetings function in the backend which will include the other group participant. Assert that the lambda does not include the busy user in the response data, and that it includes a notification to the user who called the group that a user is busy.
Input	Current user ID and the ID of the group to call
Output	Invite link to the call and notification

SAT

Calling Your Group Scripts: SATGroupsCallTest	
Type	Manual
Details	Spin up two instances of wscat in different terminals, create a connection to the websocket in both terminals. Run an action to create a group from one instance with the other instance user in it. Run the test to call the members of the group, check on the other instance to see if the other instance received the invite data. Run an action to close that test call. Close the wscat connection on both instances.
Test Conditions	Two terminal instances running wscat connected to the websocket, a group created by one of the instances
Expected Output	An invite link to the Zoom call to all connected parties

Calling Your Group Scripts: SATGroupsCall2MembersUserBusy	
Type	Manual
Details	Spin up two instances of wscat in different terminals, create a connection to the websocket in both terminals. Run an action to create a group from one instance with the other instance user in it. Run an action to create another group and connect to it from the second instance. Run the test to call the members of the group, check on the other instance to ensure they didn’t receive an invite link. Assert that the Run an action to close that test call. Close the wscat connection on both instances.
Test Conditions	Two terminal instances running wscat connected to the websocket, a group created by one of the instances, one of the instances in another call
Expected Output	A message to the calling instance that the other user was busy

UAT

Calling Your Group Scripts: UATGroupsCallTest2Members	
Type	Manual
Details	Create a group with one other member, ensure the other member has the app open and is not busy, call the group. Assert that both users see a popup with an invite link to the new meeting.
Test Conditions	Two users, a group created by one user with the other user in it
Expected Output	Both users see a popup in the UI with a clickable invite to the call

Calling Your Group Scripts: UATGroupsCallTest5Members	
Type	Manual
Details	Create a group with 4 other members, ensure the other members have the app open and are not busy, call the group. Assert that all users see a popup with an invite link to the new meeting.
Test Conditions	Five users, a group created by one user with the other users in it
Expected Output	All users see a popup in the UI with a clickable invite to the call

Calling Your Group Scripts: UATGroupsCallUserBusy	
Type	Manual
Details	Create a group with one other member, ensure the other member has the app open and has joined another call, call the group. Assert that the other user does not see a popup in the app and that the caller sees a notification stating which users in the group were busy and could not be called.
Test Conditions	Two users, a group created by one user with the other user in it, the other user joins a different call
Expected Output	The calling user sees a popup saying the other user is busy

7.8 Add User to Favourites Scripts

Unit Tests

Add User to Favourites Scripts: UnitAddUserFavourite	
Type	Automated
Automation Method	Automated white box testing with Mocha/Chai, lambda-test and proxyquire.
Details	Mock an event to the lambda function with the lambda-tester library that includes the new favourite user ID, and mock an instance of the DynamoDB database with a User table and a specific user entry using proxyquire. Then assert that the lambda function successfully makes a write call to the mocked database with the new favourite user ID. After this, assert that the mocked database now contains the new favourite user ID.
Input	Current user ID and ID of the new favourite member
Output	Current user’s favourites list

Add User to Favourites Scripts: UnitUserAlreadyFavourite	
Type	Automated
Automation Method	Automated white box testing with Mocha/Chai, lambda-test and proxyquire.
Details	Mock an event to the lambda function with the lambda-tester library that includes the new favourite user ID, and mock an instance of the DynamoDB database with a User table and a specific user entry using proxyquire. The favourites array for this user entry should already include the user ID of the new favourite user. Then assert that the lambda function fails to make a write call to the mocked database with the new favourite user ID. After this, assert that the lambda returns an error message stating that the new favourite user cannot be added as it already exists. Then assert that the favourite user still exists in the mocked database.
Input	Current user ID and ID of the new favourite member
Output	Error message about existing user

SAT

Add User to Favourites Scripts: SATAddUserFavourite	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, send an add favourite action with the new favourite user ID. Ensure that the correct response is received with the new list of favourites that the frontend should display.
Test Conditions	A terminal instance running wscat with the current user with an empty favourites list, and the user ID of another user
Expected Output	The current user’s updated favourites list with the new favourite user in it

Add User to Favourites Scripts: SATUserAlreadyFavourite	
Type	Manual
Details	Connect to the WebSocket using wscat in a terminal, send an add favourite action with the new favourite user ID. Wait for the reply that says the group was successfully created. Send another add favourite action with the same user ID as the first favourite. Ensure an error-reply is sent from the backend that says that the user could not be added because the user is already in the favourites list.
Test Conditions	A terminal instance running wscat with the current user with a favourites list containing another user ID, and the user ID of the other user
Expected Output	A message to the calling instance that the other user was already in their favourites list

UAT

Add User to Favourites Scripts: UATAddUserFavourite	
Type	Manual
Details	Click on the favourites button in the nav bar, click on a user on the left side to add them to your favourites list on the right side. This should automatically update in the database and the modal can be closed. Reload the application and open the favourites modal again and assert that the user still remains in the user's favourites list.
Test Conditions	Current user logged in, has an empty favourites list
Expected Output	The new favourite user appears in the favourite modal

7.9 Viewing Your Favourite Members Locations Scripts

Unit Tests

Viewing Your Favourite Members Locations: UnitNoFavouritesToView	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Simulate a call to the lambda by a user who has no favourites.
Test Conditions	Create a fake dynamo entry for a user with a userID who has no favourites using proxyquire. Call the function with that userID.
Expected Output	Empty List

Viewing Your Favourite Members Locations: UnitAllFavouritesAvailable	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Simulate a call to the lambda by a user who has five favourites, none of whom are in a meeting.
Test Conditions	Create five fake dynamo entries using proxyquire to mock various users, each including a userID and zoomID. Also mock the zoom API results - again with proxyquire - returning two meetings with participants that do not match given zoomIDs. Lastly, create a caller user using proxyquire with a userID and a favourites list including all the userIDs from above. Call the function with the caller userID.
Expected Output	A list containing all five userIDs associated without a zoom location.

Viewing Your Favourite Members Locations: UnitSomeFavouritesInCall	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Simulate a call to the lambda by a user who has five favourites, three of whom are currently in a meeting.
Test Conditions	Create five fake dynamo entries using proxyquire to mock various users, each including a userID and zoomID. Also mock the zoom API results - again with proxyquire - returning two meetings each with a participants list which includes one of the mocked dynamo zoom IDs . Lastly, create a caller user using proxyquire with a userID and a favourites list including all the userIDs from above. Call the function with the caller userID.
Expected Output	A list containing all five userIDs - two of which are associated with a distinct zoom location.

Viewing Your Favourite Members Locations: UnitInvalidUserID	
Type	Automated
Automation Method	White box unit testing with mocha/chai, lambda-tester and proxyquire
Details	Simulate a scenario where the frontend sends a request with a userID that does not exist in our database.
Test Conditions	Create a mocked dynamo table using proxyquire containing no entries. Call the function with a userID that is not in our mocked table.
Expected Output	A User-Not-Found Error

SAT

Viewing Your Favourite Members Locations: SATNoFavouritesToView	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who has no favourites.
Test Conditions	Send a wscat request with an ID associated with a test user in our dynamo table with no favourites.
Expected Output	Empty List

Viewing Your Favourite Members Locations: SATAllFavouritesAvailable	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who has five favourites, none of which are in a zoom call.
Test Conditions	Create five test users in our dynamo tables, each with a userID and a ZoomID. Create an additional test user in our dynamo table with a userID, zoomID, and a favourites list containing the above users. There is a test endpoint which adds two predetermined hard coded users to a zoom call. Send a wscat request with the ID associated with the test user with favourites. None of those favourites were added to a zoom meeting in the test endpoint.
Expected Output	A list of five user IDs pertaining to the favourites - each of which have no location returned.

Viewing Your Favourite Members Locations: SATSomeFavouritesInCall	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who has five favourites, two of which are in a distinct zoom call.
Test Conditions	Create five test users in our dynamo tables, each with a userID and a ZoomID. Create an additional test user in our dynamo table with a userID, zoomID, and a favourites list containing the above users. There is a test endpoint which adds two predetermined hard coded users to a zoom call. Send a wscat request with the ID associated with the test user with favourites. Two of those favourites were added to a zoom meeting in the test endpoint.
Expected Output	A list of five user IDs pertaining to the favourites - two of which have unique zoom location data returned.

Viewing Your Favourite Members Locations: SATInvalidUserID	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who has somehow not been properly added to our database.
Test Conditions	Send a wscat request with an ID that does not exist in our dynamo table.
Expected Output	A User-Not-Found Error.

UAT

Viewing Your Favourite Members Locations: UATNoFavouritesToView	
Type	Manual
Details	Mimic a user story where a user who has no favourites attempts to view their favourite locations.
Test Conditions	Login to a test account which is registered in the DB with no favourites. Open favourites modal.
Expected Output	No entries in the favourites modal.

Viewing Your Favourite Members Locations: UATAllFavouritesAvailable	
Type	Manual
Details	Mimic a user story where a user who has five favourites attempts to view their favourite locations - and all are available.
Test Conditions	Login to a test account which is registered in the DB with no favourites. Add several test users as favourites - none of whom are in a meeting. Open favourites modal.
Expected Output	Five entries in the favourites modal, all marked as Available.

Viewing Your Favourite Members Locations: UATSomeFavouritesInCall	
Type	Manual
Details	Mimic a user story where a user who has five favourites attempts to view their favourite locations - and all are available.
Test Conditions	Login to a test account which is registered in the DB with no favourites. Add several test users as favourites - none of whom are in a meeting. Call a test endpoint using wscat which adds two of the favourites list to a zoom meeting. Open favourites modal.
Expected Output	Five entries in the favourites modal, two of which marked as In Meeting, with zoom link info available.

7.10 Viewing Your Past/Present/Future Meetings

Unit Tests

View Your Past/Present/Future Meetings: MeetingsBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	A test which checks that when the frontend prompts for the current user's meetings, it will pull all their past, ongoing and future meetings, and return them in a format that can be used by the frontend to then display on the calendar view.
Input	An API request for the meetings (past present and future) of an individual.
Expected Output	The list of meetings the individual will have, including the participants, the times, and the meeting topics, presented in a list of tuples (Or an empty list if no meetings are found).

View Your Past/Present/Future Meetings: RecordingsBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	A test which checks that when given a list of past meetings, all relevant recordings are pulled from zoom and returned together with their correlated meetings in a format that can then be used to display recording links on the frontend with associated meetings.
Input	A zoom API request that gets all relevant existing recordings for meetings.
Expected Output	A list of all zoom recording links for requested meetings, paired to the meetings they are associated with.

View Your Past/Present/Future Meetings: DisplayMeetingsFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Displays meetings and recordings fetched from the backend in the calendar view, with blocks of time representative of meeting lengths being allocated, and information regarding the meeting topic, participants, and possible recordings for past meetings being displayed.
Test Conditions	Page load (Empty calendar view)
Expected Output	Calendar view loaded with logged in user’s meetings

SAT

View Your Past/Present/Future Meetings: FetchMeetingsSAT	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who requests to see their meetings.
Test Conditions	Send a wscat request with an ID associated with our current user in our dynamo table pulling their meetings from the zoom API.
Expected Output	A list of tuples containing the information regarding all past, ongoing and future meetings.

View Your Past/Present/Future Meetings: FetchRecordingsSAT	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who requests to see a recording for a past meeting.
Test Conditions	Send a wscat request with an ID associated with our current user in our dynamo table pulling their recorded past meetings from the zoom API.
Expected Output	A list of all recordings from past meetings, paired with the meeting topic.

UAT

View Your Past/Present/Future Meetings: FetchMeetingsUAT	
Type	Manual
Details	Mimic a user story where a user wants to see their own calendar.
Test Conditions	User loads calendar view
Expected Output	User sees all their meetings

7.11 Viewing Other Users Availability

Unit Tests

View Other Users Availability: AvailabilitiesBackend	
Type	Automated
Automation Method	Black box unit testing with mocha/chai, lambda-tester and proxyquire
Details	A test which checks that when the frontend prompts for the current user's availability, that returns said user's availability (past,ongoing, future).
Input	An API request for the availability (past present and future) of an individual.
Expected Output	The list of availability for an individual, with starting times and durations.

View Other Users Availability: DisplayAvailabilitiesFrontend	
Type	Automated
Automation Method	White box unit testing with Jest and React Testing Library
Details	Displays availability on a given user fetched from the backend in the calendar view, with blocks of time representative of meeting lengths being allocated.
Test Conditions	Page load (Empty calendar view)
Expected Output	Calendar view loaded with logged in user's availability

SAT

View Other Users Availability: FetchAvailabilitySAT	
Type	Manual
Details	Simulate a call over our websocket from the frontend from a user who requests to see the availability of another user.
Test Conditions	Send a wscat request with an ID associated with a user in our dynamo table pulling their meetings from the zoom API.
Expected Output	A list of timeslots during which the target user is not available/occupied.

UAT

View Other Users Availability: FetchAvailabilityUAT	
Type	Manual
Details	Mimic a user story where a user views the calendar availabilities of another user
Test Conditions	User loads calendar view and selects another user to view their availabilities
Expected Output	User sees the selected other user's availabilities

7.12 Browser Notifications Test Scripts

SAT Tests

Browser Notifications: SAT15MinWebsocketNotificationTest	
Type	Manual
Details	Create two connections to the websocket with wscat in two separate terminals, and use one to send an action that triggers the 15-minute event notification for the other connected terminal. Ensure that the second terminal gets the 15-minute response in their terminal without having to perform any actions
Test Conditions	Send a wscat request to create a 15-minute event notification for a meeting.
Expected Output	Terminal gets a 15-minute event notification as a response from the server

Browser Notifications: SAT30MinWebsocketNotificationTest	
Type	Manual
Details	Create two connections to the websocket with wscat in two separate terminals, and use one to send an action that triggers the 30-minute event notification for the other connected terminal. Ensure that the second terminal gets the 30-minute response in their terminal without having to perform any actions
Test Conditions	Send a wscat request to create a 3-minute event notification for a meeting.
Expected Output	Terminal gets a 30-minute event notification as a response from the server

UAT Tests

Browser Notifications: UAT15MinuteNotificationTest	
Type	Manual
Details	Create a meeting 20 minutes away, while logged into another invited Zoom account simultaneously set a timer, at the 5 minute mark check for an appropriate popup
Test Conditions	Have another user create a scheduled meeting that is 20 minutes away
Expected Output	Have your user see a browser notification for a meeting in 15 minutes, 5 minutes after the other user created the scheduled meeting

Browser Notifications: UAT30MinuteNotificationTest	
Type	Manual
Details	Create a meeting 35 minutes away, while logged into another invited Zoom account simultaneously set a timer, at the 30 minute mark check for an appropriate popup.
Test Conditions	Have another user create a scheduled meeting that is 35 minutes away
Expected Output	Have your user see a browser notification for a meeting in 30 minutes, 5 minutes after the other user created the scheduled meeting

7.13 Popup Modal to Join Scheduled Call

SAT

Popup Modal to Join Scheduled Call: SATJoinMeetingPopup	
Type	Manual
Details	Simulate sending a meeting start trigger to the frontend using wscat (WebSocket cat)
Input	Set up two terminals connected by the websocket. Send an event trigger from one terminal to the other indicating a meeting has started.
Expected Output	The second terminal has properly received the event trigger.

UAT

Popup Modal to Join Scheduled Call: UATJoinMeetingPopupJoin	
Type	Manual
Details	Simulate a user story where a user's meeting has just begun on zoom, and they want to click a popup to join the meeting.
Input	Schedule a meeting, either through our app or through zoom, for a point in the near future. Wait for the scheduled meeting time while the app displays no popups.
Expected Output	At the scheduled meeting time, expect to see a popup prompting the user to join the meeting. Upon clicking the zoom join button, expect to be redirected to Zoom.

Popup Modal to Join Scheduled Call: UATJoinMeetingPopupIgnore	
Type	Manual
Details	Simulate a user story where a user's meeting has just begun on zoom, and they don't want to join.
Input	Schedule a meeting, either through our app or through zoom, for a point in the near future. Wait for the scheduled meeting time while the app displays no popups.
Expected Output	At the scheduled meeting time, expect to see a popup prompting the user to join the meeting. Click to close the popup.

7.14 Non-Functional Test - Many Concurrent Users Test Scripts

SAT Tests

Scale Design Capacity: ManyConcurrentUsersSAT	
Type	Automated
Automation Method	Automated performance testing using JMeter
Details	This involves creating a recipe in JMeter which will explicitly detail the endpoints and order the virtual users to join a meeting. This will simulate many concurrent users logging in and creating a new meeting
Input	A JMeter recipe to simulate 20+ users interacting with endpoints concurrently
Expected Output	A JMeter output with less than 1% error rate and 3 seconds or less response time for all actions

UAT Tests

Scale Design Capacity: ManyConcurrentUsersUAT	
Type	Manual
Details	This will involve multiple team members signing into more than one user account and performing various actions like creating meetings in order to assess the performance of the system
Input	20+ real accounts interacting with the web application
Expected Output	No noticeable performance effects (i.e. no 400 errors, incorrect state, etc)

7.15 Non-Functional Test - Response Time Test Scripts

Unit Tests - Front End

Scale Design Capacity: FilterAndDisplaySearchResultsSAT	
Type	Automated
Automation Method	Automated black box testing with Jest and React Testing Library
Details	These tests will ensure that any front-end sorting/filtering of searches will takes less than 1 second to complete(assuming a 1 second max RTT for the back-end API call)
Input	A call to the search function with a timer attached
Expected Output	A total time of execution which is less than 1 second

SAT

Scale Design Capacity: FilterAndDisplaySearchResultsSAT	
Type	Automated
Automation Method	Automated performance testing using JMeter
Details	This use case will involve creating a JMeter recipe to perform a search with an already logged in user X numbers of times.
Input	A virtual user performing a search X numbers of times repeatedly
Expected Output	Jmeter will record an average response time of all actions that is less than 2 seconds.

UAT

Scale Design Capacity: FilterAndDisplaySearchResultsUAT	
Type	Manual
Details	This will involve the team members signing into the application with multiple test accounts each and performing functions to ensure that speed and accuracy of the system is adequate.
Input	A real user performing a search
Expected Output	A response time of less than 3 seconds recorded in the chrome developer tools

7.16 Non-Functional Test - Scale Design Capacity Test Scripts

SAT

Scale Design Capacity: JoinLargeParticipantMeetingTestSAT	
Type	Automated
Automation Method	Automated performance testing using JMeter
Details	This involves creating a recipe in JMeter which will explicitly detail the endpoints and order the virtual users to join a meeting. Then the test user will try to join that meeting
Input	Set up an active zoom meeting with a large number of test participants.
Expected Output	Test user is able to join the large active zoom meeting

UAT

Scale Design Capacity: JoinLargeParticipantMeetingTestUAT	
Type	Manual
Details	Have a large number of users, preferably near the limit for the maximum number of participants in a zoom meeting. Have them all join a meeting room together. Now, the user who is testing will log into the application and navigate to the office page. They then should be able to join the meeting where all the participants are in.
Input	Set up an active zoom meeting with a large number of real participants.
Expected Output	User is able to join the large active zoom meeting