Anthony Krivonos (ak4483)
Kenneth Chuen (kc3334)
Kevin Wong (hw2735)
William Pflueger (fwp2108)

December 1st, 2020
Adv. SWE Assignment T5
Prof. Gail Kaiser
Team: Starmen

Github Link: https://github.com/anthonykrivonos/4156-Starmen

Part 1 – User Stories

Login as an HCP and view my profile:

As an HCP, I want to be able to login with my Google account and view and edit my current upMed account information. If I am not already logged in, I want to be able to easily create an account with my Google account, business details, and 10-digit National Provider Identifier (NPI). I should be able to login via my Google account and then take me to the HCP portal, where I should be able to view the information that is currently in my profile and edit any fields, such as phone numbers or my hours. My conditions of satisfaction is to login, view my profile, edit my profile and save my edits.

Access patient's own health records:

As a patient, I want my health records to be kept securely with easy access so that I can take reference to them during my consultation. I should be able to input some health data such as whether or not I'm a smoker, whether or not I drink, my height, and my weight. My conditions of satisfaction are able to review all of my records and to edit basic personal information.

View, add and edit patient's health records:

As an HCP, I want to view, add, and modify my patients' generic (text-based) healthcare records. My conditions of satisfaction are able to conveniently view, add, update and edit health records if I am assigned to my patient.

Keep a schedule of my appointments:

As a HCP, I need to schedule appointments with all my patients so that I can meet my patients in an orderly manner and use my time efficiently. My conditions of satisfaction are able to use a calendar to schedule

appointments and notify my patients about their appointments.

Virtually consult healthcare providers (HCPs):

As a patient, I want to meet my HCP virtually so that I can obtain healthcare advice and consultation anywhere and free from COVID. My conditions of satisfaction are able to conduct video conferencing with my HCP.

Part 2 – Test Plan

Frontend

The major input tests in the frontend include everything that follows. Since we created smart components that disallow form submission when inputs fail their validation requirements, all input tests have neatly been placed into *upmed-web/tst/utils/Validator.spec.ts*. All other unit tests for utility functions are in the same folder, whereas an extensive amount of frontend automation and rendering tests lie in the adjacent */components* and */pages* folders. Validation is done similarly for both patient and doctor information.

Hours

Hours objects contain the seven days of the week as keys and dictionary values containing the startTime and endTime of the workday, in UTC minutes of the day.

Equivalence Partition 1: (Valid) all startTimes and endTimes between 0 and 1440 and all startTimes <= endTimes

Equivalence Partition 2: (Valid) all startTimes and endTimes valid and for some day, startTime == endTime == -1 (to mark day as closed)

Equivalence Partition 3: (Invalid) some startTime < 0

Equivalence Partition 4: (Invalid) some startTime > 1440

Equivalence Partition 5: (Invalid) some startTime > endTime

Equivalence Partition 6: (Invalid) some endTime > 1440

Equivalence Partition 7: (Invalid) some startTime == -1 and endTime != -1 for the same day

Equivalence Partition 8: (Invalid) some startTime != -1 and endTime == -1 for the same day

Boundary Condition 1: (Valid) some startTime == endTime == 0

Boundary Condition 2: (Valid) some startTime == endTime == 1440

Boundary Condition 3: (Valid) some startTime == 0, some endTime == 1440

Names

Names, such as first name, last name, business name, and specialty name (i.e. Dentistry) are run through basic length and bad word checks to ensure proper names are inputted.

Equivalence Partition 1: (Valid) a name contains alphabetical characters and spaces, and does not have bad words in them

Equivalence Partition 2: (Valid) a name contains solely alphabetical characters and does not have bad words in them

Equivalence Partition 3: (Valid) a name is blank

Equivalence Partition 4: (Invalid) a name contains a non-alphabetic character

Equivalence Partition 5: (Invalid) a name contains purely spaces

Equivalence Partition 6: (Invalid) a name contains a bad word

Boundary Condition 1: (Valid) a name is a single alphabetic character

Boundary Condition 2: (Invalid) a name is a single non-alphabetic character

Generic Text

Generic text validation is used for doctor's notes and appointment notes. This text solely goes through a bad word validator.

Equivalence Partition 1: (Valid) text is nonempty and does not have bad words in it

Equivalence Partition 2: (Invalid) text is nonempty and has bad words in it

Boundary Condition 1: (Valid) text is empty

Phone Numbers

Phone numbers are assumed to have the United States country code are only valid when in the form XXX-XXX-XXXX, where all Xs are valid digits, or empty.

Equivalence Partition 1: (Valid) phone number is in the form XXX-XXX-XXXX, where all Xs are valid digits

Equivalence Partition 2: (Valid) phone number is empty

Equivalence Partition 3: (Invalid) phone number is non-empty and not in the form XXX-XXX-XXXX

Boundary Condition 1: (Invalid) phone number is non-empty and in the form XXX-XXX-XXXX, but not all Xs are digits

Emails

Emails are always in the form X@Y.Z, where X, Y, and Z are non-space characters.

Equivalence Partition 1: (Valid) email is in the form X@Y.Z with the above conditions

Equivalence Partition 2: (Invalid) email does not contain @ or .

Boundary Condition 1: (Invalid) invalid is in the form @Y.Z

Boundary Condition 2: (Invalid) invalid is in the form X@.Z

Boundary Condition 3: (Invalid) invalid is in the form X@Y.

Boundary Condition 4: (Invalid) invalid is in the form XY.Z or XYZ

NPI

National Provider Identifiers (NPIs) are exactly 10 digit-strings with no special characters, or just empty.

Equivalence Partition 1: (Valid) NPI is a 10-digit string

Equivalence Partition 2: (Valid) NPI is empty

Equivalence Partition 3: (Invalid) NPI contains 10 characters that are not all digits

Boundary Condition 1: (Invalid) NPI contains 10 valid digits, plus extra characters

Boundary Condition 2: (Invalid) NPI is a digit string of length greater than 10

Boundary Condition 3: (Invalid) NPI is a digit string of length less than 10

Height

Height and weight were tested manually, as we created another sophisticated Counter component that ensures only correct inputs may be inputted by the user, either via text input or by pressing the – and + buttons.

Equivalence Partition 1: (Valid) height >= 0 and height <= 244

Equivalence Partition 2: (Invalid) height < 0

Equivalence Partition 3: (Invalid) height > 244

Boundary Condition 1: (Invalid) height == 0

Boundary Condition 2: (Invalid) height == 244

Weight

Weight also tested manually, as mentioned above.

Equivalence Partition 1: (Valid) weight >= 0 and weight <= 227

Equivalence Partition 2: (Invalid) weight < 0

Equivalence Partition 3: (Invalid) weight > 227

Boundary Condition 1: (Invalid) weight == 0

Boundary Condition 2: (Invalid) weight == 227

Calendar

Appointment times, which are recorded via the FullCalendar plugin, are manually tested due to Jest's inability to change states post-render.

Equivalence Partition 1: (Valid) appointment time > current date

Equivalence Partition 2: (Invalid) appointment time < current date

Boundary Condition 1: (Invalid) appointment time == current date

Radio buttons (such as for Smoker? option) were omitted because the correct output was ensured by providing choices to the user in the UI.

Link to frontend automated test suite in GitHub (see scripts in package.json for each test).

Some examples of boundary testing (for email) follow:

```
it('email.valid', () => {
    expect(Validator.email('my@email.com')).equal(true)
    expect(Validator.email('firstname@gmail.net')).equal(true)
})

it('email.invalid.noAtOrDot', () => {
    expect(Validator.email('myemail@com')).equal(false)
    expect(Validator.email('myemail.com')).equal(false)
    expect(Validator.email('myemailcom')).equal(false)
})

it('email.invalid.AtYDotZ', () => {
    expect(Validator.email('@email.com')).equal(false)
})

it('email.invalid.XAtDotZ', () => {
    expect(Validator.email('myemail@.com')).equal(false)
})
```

Backend

The major input tests in the backend include everything that follows. It is assumed the input from frontend are tested to be valid, all unit tests have neatly been placed into upmed-api/tst/api.

upweb-api/src/appointment/appointment.py

This package contains all the endpoints for the frontend to interact with, the data will be transmitted to its corresponding helper function. All the functions and methods within this package does nothing besides transmitting the post data to and from the helper function, and each function contains only 3 lines of code. Hence the tests (appointment_endpoint_test) only tests whether all the helper functions are called.

upweb-api/src/appointment/appointment_helper.py

This package contains the logic behind the appointment related function, the partitions for each method is outlined below, the test cases are all included within the same test method:

appointment_get_by_token (Test method: test_getByToken_test)

Equivalence Partition 1: (Valid) a valid token and valid appointment id

Equivalence Partition 2: (Invalid) token not from the concerned patient or hcp

Equivalence Partition 3: (Invalid) invalid / empty token

Equivalence Partition 4: (Invalid) no token supplied

Equivalence Partition 5: (Invalid) invalid / no appointment id

appointment_get_calendar (Test method: test_getCalendar_test)

Equivalence Partition 1: (Valid) a valid hcp token

Equivalence Partition 2: (Valid) a valid patient token

Equivalence Partition 3: (Invalid) token not from by patient or hcp

Equivalence Partition 4: (Invalid) invalid / empty token

Equivalence Partition 5: (Invalid) no token supplied

delete_appointment (Test method: test_delete_appointment_test)

Equivalence Partition 1: (Valid) a valid hcp token

Equivalence Partition 2: (Valid) a valid patient token

Equivalence Partition 3: (Invalid) token not from by patient or hcp

Equivalence Partition 4: (Invalid) invalid / empty token

Equivalence Partition 5: (Invalid) no token supplied

create_appointment (Test method: test_createAppointment_test)

Equivalence Partition 1: (Valid) a valid hcp token and valid appointment details

Equivalence Partition 2: (Valid) a valid patient token and valid appointment details

Equivalence Partition 3: (Invalid) invalid hcp / patient token

Equivalence Partition 4: (Invalid) invalid / empty token

video (Test method: test_video)

Equivalence Partition 1: (Valid) a valid hcp token and valid appointment id

Equivalence Partition 2: (Valid) a valid patient token and valid appointment id

Equivalence Partition 3: (Invalid) invalid hcp / patient token

Equivalence Partition 4: (Invalid) invalid / empty token

Equivalence Partition 5: (Invalid) Broken Twilio API

upweb-api/src/patient/patient.py

This package contains all the endpoints for the frontend to interact with, some logic will be applied to the input data and then transmitted to its corresponding helper function.

login (Test method: test_login)

Equivalence Partition 1: (Valid) a valid patient token and valid email

Equivalence Partition 2: (Invalid) invalid / empty token

signup (Test method: test_signup)

Equivalence Partition 1: (Valid) a valid patient signup data

Equivalence Partition 2: (Invalid) no / defective signup data

remove (Test method: test_remove)

The endpoint is for testing purposes only not used in production

Equivalence Partition 1: (Valid) a valid patient token

getbytoken (Test method: test_getbytoken)

Equivalence Partition 1: (Valid) a valid token and valid appointment id

Equivalence Partition 2: (Invalid) invalid / empty token

Equivalence Partition 3: (Invalid) no token supplied

get_records (Test method: test_get_records)

Equivalence Partition 1: (Valid) a valid token and valid appointment id

Equivalence Partition 2: (Invalid) empty token

edit_profile (Test method: test_edit_profile)

Equivalence Partition 1: (Valid) a valid token

Equivalence Partition 2: (Invalid) empty token

gethcps (Test method: test_gethcps)

Equivalence Partition 1: (Valid) a valid token

Equivalence Partition 2: (Invalid) empty token

get_all (Test method: test_get_all)

Equivalence Partition 1: (Valid) a valid token

Equivalence Partition 2: (Invalid) empty token

set_profile_picture (Test method: test_set_profile_picture)

Equivalence Partition 1: (Valid) a valid token

Equivalence Partition 2: (Invalid) empty token

search (Test method: test_search)

Equivalence Partition 1: (Valid) a valid token and search text length >= 3

Equivalence Partition 2: (Invalid) empty token

Equivalence Partition 2: (Invalid) search text length < 3

Boundary Condition 1: (valid) len(text) == 3

Boundary Condition 2: (Invalid) len(text) == 2

upweb-api/src/patient/patient_helper.py

This package contains the logic behind the patient related function, the partitions for each method is outlined below, the test cases are all included within the same test method:

pat_login (Test method: test_login)

Equivalence Partition 1: (Valid) a valid email address

Equivalence Partition 2: (Invalid) invalid email address

pat_signup (Test method: test_pat_signup)

Equivalence Partition 1: (Valid) a valid patient details

Equivalence Partition 2: (Invalid) empty patient details

pat_get_by_token (Test method: test_getByToken)

Equivalence Partition 1: (Valid) a valid patient token

Equivalence Partition 2: (Invalid) invalid patient token

pat_delete (Test method: test_delete)

Equivalence Partition 1: (Valid) a valid patient token

Equivalence Partition 2: (Invalid) invalid patient token

pat_edit_profile (Test method: test_edit_profile)

Equivalence Partition 1: (Valid) a valid patient details

Equivalence Partition 2: (Invalid) invalid patient details

pat_get_records (Test method: test_pat_get_records)

Equivalence Partition 1: (Valid) a valid patient details

Equivalence Partition 2: (Invalid) invalid patient details

pat_get_hcps (Test method: test_get_hcps)

Equivalence Partition 1: (Valid) a valid patient details

Equivalence Partition 2: (Invalid) invalid patient details

set_profile_picture (Test method: test_set_profile_picture)

There is no user input hence only test was done to assure the code behave as expected to extract all information from database

pat_get_hcps (Test method: test_get_hcps)

This is a simple setter in firestore with one line of code, test only done to make sure it behaves as expected

pat_search (Test method: test_pat_search)

This is a simple getting in ALGOLIA with linear logic and no branch, test only done to make sure it behaves as expected

add_pat (Test method: test_add_pat)

This is a simple adding record in ALGOLIA with linear logic, test only done to make sure it behaves as expected

upweb-api/src/hcp/hcp.py

This package contains all the endpoints for the frontend to interact with, the data will be transmitted to its corresponding helper function. The functions in hcp.py get the data from the endpoints called and here parse the data from the post request into objects, which are then sent to the helper functions. This is tested with hcp_endpoint_test.py.

upweb-api/src/hcp/hcp_helper.py

Tested with the file hcp_test.py

SignUp:

/test_signup_test

Creates mock HCP with sample HCP profile data, as it would be received from the frontend.

Equivalence Partition 1: (Valid) A valid HCP object, hours and database

Equivalence Partition 2: (Invalid) No data supplied

Login

/test_login_test

Create a mock HCP object and HCP database.

Equivalence Partition 1: (Valid) A valid email and id

Equivalence Partition 2: (Invalid) No data supplied

Equivalence Partition 3: (Invaid) Non registered ID/Email

Set_record

/test_set_record
Create a mock HCP object and Patient database, Patient Object, and health event
Equivalence Partition 1: (Valid) A token, valid health event object, Patient ID

Equivalence Partition 2: (Invalid) Invalid token

Equivalence Partition 3: (Invaid) Invalid Patient ID, HCP not in list of Patient’s

Notify Test

/test_notify_test

Mocks a test notification to a patient with an HCP Object and Appointment object.

Equivalence Partition 1: (Valid) Authorized HCP token, Appointment ID
Equivalence Partition 2: (Invalid) Non authorized HCP Token
Equivalence Partition 3: (Invalid) Appointment Id not valid, does not exist

Remove Test

Tests delete HCP from db

/test_remove_test
Equivalence Partition 1: (Valid) Valid HCP Token, HCP exists
Equivalence Partition 2: (Invlid) Invalid HCP token, HCP does not exist

Test Get By Token

Tests if HCP is returned by HCP token

/test_getByToken

Equivalence Partition 1: (Valid) Valid HCP token
Equivalence Partition 2: (Invalid) Non authorized HCP Token
Equivalence Partition 3: (Invalid) Invalid HCP token

Test number

Tests the twilio call to see if a number can be texted.

/test_test_number

Equivalence Partition 1: (Valid) Valid HCP token, Valid Appointment ID
Equivalence Partition 2: (Invalid) Non authorized HCP Token
Equivalence Partition 3: (Invalid) Invalid HCP token
Equivalence Partition 4: (Invalid) Invalid Appointment token

Edit Profile

Creates a new sample HCP profile and updates the mock HCP db.

/test_edit_profile
Equivalence Partition 1: (Valid) Valid HCP token, Valid HCP profile
Equivalence Partition 2: (Invalid) Non authorized HCP Token
Equivalence Partition 3: (Valid) Optional fields, title, specialty, profile picture are not included.

Health Event

Create a mock health event and add it to the mock patient in the mock patient db.

/test_health_event
Equivalence Partition 1: (Valid) Valid HCP token, Valid Patient ID, health event string
Equivalence Partition 2: (Invalid) Invalid HCP token
Equivalence Partition 3: (Invalid) Nonvalid Patient ID, does not exist

Get All

Mocks HCP db and request to return list of HCP objects

/test_get_all

Equivalence Partition 1: (Valid) Valid Token
Equivalence Partition 2: (Invalid) Invalid Token

Get Patients

Get mock patients for the mock HCP

/test_get_patients
Equivalence Partition 1: (Valid) Valid HCP Token
Equivalence Partition 2: (Invalid) Invalid HCP Token

Set Profile Picture

Sets the profile picture field of the mocked HCP object

/test_set_profile_picture
Equivalence Partition 1: (Valid) Valid HCP Token, ProfilePic string
Equivalence Partition 2: (Invalid) Invalid HCP Token

HCP Search

Given that this function is based off of Algolia search and being able to retrieve the objects from algolia indexing services these were also tested manually while connected to the actual algolia search service.
/test_hcp_search
Equivalence Partition 1: (Valid) Valid Token, search len(string)>= 3

Equivalence Partition 2: (Invalid) Invalid HCP Token

Equivalence Partition 3: (Invalid) search len(string)< 3

Add HCP

Adds HCPs to the initialization of the Algolia Search

This is only done when the index is initially created and when an HCP signs up. This function is called on

successful HCP signup.

Equivalence Partition 1: (Valid) HCP signup was successful

Equivalence Partition 1: (Invalid) HCP was not added to the Firestore db due to Firebase error, function not called.

Link to backend automated test suite in GitHub.

Part 3 – Branch Coverage

Frontend

Coverage testing in the frontend was achieved using a combination of Istanbul for branch coverage testing and non-React unit testing and Jest for React component unit testing.

On the non-React side, as mentioned in Part 2, the Validator tests served most crucial for ensuring the correct inputs were being sent through to the endpoints. The Client utility class was ignored for testing, as we separated our backend tests from the frontend tests. Additionally, the Storage utility class was ignored because there is no way to access the browser cache from a testing environment, and this utility class is simply a wrapper over the DOM. Finally, the Users utility class, which sets and retrieves user tokens, was ignored because of its heavy reliance on the aforementioned tests. Nonetheless, the utility classes that could be tested achieved 100% branch coverage.

Regarding coverage testing for React components, branch coverage achieved was 90.75%. All Client/User requests (anything that interacts with the Firestore database) were mocked to avoid making actual requests. However, not all API requests could be mocked (such as Twilio), which is one of the key reasons for missing coverage. Other reasons include being unable to trigger component specific events (such as onClose and onOpen) using only Jest and testing-library/react, and inability to mock certain 3rd party libraries. A more in-depth explanation of some of the issues faced follows:

- NavigationBar contains a <Link /> component that requires Router context, which cannot be mocked.
- For GoogleButton, we could not mock and trigger the onSuccess and onFailure events of Google OAuth.
- For Popup, we could not mock and trigger the onOpen and onClose events (onClose could possibly be triggered by focusing on some other part of the page, however Jest DOM renders only the component itself, meaning there is no other part of the page available to move to).
- Popup (and AppointmentPopup) caused some issues in other component tests (that contain Popup) where elements on the page could not be clicked in the testing environment despite the buttons being verified that they were not disabled.
- For Participant, Room, and Appointment, implementation relied on functions/classes imported from the twilio-video library. Due to the underlying complexity of the imports, there was no reasonable way to

mock them (e.g Room component takes in a Participant as prop, however, Participant is an import from twilio-video that is wrapped with specific properties such as audioTracks and networkQualityStats that cannot be mocked). Thus, they were ignored for coverage as actions such as a participant (user such as a doctor or patient) joining a Room were unable to be replicated in a testing environment. Testing that the component was able to be rendered was still tested regardless.

- Likewise, SuggestionInput relies on react-autosuggest, and features many non-triggerable events (from a testing environment) such as onSuggestionsFetchRequested, onSuggestionsClearRequested, renderSuggestion, etc. For this and other relevant components, the events that could be triggered (such as onChange and onClick) were tested as much as possible. In particular, some SuggestionInput events could be triggered by changing the input value to something that could be auto-completed, e.g “den” to “dentistry,” but this still was not enough to achieve satisfactory branch coverage.

Backend

For the backend, we used coverage.py to measure branch coverage. The report for the checkstyle is located in tst/reports/report-T5. The files covered in the coverage run are all python files in the upmed-api and were run as unittest. The final coverage value is 90%.

The file with the lowest coverage is the util.py file which is for connecting to Twilio, and as part of a static unittest we did not want to connect to the Twilio services. The other reason for not having 100% coverage is due to some exception statements in our endpoints and helper functions.

We added try/except block around calls to the Firebase calls, such that if firebase has an unexpected issue then pur program won’t crash. For testing purposes it is difficult to simulate and trigger examples of Google Firebase document storage failures. The main reason they were added was because the Firebase documentation recommended these calls have the try except blocks.

In addition to the difficulty of testing all firebase branches, all other tests used mocking to mock out unreachable components such as firebase api calls, twilio calls. All of them have similar difficulty to write comprehensive mocking classes to fully simulate its behaviour.

Part 4 – Continuous Integration

We have configured Firebase CI/CD to deploy our frontend whenever we merge a PR to the main branch and have, additionally, configured Heroku CI/CD to deploy our backend immediately after. Both of these tools produce reports, albeit in different formats, which can be found in our project's README. The reports include outputs from running coverage, styling, bug-fixing, and, in the case of front-end, automation testing. For convenience, the reports can be found below.

Click here to find the GitHub actions outputs produced by Firebase from each new commit on main. The CI configuration for the frontend can be found here.

Click here to find a custom API endpoint we created that displays all Heroku logs from the most recent merge. Please note that, since this is a free Heroku application, you might have to wait a few minutes after your first request for the server to boot up. The CI configuration for the backend can be found here (note, the actual CI script is in upmed-api/run.sh).