PowerShare App Development Technical Report

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Spring, 2020

Technical Project Team Members
Renat Abazov
Chris Lee
Jeremy Nathan
Richard Ohr
Andy Tan
Stephen Thiringer

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Ahmed Ibrahim, Department of Computer Science

Table of Contents

Abstract

PowerShare is a mobile application with the purpose of connecting a politician directly to their constituents on their mobile device. The vision of our client was to create a system that would help bring the focus of politics back to civil service. The state of the American political system at present is one of partisanship, and this often gets in the way of the needs of the people. Too often the voices of the common man are left unheard by those who represent them. So, our client wanted this system to enable direct communication between the two parties. In contrast to the often slow pace of bureaucracy, this interaction would be instantaneous, taking place during downtime in office.

We created a mobile application in JavaScript using the React Native framework to accomplish this task. The backend uses Google Firebase for data storage, authentication, and cloud function execution. The application possesses all core functionality needed to enable the creation of actionable goals by citizens of specific localities, and the management of those goals by elected officials. Our client was a man who had an idea and worked to bring it to fruition. Similarly, we hope that our application can make this same process easier for anyone who wants to use it.

List of Tables

List of Figures

1. Introduction

In the current two-party political system of the United States, the concerns of the common citizen are often cast aside in favor of things like party allegiance or drawn out bureaucratic processes. A direct connection to constituents could help skirt these distractions and bring a more rapid realization of promises made, while making elected officials' decisions transparent to their constituents. This is the nature of the project detailed in this paper.

1.1 Problem Statement

The client has taken issue with this style of political organization and wants to create a tool to bring the focus of American politics to the needs of the individual. He stated that he wanted the tool to be accessible, so that a politician would consider checking it during a lull whilst on the job. This accessibility is necessary from the side of the citizen as well, so that they can easily propose goals for their communities. This led to the idea of a mobile app to centralize citizen goals and facilitate communication between both parties.

Currently, common ways for the average citizen to voice their concerns is to attend town hall meetings or contact the office of their representative directly. These are important processes that sometimes produce results, and the group is in no way implying that they should cease. However, even in these activities, many are ignored and they can be seen as chores. The tool we are creating would put a name and a number (score) to every idea that residents of individual communities have, all in one place. They would live in the app, in full view of the other citizens and their representative at all times, unable to be ignored like a raised hand in a meeting, a phone call sent to voicemail, or a letter put in the trash.

1.2 Contributions

To solve this problem, we were able to build a mobile application that accomplishes the problem above. As we will describe at length in forthcoming sections, the PowerShare app centers around citizens and politicians acting within their designated community in-app. They create and vote on proposed goals in order to allow the most popular goals to filter to the top of a community-wide list. The representative of this specific community then browses this list, with the ability to provide actionable steps to completion, to leave comments on the goal, or to mark a goal as complete. The app supports email notifications for various events in order to keep both representatives and citizens informed of changes within their community.

2. Related Work

As mobile applications and frameworks have become more widespread over the last decade, various organizations have tried to apply these advances in technology to improve the United States electoral system. One example of this is the Shadow app, which was used in the 2020 presidential primary election in Iowa for the Democratic party. The app was created to shorten the amount of time required for votes to be tabulated, by removing the need to manually count ballots; instead, the votes would directly be stored in the app's backend database. However, due to a bug in the reporting system, there was a period during the day of the primaries where votes stored in the app's database were misrepresented to media outlets and Democratic leadership. This error led to a massive outcry in social media that caused a distrust of the party leadership, especially the party's decision to contract the Shadow application to implement a brand-new voting system. This failure demonstrates the need for client-focused testing prior to the deployment of an application; had a smaller subgroup of voters been selected for a trial run, this situation might have been averted.

Shadow is an application that is focused on election day technology; there are fewer applications with the sole purpose of connecting the public with their elected officials during the period between elections. One existing digital tool that aims to create a channel between elected officials and their electorate is the digital application Countable, which allows users to "get clear, concise summaries of bills going through Congress, see what others think, then take action." To accomplish this, the application is divided into two feeds: an opinion feed, which consists of opinion pieces written by users, and a bill feed, which shows a dashboard of bills recently drafted by Congress. Both feeds implement a social network format, in which users can vote and comment on elements of each feed. Though promising, this application ultimately has several

shortcomings. First, it does not provide a new channel by which representatives and constituents can communicate; constituents still would have to email or video message their representatives. Next, the social-network format of the site, which enables users to comment on each other's posts and opinions, may be irrelevant to a representative trying to find the most important goals to pursue for their community. As a result, this application does not satisfy necessary high-level requirements.

3. System Design

On the highest level, our system must enable direct communication between a representative and their constituency. This communication will take place in the form of goals proposed by citizens and response by the representative in the form of comments or subgoals. There are three types of user: “user”, “customer”, and “admin.” Every “user” is part of one or more “communities” based on their address. A “user” can create goals in any community that they are part of. They can see and search through other goals posted in their communities and vote for one goal in each community. A “customer” is the elected representative of a given community. They are able to add comments and subgoals to goals in the community of which they are the representative. They can also mark goals as “complete.” An “admin” does the tasks of managing PowerShare overall. An admin can add communities and approve or deny goals submitted by users. In addition, an admin has access to user statistics, which tells the admin how many users there are, the number of goals, the number of subgoals, the number of communities, and the number of items in the goal inbox.

We decided to write the app in React Native, a popular javascript framework for mobile development. We chose this framework primarily because it fit the requirement of a mobile application, and because it allows writing a single codebase for use on both Android and iOS operating systems. This was ideal because, although our client originally wanted an iOS application, we only had two team members with the ability to develop on iOS (Apple requires that you develop iOS apps on MacOS). This way, we developed the Android application with our team members on Windows, and the iOS application with our team members on MacOS. However, as we encountered build issues on the iOS portion of the application, our finished product is the Android version.

Additionally, our code is licensed under the MIT license. This license allows anyone to obtain a copy of our code and to "use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies" of the code, with the only restriction being that the same license and copyright notice be transferred to the new product ("The MIT License", n.d.). Our client gave us permission to license code in this way, most likely in the hope that this permissive license will help carry his vision far and wide.

3.1 System Requirements

Gathering system requirements is essential for ensuring product quality and correctness. It allows the programming team to know exactly what they are building, and that they are building the right thing. It also allows the customer to explain exactly what they expect from the product. Furthermore, each requirement is broken up into its own individual story, which allows individual members of a team to work on small parts and pieces that then go on to form a collective unified project. As the programming team does this they can keep track of their progress and make sure that they are meeting their deadlines. System requirements were gathered by interviewing the customer in-person.

Minimum Requirements

STORIES	**PTS.**
As a USER, I should be able to **submit a goal** to a community that I am part of such that any other member of the community can see it and vote on it after review.	8
As a USER, I should be able to **create a verified account** with my name, email, and physical address so that I can access the app.	8
As a USER, I should be able to **search goals** in my community so that I can find relevant goals to vote on.	5

As a USER, I should be able to **view goals** such that I can see a list of their authors, supporters, sub goals, approval status, completion status, media associated with goal.	5
As a USER, I should be able to **vote on one goal** in each community that I am part of	3
As a USER, I should be able to **receive notifications** on goals that I have voted for.	5
As a USER, I should be able to **view and edit my account settings**, so that I can manage things like login information, notifications, and other general settings.	8
As a USER, I should be able to **navigate between a home page and a community page, as well as a community page and a goal page**, with one action.	3
As a USER, I should be able to **view a dashboard** of all communities I am a member of, so that I can choose which community to view goals for.	2
As a USER, I should be able to **view contact information for both Powershare and my community representative** so that I can get in touch if needed.	2
As a USER, I should be **added to all relevant communities** after creating an account with my home address.	8
As a CUSTOMER, **I should be able to do everything a USER can**.	3
As a CUSTOMER of a specific community, I should be able to **respond to a goal with feedback**	5
As a CUSTOMER of a specific community, I should be able to **add sub-goals** to a goal	8
As a CUSTOMER, I should be able to **search goals in communities of which I am not a member**, by keyword.	5
As a CUSTOMER, I should be able to **upload media** to any goal in my community.	13
As a CUSTOMER, I should **receive notifications** (Android / iOS push notifications) for the following: a new user joins a community, a new goal is created, a goal is edited, goal ranking changes, a completion date is approaching.	5
As a CUSTOMER, I should be able to **designate a goal as complete**.	3
As a USER, I should be **automatically assigned** to my relevant communities based on address upon account creation so that I can vote on the issues relevant to my communities.	8
As a USER, I should be able to login into my account.	8

Table 1. Minimum Requirements

Desired Requirements

STORIES	**PTS.**
As an ADMIN, I should be able to **approve membership** for members into the community (*Should this be automated through checking voter registration records*?)	13
As an ADMIN, I should be able to **view user statistics**.	13
As an ADMIN, I should be able to **search through a list of communities** by geographic location.	5
As an ADMIN, I should be able to **view a dashboard** which includes the above inbox and list of communities.	8

Table 2. Desired Requirements

Optional Requirements

STORIES	**PTS.**
As a USER, I should be able to log out from inactivity after 15 minutes to increase security.	5
As a USER, I should be able to follow goals that I have neither voted for nor created.	8
As a USER, I should have the option to be sent push notifications about followed goals.	3
As a USER, I should have the option to be sent notifications by email and/or SMS.	5
As a CUSTOMER, I should have the option to tag/label goals by category/topic and search for them by the label.	8
As a USER, I should be able to sign into the app with my fingerprint/faceID.	8

Table 3. Optional Requirements

3.2 Wireframes

Through the use of wireframes, we were able to produce a prototype frontend for the customer. By agreeing on the design of the different screens of the application, as well as the navigational flow for users of different types and different levels of authorization, we were able to quickly implement a front-end for our client that simulated the desired behavior of the application.

Below on the left are some of the wireframes that we had developed at the start of the project, along with their corresponding final version on the right.

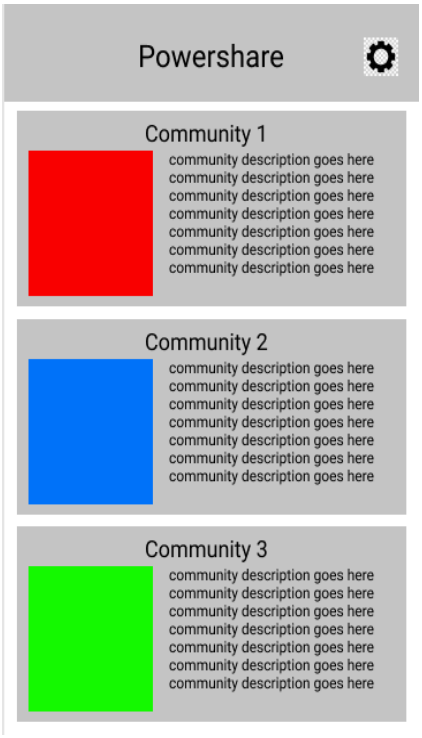

Fig. 1: Home Screen Wireframe

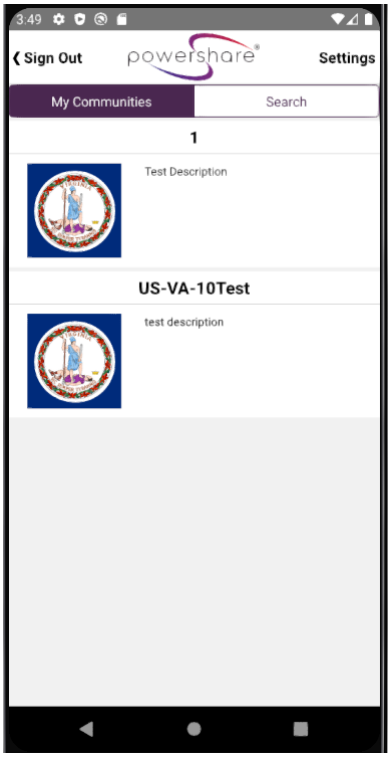

Fig. 2: Final Home Screen

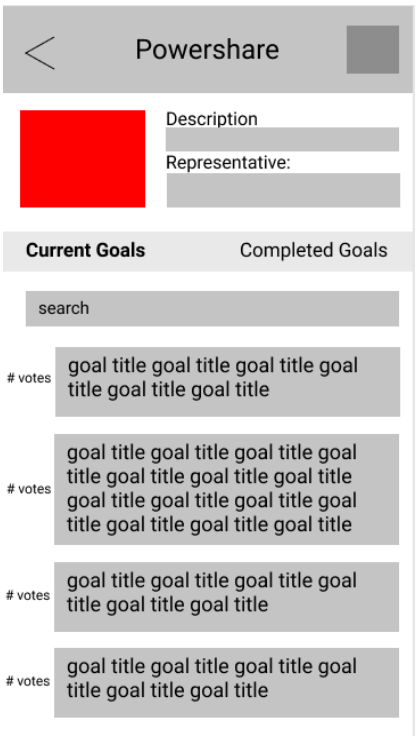

Fig 3: Community Screen Wireframe

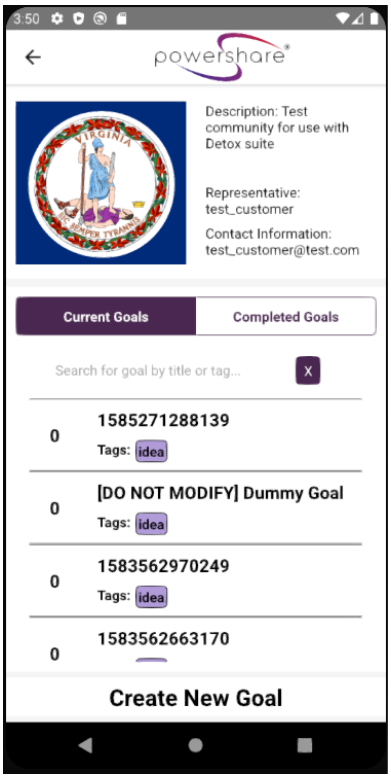

Fig 4: Final Community Screen

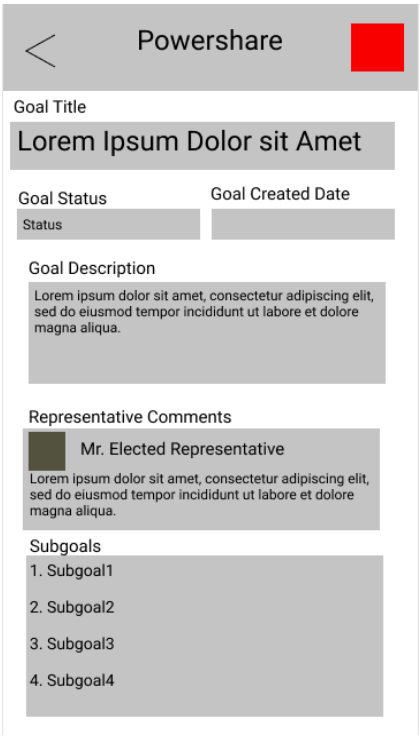

Fig 5: Goal Screen Wireframe

Fig 6: Final Goal Screen

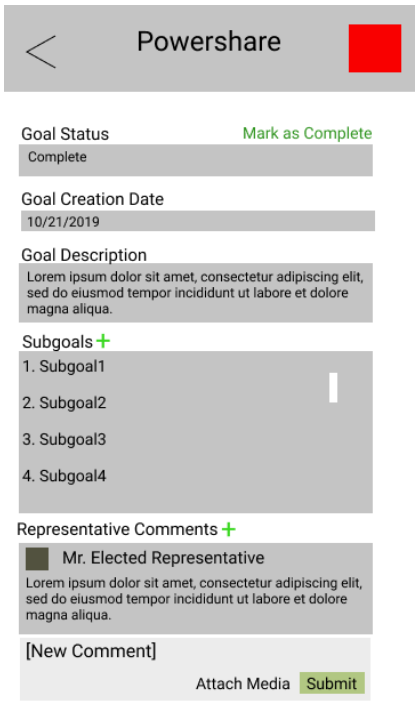

Fig 7: Create Goal Screen Wireframe

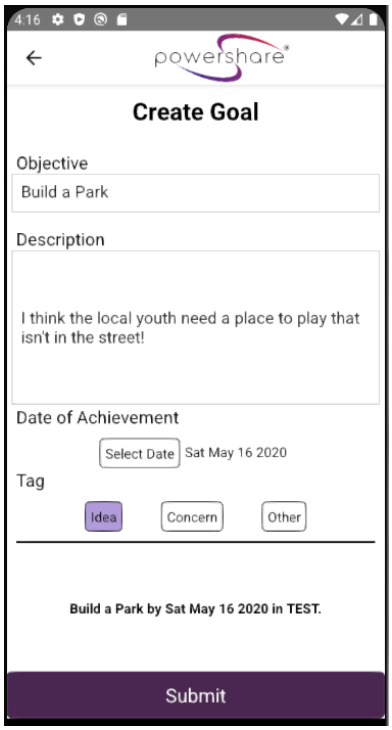

Fig 8: Final Create Goal Screen

3.3 Sample Code

This sample is one of our Firebase functions, which automatically triggers when a certain spot in the database is changed.

```
//Email sent when user's goal is marked as complete by their representative
exports.sendGoalCompleteEmail = functions.database.ref('/goals/{goalID}/isComplete')
    //this function is triggered when the "isComplete" section of one of the user's goals is updated
    .onUpdate((change, context) => {
        const goalID = context.params.goalID;
        let goalData = {};
        let creatorID, targetEmail = '';
        let permissionsEnabled = false;

        // reference the updated goal to get the creator's ID
        return admin.database()
            .ref('goals/' + goalID)
            .once('value', (snapshot) => {
                goalData = snapshot.val();
                creatorID = snapshot.child('creator').val();
        }).then(() => {
            // check the settings of the goal's creator and get their email
            return admin.database()
                .ref('users/' + creatorID)
                .once('value', (settingsSnapshot) => {
                    permissionsEnabled = settingsSnapshot.child('settings/goalStatus').val();
                    targetEmail = settingsSnapshot.child('email').val();
            })
        }).then(() => {
            /* if the user has the setting enabled to receive updates on their created goals,
             construct the email and send it */
            if(!permissionsEnabled){return null;}
            const body = "This email is to notify you that the following goal you created has"+
                         "been marked as complete:\n\n" +
                          JSON.stringify(goalData, null, '\t') + "\n\n" +
                          "Thank you for using PowerShare!";
            return sendEmail(DEBUG_ENABLED ? DEBUG_EMAIL : targetEmail,
            "Your Goal has been Marked as Complete!", body)
        });
})
```

Fig 9: Goal Completion Listener

This is the “render” function for the Home screen. This is what the Home screen displays.

```
render() {
  return (
    <View testID="home-view" style={styles.parentContainer}>
      {/* this if statements displayings loading until the done flag is
      changed to true, it then displays the content */}
      {!this.state.done ? (
        <View
          style={{
            justifyContent: 'center',
            alignItems: 'center',
            marginTop: '80%',
          }}>
          <ActivityIndicator size="large" color="grey" />
        </View>
      ) : (
        <View testID="home-view" style={styles.parentContainer}>
          {/* these functions render all of the page's content */}
          {this.renderCommunityView()}
          {this.renderCommunitySearchView()}
        </View>
      )}
    </View>
  );
}
```

Fig 10: Home Screen render Function

This is the “renderCommunityView” function, which is called in the render function

```
renderCommunityView() {
  if (this.state.otherCommunityList.length == 0) {
    return (
      <View style={styles.communityContainer}>
        {/*
        Home Screen
          FlatList of community cards loaded in from database.
          Each card contains the community picture (placeholder for now),
          the community title, and description. Pressing a card loads the community
          screen for that community, and passes the community ID along for future use.
        */}
        <FlatList
          data={this.state.myCommunityList}
          renderItem={this.renderCommunityList}
        />
      </View>
    );
  }
}
```

Fig 11: Community Screen renderCommunityView Function

This is the “renderCommunityList” function, which is called in the above function to load each list element.

```
renderCommunityList = ({item}) => (
  <TouchableOpacity
    testID={item.key}
    style={styles.button}
    onPress={() => {
      this.props.navigation.navigate('Community', {
        // parameters to pass on to the community screen
        communityID: item.key,
        communityPic: item.imageurl,
        /*pass on whether the current user is a member of the community
          that they are navigating to.  This will prevent them from having access to
          features, such as goal creation, that they shouldn't have access to.
          */
        member: this.state.myCommunityList
          .map(community => community.key)
          .includes(item.key),
      });
    }}>
    <View style={styles.titleContainer}>
      <Text style={styles.communityTitle}>{item.key}</Text>
    </View>
    <View style={styles.contentContainer}>
      {/* load the image from this community's imageurl*/}
      <Image
        testID={'comIcon' + item.key}
        source={{uri: item.imageurl}}
        style={styles.image}
      />
      {/* display community description*/}
      <Text style={styles.fieldcontentsmall}>{item.data.description}</Text>
    </View>
  </TouchableOpacity>
);
```

Fig 12: Community Screen renderCommunityList Function

3.4 Sample Tests

Testing ensures product quality and consistency and allows for easier maintenance of the codebase. Testing especially improves the speed at which bugs are discovered and reported. With less time and effort spent maintaining the codebase, the overall maintenance costs decrease. The client can also validate the progress on the project, by looking at the test cases relevant to each system requirement.

This is the “login” function from goal_functions.spec.js. We call this function to have the emulator perform all necessary actions to log in to the app.

```
/* calling this function will log you in as test_<username>@test.com */
var login = async username => {
    await expect(element(by.id('username-input'))).toBeVisible();
    /* You can do actions on elements (https://github.com/wix/Detox/blob/master/docs/APIRef.ActionsOnElement.md)*/

    /* Type in username and password, then tap the login button */
    await element(by.id('username-input')).replaceText(
      'test_' + username + '@test.com',
    await element(by.id('password-input')).replaceText('password');

    await waitFor(element(by.id('login-button')))
      .toBeVisible()
      .withTimeout(5000);
    await element(by.id('login-button')).tap();
};
```

Fig 13: Test Case Setup and Login Function

This test simply tries to log in with a valid user account. It checks to see that the home screen appeared after entering credentials and tapping “log in”.

```
it('should be able to log in with a valid account and see home screen', async () => {
  /* Wait for emulator to reload (you likely should do this at the beginning of every test) */
  await login('user');
  /* Need to waitFor again to give it time to log in */
  await waitFor(element(by.id('TEST')))
    .toBeVisible()
    .withTimeout(10000);
});
```

Fig 14: Login Function Test Case

This test checks that a user can access the “edit” function of a goal that they created. It logs in to a user, then navigates to a goal that user created. Then it checks to see if the edit button exists.

```
it('should be able to edit goal user did create', async () => {
  //Steve sprint 5 test 3
  await device.launchApp({delete: true});

  await login('customer');
  /* move to community screen */
  await waitFor(element(by.id('TEST')))
    .toBeVisible()
    .withTimeout(10000);

  await element(by.id('TEST')).tap();

  /* wait for the community screen to load in */
  await waitFor(element(by.id('communityScreen')))
    .toBeVisible()
    .withTimeout(10000);

  await waitFor(element(by.id('incompleteList')))
    .toBeVisible()
    .withTimeout(10000);

  await waitFor(element(by.id('-M1nL_p8ZPc6_0y0NSDg:goal')))
    .toBeVisible()
    .withTimeout(10000);

  /* go to goal screen */
  element(by.id('-M1nL_p8ZPc6_0y0NSDg:goal')).tap();

  /* make sure that the edit button is visible,
      since we are the creator of this goal*/
  await waitFor(element(by.id('goalTitle')))
    .toBeVisible()
    .withTimeout(10000);

  await waitFor(element(by.id('edit')))
    .toBeVisible()
    .withTimeout(10000);
});
```

Fig 15: Edit Goal Function Test Case

This test checks that a user can access the “delete” function of a goal they created. It logs in to a user, then navigates to a goal that user created. Then it checks to see if the delete button exists.

```
it('should be able to delete goal user did create', async () => {
  //Steve sprint 5 test 4
  /* log in */
  await login('customer');
  /* move to community screen */
  await waitFor(element(by.id('TEST')))
    .toBeVisible()
    .withTimeout(10000);

  await element(by.id('TEST')).tap();

  /* wait for community screen to load */
  await waitFor(element(by.id('communityScreen')))
    .toBeVisible()
    .withTimeout(10000);

  await waitFor(element(by.id('incompleteList')))
    .toBeVisible()
    .withTimeout(10000);

  await waitFor(element(by.id('-M1nL_p8ZPc6_0y0NSDg:goal')))
    .toBeVisible()
    .withTimeout(10000);

  /* move to goal screen */
  element(by.id('-M1nL_p8ZPc6_0y0NSDg:goal')).tap();

  await waitFor(element(by.id('goalTitle')))
    .toBeVisible()
    .withTimeout(10000);

  /* make sure the delete button is visible */
  await waitFor(element(by.id('delete')))
    .toBeVisible()
    .withTimeout(10000);
});
```

Fig 16: Delete Goal Function Test Case

3.5 Code Coverage

Our group is unable to obtain code coverage data due to the nature of our project. The default test runner for React is Jest. Jest can track code coverage, but we are not able to run Jest on its own on our project. This is due to the various React Native packages that we had to use. When Jest runs our app, it cannot import these packages. Therefore, it essentially cannot run any page of our app.

For unit testing, we instead used a package called Detox (https://github.com/wix/detox). Instead of running code directly, Detox runs an emulator and interacts with the front-end only. This worked for our own testing. However, this package cannot track code coverage, since it can only "see" the code as a black box.

3.6 Installation Instructions

The following instructions have been written assuming that the person following them only has access to the project's Github repository.

Note: the React Native project itself is not located in the repo root; it is located in PowerShare/src/PowerShare. All React Native commands must be run from there.

3.6.1 Running the app on an emulator

Note: You can skip this section if you are just deploying the app and don't have any reason to test it on an emulator first.

Using the already-generated .apk file (recommended)

To test the app on an Android emulator, first follow this guide to install Android Studio and run an emulator:

https://developer.android.com/studio/run/emulator

Then take the .apk file from the Github repository and drag it on top of the emulator's screen. It should install the app on the emulator. You can then launch the app on the emulator.

Using React Native

If you want to test run the app on an emulator on your computer, follow this guide:

https://facebook.github.io/react-native/docs/getting-started

This will guide you through the following steps (on a Windows pc):

1. Installing Node.js and a JDK

2. Installing Android Studio and all the required components

3. Setting the ANDROID_HOME environment variable

4. Creating and starting an Android emulator through Android Studio (skip the "Creating a New Application" section)

5. Running "react-native run-android" in the "PowerShare/src/PowerShare" directory to run the app*

*Note: you might need to run "npm install" or "npm install yarn && yarn" to install all of the necessary packages that the application uses

3.6.2 Connecting a Firebase project to the app

Note: You can skip this section if you are using the Firebase database that is currently connected to the app.

Creating the Firebase project

Go to the firebase console here: https://console.firebase.google.com/

Click "add project" and follow the steps to create a new Firebase project.

Initializing the database

In the firebase console, go to the "authentication" tab and enable "email" as an authentication method. Next, go to "database", scroll down to "realtime database", click "create database", and finally "start in test mode" (so the app will be able to read and write from the database).

On the database screen, click on the three dots in the upper right and go to "import JSON".

Select the "powershare-initial-database.json" file from the root folder in the project (the repo

root). The database should populate with many Virginia communities and one admin account. The credentials for this account are as follows: the email is "powershare.dev@gmail.com" and the password "react native error."

Adding credentials from Firebase project to the app

https://invertase.io/oss/react-native-firebase/quick-start/existing-project

Follow the steps in this link to connect the Firebase project to both the Android and iOS versions of the app.

For Android, you will click the "add Android App" button in the Firebase console. It will ask for the package name of the app: the package name is "com.powershare". Click "next" or "register app" to download the credential file (google-services.json) from the Firebase console, at which point you can add it to the project (delete the previous google-services.json file). Note: you shouldn't need to make the other changes that Firebase instructs you to do, such as editing the build.gradle file.

For iOS, this will also involve downloading the credential file (GoogleService-Info.plist), adding it to the project, and initializing the Firebase service in the iOS files of the app.

3.6.3 Deploying Firebase functions for the project

Note: if you are not creating a new Firebase database, then you can skip this section

https://firebase.google.com/docs/functions/get-started

Navigate to the `Firebase/firebase_functions/functions/` directory after cloning the github repo. Install firebase-cli tools using `npm install -g firebase-tools`. Login with firebase credentials using `firebase login`. Select the project to deploy the functions to using `firebase use`. Deploy the functions to the Firebase project using `firebase deploy`.

3.6.4 Uploading the app to the Google Play Store

Signing the app and packaging the apk

Note: if no changes have been made to the app, this whole subsection can be skipped: simply use the .apk file in the Github repo located in the "release files" folder.

Follow the instructions at this link:

https://facebook.github.io/react-native/docs/signed-apk-android

This will walk you through the following steps:

- Creating a signing key and adding it to the app (note: you must have a Java installation in order to do this)
- Configuring the gradle properties to use the signing key
- Packaging the app into an aab or apk file, which can be uploaded to the Play Store

Uploading the app to the Play Store

Follow the instructions at this link:

https://support.google.com/googleplay/android-developer/answer/113469?hl=en

You can use the following gmail account as the developer account:

Email: powershare.dev@gmail.com

Password: react native error

This will allow you to access the Google Play Console (https://play.google.com/apps/publish/), where you can follow the rest of the instructions to create the app and upload your aab or apk file.

Uploading the app to the Apple App Store

Note: Our team does not have access to an Apple Developer Account at the moment. We could not follow these steps ourselves because of this. Here are the instructions once a developer account is acquired:

https://clearbridgemobile.com/how-to-submit-an-app-to-the-app-store-updated/

The app must be submitted for review before it can be released on the app store.

https://developer.apple.com/app-store/review/

Be sure to have the firebase project running in production mode before submitting it.

Include member and customer dummy accounts for testing.

Once the app has passed the review phase, you can manually release the app through the Apple app store connect page: https://appstoreconnect.apple.com/

After logging in, click My Apps, then select the app.

In the left column, select the platform version that is Pending Developer Release.

In the upper-right corner, click Release This Version/Make App Release.

4. Results

Our system provides a solution for most problems identified by our customer. The original problem was that there was no previously efficient way for users to directly share goals with elected officials within communities. However, with Powershare, users can now make goals and vote on them for elected officials to see. In addition to this, people are separated into their respective communities, so only people within a certain community can make and vote on goals that matter to them. Something we were not able to solve was that our customer wanted to use voter records to verify the users. However, we were not able to achieve this as the records for the voters were not accessible for us to use at this time. Our client talked to various local government representatives and could not get access to these records.

The result of this new system is that users can now easily make and vote on goals for elected officials to see within their community. Based on this feed of goals, elected officials can now gain a general sentiment of issues that members of that particular community find important. With this information given through the digestible format of goals, elected officials can act on these decisions and focus on what members of the community find important rather than the elected official guessing what members of the community would want.

The main customer for this application is the elected official. The elected official would use this application through viewing what the users have posted in terms of the goals and subgoals. The elected official can view the goal, the number of votes, and who voted for the goals. In addition, the customer can make follow up subgoals for the goal. After viewing these goals, the elected official can work on the goal and update users of the app on progress of the goal. The elected officials can post media such as pictures on progress being taken on the goal. If the goal were to be completed, the elected official can then mark the goal as completed, which

would make the goal viewable in the completed tab, and then work on making progress in other goals that users find important. Before, elected officials had to go through political consultants and hold hearings to see what community members wanted. This could take hours of valuable time away from the elected officials. However, PowerShare enables elected officials to see what users in their community want to get done in only a few minutes.

The other primary stakeholder that would use the system would be the voters in that district. The voters would actually make the goals and vote on the goals. Like the customer, voters can view other people's goals and see who voted on the goals. PowerShare enables voters to express themselves and communicate to elected officials what issues are important to them. Before PowerShare, voters had to rely on getting their voices heard through other means such as social media. However, an elected official is unlikely to see an individual's request for a goal in a flooded social media inbox. However, PowerShare allows a popular goal idea to be viewed by an elected official within seconds.

5. Conclusions

We currently live in the information age, with constant streams of data, knowledge, and communication available at our fingertips. Unfortunately this large amount of information can overload the user, allowing little valuable knowledge to spread between communities. The PowerShare app cuts through the whitenoise allowing direct communication between constituents and their representatives. The goal being the creation of a sense of community and connectedness.

In the creation of this app we have learned that every decision has its tradeoffs. To increase accessibility we can allow anyone to join the app with no verification of their identity, but this allows outside agents access to communities they may not be a part of in an attempt to unfairly influence decisions that should be made by the people actually living in that community. On the other hand if we focus on security, we can reach the point where the barrier of entry to the app is too great, limiting the actual usefulness of the app itself. In striking a fair balance between the two extremes the PowerShare app works to create an environment where communities can share and follow issues that actually affect them in a simple and intuitive way.

Creating an app that brings together communities and their representatives is a process that will continue as the app is used. The design decisions made along the way will of course need to be tweaked as new uses for the app are discovered by its users. Going from a list of requirements to diagrams to a fully functional app takes a great deal of work and planning to create the final product. This will all be worth it as the application attempts to solve a key communication issue in our modern world.

6. Future Work

Powershare can be further developed by improving capacity and performance for scaling the service and by expanding its accessibility. The primary way to expand the platform's capacity would be to upgrade the service plan for Google's Firebase platform, which would increase the concurrent connections limit. As the application usage grows, retrieving lists of items from the database may increase screen load times. Database request times can be monitored to see which areas of the app are performing poorly as usage scales. To keep screen loading times short, the functions that retrieve lists of items can later be changed to asynchronously request items from the database as necessary.

User accessibility can be improved by integrating push notifications. Upgrading the Firebase service plan would increase the messaging limits for Firebase Cloud Messaging, which would allow for integrating push notifications into Powershare. Push notifications would also improve user engagement. Another way to increase accessibility in the application would be to develop a Powershare web application hosted on a web server. A Powershare service available for web browsers would increase accessibility for users, customers, and administrators. A web application can be set up to communicate with the existing database on Firebase.

User verification is essential for protecting users against impersonation, and may be further augmented by verifying user accounts with each states' list of registered voters. However, these lists of registered voters may not be immediately accessible. Many state government websites provide the information in different formats, and some require paid access and/or other organization status requirements to access the data. A future implementation of this feature would need to obtain access to each state's platform and then request and handle the different data formats. We were unable to obtain access to the Virginia voter registration list during the

project because of these barriers. In the future, Powershare may be able to reduce these barriers by filing as a non-profit organization.

7. References

The MIT License. (n.d.). Retrieved from https://opensource.org/licenses/MIT