WeatherWear: Context-Aware Clothing Recommendations

FINAL REPORT

Team 34

Client/Advisers
Goce Trajcevski

Team Members/Roles
Ethan Wieczorek - Lead Back-end Developer
Nickolaus Eaton - Product Manager
Will Parr - Lead Front-end Developer
Nicus Hicks - Head of Report Development
Christian Ehlen - UI/UX and Quality Assurance Developer
Tyler Witte - Lead Software Architect

Team Email
sdmay19-34@iastate.edu

Team Website
https://sdmay19-34.sd.ece.iastate.edu/

Table of Contents

0. Executive Summary

This document aims to provide a complete summarization of the motivations, designs, implementation, and reviewal of the senior-design project WeatherWear.

The WeatherWear mobile app, for iOS and Android, leverages React Native, the DarkSky API, and an Azure SQL Database alongside a clothing-recommendation system to give users outfit ideas for the current day or packing lists for upcoming trips.

1. Requirements specification

1.1 Functional requirements

The functional requirements of this project include the ability for the application to communicate with an external weather API. The application is written in JavaScript using the React Native framework to compile on both iOS and Android. Once the data is retrieved from DarkSky, the back-end is capable of providing clothing suggestions to the user upon request for the current day and for planning a trip. These recommendations are provided from the database of user-specific clothing, and reflects specified preference settings. The user is able to add and remove clothing at will, and can edit clothing they have previously added to their digital closet.

- Login and logout
- View digital closet of available clothing
- Receive an outfit recommendation for the current day
- Receive an outfit recommendation for an array of days in a given location
- Add/Remove clothing from wardrobe

1.2 Non-functional requirements

- Scalability - The database of the application must be scalable to ensure many users will be able to access the application and their wardrobes.
- Availability - The application needs to be available 24/7 outside of maintenance for when the users require context updates for their wardrobe.
- Reliability - Application must be able to recover loss of database and preserve user accounts and wardrobes.
- Maintainability - Database must be maintained to ensure proper updates via the weather and calendar.
- Security - Database must be secure to protect information about the users and their clothing.

- Data Integrity - The clothing items stored in the database should only be able to be modified by the users or an administrator.
- Usability - Will be accessible by all users and administrators, so all users can receive updates for clothing recommendations. Users will connect to the application via a mobile device and recommended clothing will be displayed as the weather and trip plans are made.
- Performance - The application should generate a recommendation based off the weather and calendar events in less than 3 seconds.

2. System Design & Development

2.1 Design plan

After discussion and research, the team decided to use React Native for our team's implementation of this project. The team came to the decision on this framework because it allows us to develop for both Android and iOS simultaneously. For the team's external APIs, the team chose DarkSky for weather data because it is well supported and has libraries that streamlines its implementation and use within React Native. For the project database, the project uses a SQL database that is hosted on a Microsoft Azure SQL Server. Firebase also offers Google Authentication, so instead of the developer team handling authentication and encryption, the team can use the Google authentication service and have the users sign into the app using their own Google account. The team took an Agile development approach to completing this problem by assigning user stories to each member of our team and completing the entire project in sprints.

2.2 Design Objectives, System Constraints, Design Trade-offs

Assumptions:

- System will run on a single-user basis, so should be able to run on many concurrent systems.
- That the user will be able to have internet connection a majority of their time, and always when they are getting ready for the day.

Limitations:

- The system will need to be connected to the internet in order function.
- Users will need a smartphone or tablet in order to run the end product.

2.3 Architectural Diagram, Design Block Diagram

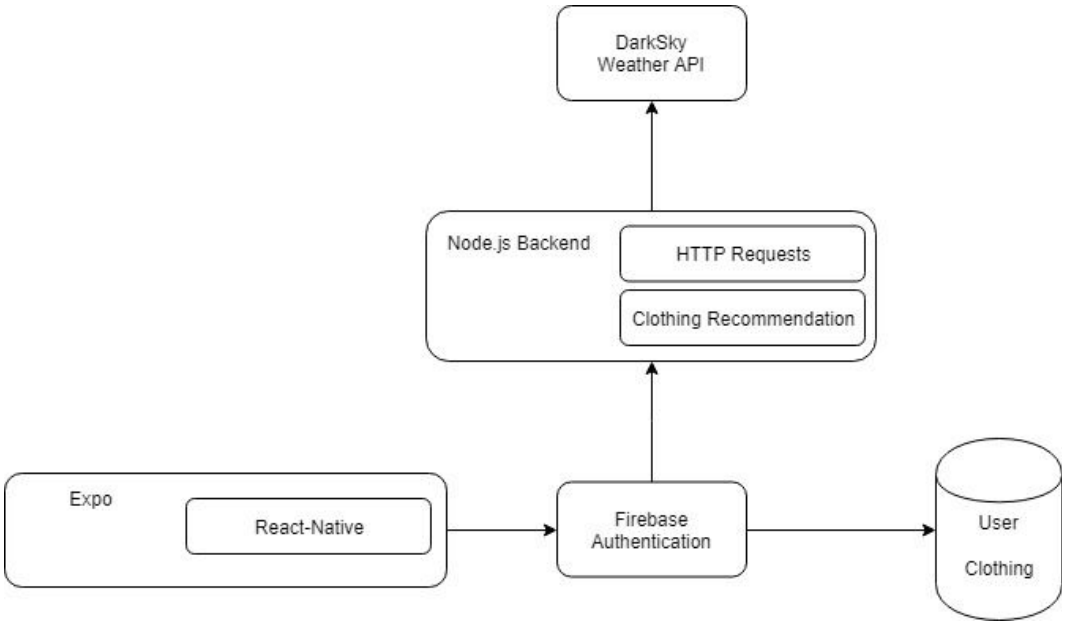


Figure 1: Architectural Diagram

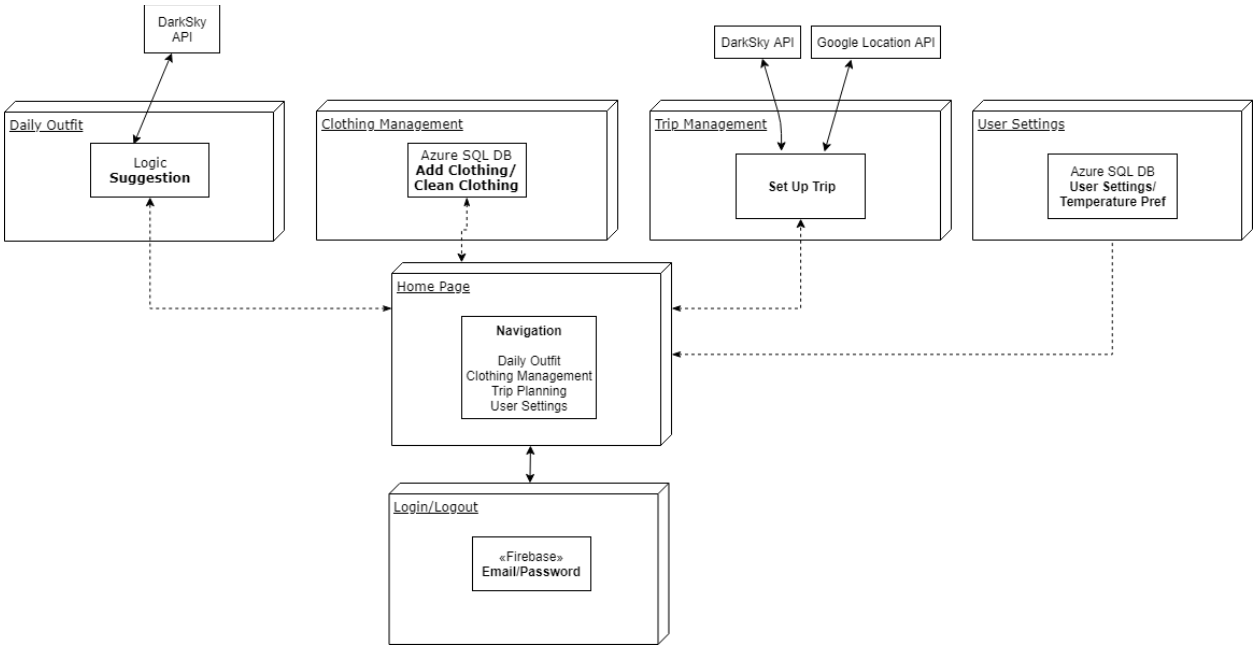


Figure 2: Design Block Diagram

2.4 Description of Modules, Constraints, and Interfaces

Daily Outfit

Modules:

- List of clothing items returned
- Weather icon/precipitation warning
- Navigation bar
- Clothing item modal

Constraints:

- When the user enters the page they will be presented with a loading screen
- When the loading screen disappears, there will be a list of clothing items that match the weather of a given day
- The weather icon/precipitation warning should display a different icon based on the precipitation chance and precipitation type (rain/snow/sleet) of the current day

Interfaces:

- Upon clicking on a clothing item, a modal will open with all of the details of that clothing item
- The navigation bar will contain clickable buttons to navigate to home, outfit, closet, and profile

Clothing Management

Modules:

- List of user clothing items
- Clothing item update modal
- Navigation bar

Constraints:

- When the users enters the page, a list of all clothing items in their closet is shown.
- When the user selects a clothing item, the details of the item are brought up in the item modal
- When the user changes and updates the information for a clothing item, the item should be updated on the back-end
- When the user updates or deletes a clothing item the list of clothing list should update

Interfaces:

- The navigation bar will contain clickable buttons to navigate to home, outfit, closet, and profile
- Clicking on a clothing item will cause its details to appear in a modal
- Inside the modal:
    - Clicking the update button will update the clothing item in the database to what is currently in the editable fields

    - Clicking the delete button will delete the clothing item on the database and update the clothing list
    - Clicking the close button will hide the modal

Trip Management

Modules:

- Date selection boxes for start date and end date of trip
- Trip dress code selector
- Trip location selector
- Button to request trip recommendations
- List of days for the trip they requested populated with arrays of clothing for each requested day's weather and dress code
- Home button on the bottom of the screen to return to the home screen

Constraints:

- The user should not be able to click the button to request a trip recommendation if they have not entered a valid location
- If the location the user enters does not come back as a valid location the app will not request a trip recommendation and will request the user enters a new location
- The trip recommendation should come back as a list of recommendations organized by day
- The trip recommendations for each day should only include clothes that match that day's dress code and weather
- The default selected dress code should be casual
- The default selected trip start date and end date should be the current date
- The maximum amount of time the user can request a trip for will be up to 60 days after the current day
- If the user goes back to the home screen and then re-enters the trip recommendation they should have to start a new trip recommendation.
    - The component should unmount upon the user leaving the page

Interfaces:

- Clicking the button to request a trip recommendation will move the user to a loading screen followed by their trip recommendation
- Clicking any of the clothing items on the page will open the modal
- Clicking the home button will redirect the user back to the home page

User Settings

Modules:

- A number entry field for the user specific temperature for each clothing category
- A button to update the user settings against the database

- Navigation bar

Constraints:

- When the user loads the page, each number entry field should be pre-filled with their current temperature settings in fahrenheit
- When the user presses the button to update their settings it should send a request to our api.
    - It should not send a request if any of the fields are empty or have non-numeric characters in them
- The user should not be able to enter any characters other than 0-9 in the number entry fields

Interfaces:

- The navigation bar will contain clickable buttons to navigate to home, outfit, closet, and profile
- Pressing the update button will send the api request to update the user's temperature settings

Home Page

Modules:

- Weather Widget displaying current high, low, temperature, and forecast for the users location.
- Button to open the trip management module
- Navigation bar

Constraints:

- The front page should only load once the user has been authenticated against Google and our database

Interfaces:

- The navigation bar will contain clickable buttons to navigate to home, outfit, closet, and profile
- Clicking the trip planner button will open the trip management module

Login Page

Modules:

- Button to load the Google Login modal

Constraints:

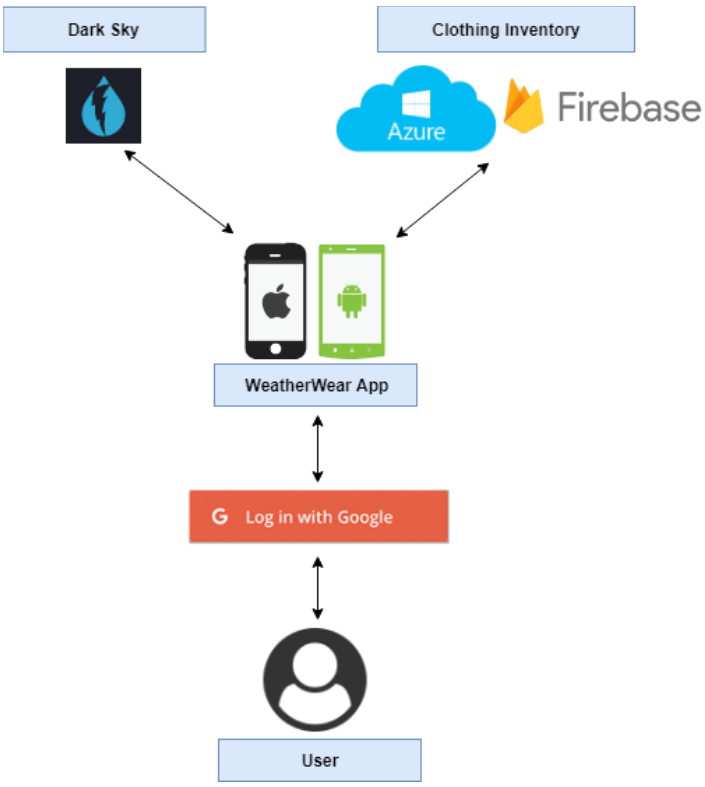
- When the login button is pressed the user will be redirected to an external Google Login Page
- When the user has entered their credentials, the page will redirect to a loading screen
- The navigation bar should not display on the login page
- The user should not be able to leave the login page if they are not logged in

Interfaces:

- The login button will redirect to an external Google Login Screen
- Correctly entering credentials will bring user back to loading screen from Google

3. Implementation

3.1 Implementation Diagram, Technologies, Software Used



3.2 Rationale for Technology/Software Choices

The technologies used in the project include:

DarkSky

The DarkSky API was used to retrieve weather data for daily clothing recommendation and trip planning. DarkSky was selected for its pricing and ease of use. It was a strongly recommended API from mobile and Javascript developers, and had been widely used so resources were readily available. It also contained all of the calls necessary to complete both the current day recommendations and planning a multiple day trip at given locations.

Azure Cloud Services

Azure Cloud Services is used to host the Node.js back-end server as well as the SQL database for the project. The decision to use Azure to host the server and database was

based on Azure's all-in-one resource management system, and in browser SQL Query Editor.

Google Firebase

User image storage and upload was managed through Google Firebase. Firebase was used for its Google Authentication support and file retrieval system.

Google Authentication

Google Authentication is used for user authentication. This decision was made because of the widespread use of Google in other applications and its cross application authentication availability. Google also handles all of the encryption of the authentication service and allows the development team to focus on the other acceptance criteria of the project.

React Native

The frontend was written in Javascript using the React Native framework. It was selected because of the ease of cross-platform development for Android and iOS, as well as the ability to live test on physical devices with the Expo framework.

Express.js

The back-end server was written using the Express framework for Node.js. Express was selected to be used so that the entire project would be in one language, and based off available documentation and community support.

Axios

Axios is a JavaScript library that can be used to perform HTTP requests that is compatible with both the Browser and Node.js. This was used so that the team could standardize the format of API requests on both the frontend and back-end. The main reason that Axios was the choice for the team was due to community support and that it is promise based and allows for both synchronous and asynchronous calls with callbacks.

3.3 Applicable Standards and Best Practices

- Testing using Enzyme, Expo, Mocha and Chai for Express/Node.js unit testing
- Git Monorepo with Dev branch
- Team members take ownership of unique positions and user stories

- Peer reviews by at least two team members for merge requests into the dev branch and release environment

3.4 Software Engineering Standards

- Agile Development Methodology
  - Efficient task scheduling per team member
  - Sprints performed weekly, with each task(s) having acceptance criteria assigned by a project manager
- SQL Database Usage
  - Properly formatted relational table and query structure
- Version Control Standards
  - Used Git branches for each task and merge requests for branches upon completion of the task
- Reviewal of all code merges
  - Performed merge requests from user story branches to the dev branch every time a task was completed
  - required at least two other team members to review code changes before merging code into the dev branch
- Coding standards
  - Modular components
  - Variable naming conventions
  - Consistent tab-spacing for code readability
  - Testing components
  - Testing api calls on successes and failures

4. Testing, Validation, and Evaluation

4.1 Objectives And Tasks

4.1.1 Objectives

The objective of this test plan is to provide a list of procedures to be referenced during the development of the WeatherWear system. Such a framework will assure software quality

and security, and will assist in the development of the system.

4.1.2 Tasks

- Unit Testing
- System and Integration Testing
- Performance and Stress Testing
- User Acceptance Testing
- Beta Testing

4.2 Scope

General:

The major components to be tested are the client application (on both Android and iOS systems), the system back-end (API and database functionality), and the bi-directional testing between client and server.

Tactics:

Those developers who are responsible for writing their respective product components will write base tests during the same process that they write functional production code. Scale testing frameworks, such as batch testing, load testing, and beta testing, will be observed by testing agent(s) that collect useful data for further system development.

4.3 Testing Strategy

4.3.1 Unit Testing

Definition:

Unit Tests should be developed alongside normal code in order to facilitate good development practices. Unit tests will be written for all necessary function calls and should encompass all reasonable values for given parameters.

Methodology:

- Developers will write tests using Enzyme for visual and UI components.
- Jest will be used to unit test functions without rendering.

4.3.2 System and Integration Testing

Definition:

System and integration testing represents the functionality of each software component with other connected components. For example, the client user interface connects to the back-end host in order to interact with the clothing recommendation system.

Methodology:

The developer will send requests from the client application to the back-end to verify functionality, continuing to test each feature that touches the back-end throughout implementation.

4.3.3 Performance and Stress Testing

Definition:

Performance and stress testing will be performed on the project by testing the number of requests the back-end can properly handle in a reasonable amount of time for the client. Load balancing may need to be integrated if stress testing proves weak.

Methodology:

The team has tested the maximum amount of users using the Azure portal performance testing metrics against our uploaded server. The team will also be using stress and load testing node packages available in the NPM repository to overload our APIs with unrealistic amounts of traffic to find the crash point. Ethan and Tyler will be implementing the load tests using JMeter and the built in performance testing hosted on Azure.

4.3.4 User Acceptance Testing

Definition:

The purpose of acceptance test is to confirm that the system is ready for operational use. During acceptance test, end-users (customers) of the system compare the system to its initial requirements.

Methodology:

Acceptance testing will be performed by the team by using the application and comparing the possible use-cases of the application with that of the initial application vision. Acceptance testing is considered successful if the *WeatherWear* application performs the minimum specified functions listed in the design documentation.

4.3.5 Beta Testing

Definition:

Testing the application in a real-world environment. Users, including developers will engage in a week long test session where users will use the application and report any bugs back to the developers. Developers will use this feedback loop to patch issues and fine-tune the application workflow.

Participants:

Each member of the *WeatherWear* team will use this app for a week and bring their feedback directly back to the team. In addition to the 6 members of the team, each member will select another outside individual who they interact with daily to act as additional testing clients.

Methodology:

Each team member will select one to two outside parties and help them to install the *WeatherWear* beta application from the Expo site. During the week of beta testing, the team will collect feedback about issues with *WeatherWear* functionality and usability from both outside parties and other team members. This feedback will be used to patch the app during the week of testing and thereafter.

4.4 Hardware Requirements

- Clients will require either Android or iOS devices to run the WeatherWear app
- Server-side consists of an Azure hosted Node.js server running an Express.js service and an Azure hosted SQL Server and database

4.5 Environment Requirements

- Developers can use Expo to load version-by-version updates during development
- Clients for the beta testing phase of the application will be loaded onto personal devices via the Expo application sharing site

4.6 Test Schedule

Beta Testing Schedule:

- 1 week-long testing period, with constant feedback loop (daily feedback request from team members and outside users)
- Each day, the team will look to patch issues with app functionality and usability as long as those fixes do not compromise application functionality
- If a fix is made, the team will request that all members in the beta install the most up-to-date version of the application
- At the end of the testing period, other backlogged changes and additional features will be processed for integration into the *WeatherWear* Application

4.7 Control Procedures

Problem & Change Request Reporting:

All testing users will have access to a Google forms link where they will be able to submit feedback to the *WeatherWear* team.

4.8 Features To Be Tested

- App installation
- Google sign-in
- DarkSky api functionality
- Outfit recommendation system
- Adding clothing to the wardrobe
- Closet view
- Trip planning function

4.9 Features Not To Be Tested

- Image recognition
- Trip planning based on calendar

4.10 Schedules

Major Deliverables:

- Test Plan
- Test Cases
- Test Incident Reports
- Test Summary Reports

4.11 Dependencies

- Beta Testing will be contingent on a functional and deliverable application on the Expo site
- Deadline for all testing is Monday, April 15th

4.12 Risks/Assumptions

- Depending on severity of issues discovered during the beta testing phase of the project
    - Patches may need extra development focus during or immediately after week of Beta Testing.

4.13 Tools

- Expo Applications site
- Google forms for feedback
- Bug tracking via Gitlab

4.14 Functional Unit Testing

Application collects weather data.

Test Case:

For this, developers want to ensure that the application correctly retrieves information from the weather reporting service API, DarkSky.

Test Steps:

a. Manually collect weather data.
b. Check weather data collected by app against manually collected data.

Success:

Success in this test is measured by accuracy of reported weather compared to other reliably-collected weather data. If the app weather and current actual weather agree, it's successful. Weather responses matching 90% of the time are considered a success, matching the approximate accuracy of weather predictions.

Reasonable recommendations are given.

Test Case:

For this FR, the team ensured that the system can give the needed recommendations and that the recommendations are reasonable and correct. The sample wardrobe will include a set of clothing for warm weather and a set for cold weather, and provided a cold or warm weather pattern.

Test Steps:

c. Give program sample wardrobe.
d. Review returned clothing suggestions compared to weather.

Success:

In a wardrobe of warm clothing and cold clothing, any suggestion including the off kind of clothes is a failure. Otherwise, it's successful, with all returned clothing suggestions being appropriate to the weather conditions. Due to the polarized nature of this test case, suggestions should be accurate 100% of the time, or else some logic is incorrect.

User data is saved correctly and retrievable.

Test Case:

For this FR, the team made sure that user information is correctly stored in the database and retrieved from the database. A mock user account will have its clothing edited, saved, and re-accessed.

Test Steps:

e. Upload user information.
f. Load and edit user information through the application.
g. Check that the information was correctly saved.

Success:

A successful test here occurs when data is saved as intended. A new shirt, pants, socks, and underwear being entered and remaining altered after re-accessing is a success. Edits should be successful 99% of the time, leaving room for connection errors.

Clothing is correctly categorized and displayed.

Test Case:

For this FR, the team wants to ensure that when a user inputs clothing into the system, it is put into the correct place in the databases and can be accessed again.

Test Steps:

h. Input clothing items through the application.
i. Check the databases to ensure that the items were categorized correctly.

Success:

For each clothing option entered, checking that the database has the new clothing under the appropriate location will confirm a success or failure. Again, success should occur 99% of the time.

Application runs on Android and iOS.

Test Case:

For this FR, the team wanted be sure that the application runs on both Android and iOS at comparable qualities.

Test Steps:

j. Start the application on an Android and iOS device.
k. Run all previous tests for both systems.

Success:

Success for this case would be represented by no fails in the previous test cases, when replicated on both systems.

4.15 Non-Functional Testing

Performance

The user should receive a recommendation based on the weather in under 3 seconds.

Test Case:

The user will logout and login making sure that nothing will be saved in the cache, and then request a recommendation for their clothing. This should render on their phone in under 3 seconds.

Success:

Success in this test case is determined by the time between the app being opened to the responsible page, and the time taken for a suggestion to be displayed on the screen. As long as the time between request and completion of logic is under 3 seconds, it's a success.

The user should be able to be authenticated in less than 2 seconds.

Test Case:

When a user has logged out of the application or is creating a new account, the authentication service should authenticate them less than 1 second and be displayed in the database.

Success:

This case measures success in time between a request being created for authentication and getting a positive or negative response. As long as the time between request and response is 1 second or less, it's a success.

Security

Users will not have access to other user's information

Test Case:

Create a new user account within the database and test whether or not their privileges are escalated enough to make any malicious changes within the application and the database.

Test Steps:

a. Login as user
b. Makes changes to their closet and settings
c. Logout
d. Log back in as different user
e. Ensure all changes did not affect a different user

Success:

Success in this case is measured by a user having no way to access another user's data within the application

Usability

Efficiency of New Clothes

Test Case:

When the user is adding new items of clothes, they should be able to do this in an efficient manner. The user should be able to select what temperature range this should be worn at and the category of the article of clothing. This functionality should execute under 10 seconds based on user experience with technology. The user in the future may be able to add their clothing from a csv file making a web interface extremely efficient.

Success

Success in this area entails clothing items being added in less than 10 seconds in the plans current state. There may be updates to this criteria later based on future design plan changes.

Performance Tracking

Test Case:

Any process that takes more than 2 seconds should display a loading icon. This will allow the user to understand that the app is working and that it is not user error. This case should force a process to take extra time, upwards of two seconds.

Success:

The app should display a loading icon when a process takes more than 2 seconds 100% of the time.

Compatibility

External APIs

Test Case:

Test various ways the APIs are used for the application and how they can be plugged into the environment. This way the application can get the best use out of these APIs.

Success:

A success for each API is a response to API calls containing requested information- DarkSky giving weather feedback, for example.

4.16 Results of Test Implementation

The testing plan was configured and implemented successfully though the use of:

- Unit Testing
- System and Integration Testing
- Performance and Stress Testing
- User Acceptance Testing
- Beta Testing

After implementing all of the above test plan procedures, especially the beta testing period, the WeatherWear application shows significant improvement in problem areas and stability. User-acceptance rating for the application increased as an overall result of the testing protocols.

Performance and stress testing:

Running a performance test against our uploaded server in Azure reported and average response time of .10 seconds for each response. Using the SMARTBEAR LoadUI testing system, we ran a performance test over 5 minutes with 2741 tests run, with an average response time of 72 ms, as shown in Figure 3.

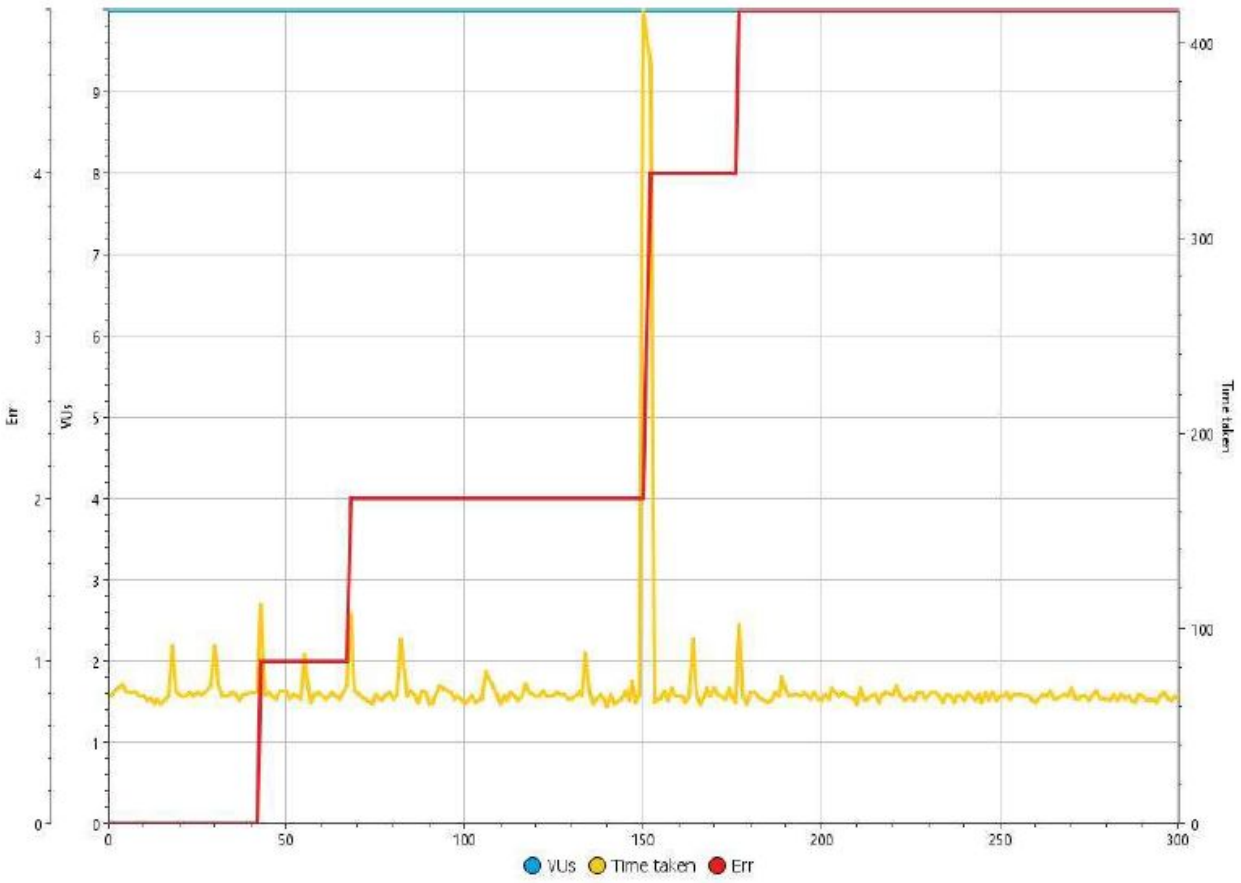


Figure 3. Load Testing Graphical Results

Beta testing results:

After the release of the beta version of our application the team opened bug reporting and feature reporting surveys to our beta users. The developer team received the results of those surveys and updated some parts of our app to match those reports. The reports included items such as:

- Padding at top on Android
- Navigation Icons on Android won't show up
- The "Outfit" page would not open for me.
- Android navigation back button doesn't function
- Camera still distorted
- On the home screen, the little cloud at the top gets cut off on my iPhone :(I would also love some more color on the home screen for the ~aesthetics~
- The current temperature would be helpful on the home screen.
- I don't know if its just my phone, but the "Outfit" page won't open for me.

- When I click "Profile", it takes me to a screen that just says "Open Modal" in the very top left (it is almost cut off). Its kinda weird that it's so far up in the corner.
- When I click "Closet", the "Open Camera" button is really far up on the screen, which makes it hard to click. Then, when I click it, I can't close the camera, because it is so far down in the bottom left corner.
- I'm assuming y'all will make it not say "test" and stuff in certain places.
- It is kinda weird how some of the words under "Name" and "Category" start with a capital letter and some do not. I enjoy consistency.

Front End Unit Testing:

Through the use of Jest and Expo, each screen contained within the application were tested for proper rendering under certain state conditions. When the tests were ran the rendering of the screen is compared to a snapshot constructed by Jest to test if it rendered properly. This process was ran on all individual screens within the app and all tests passed.

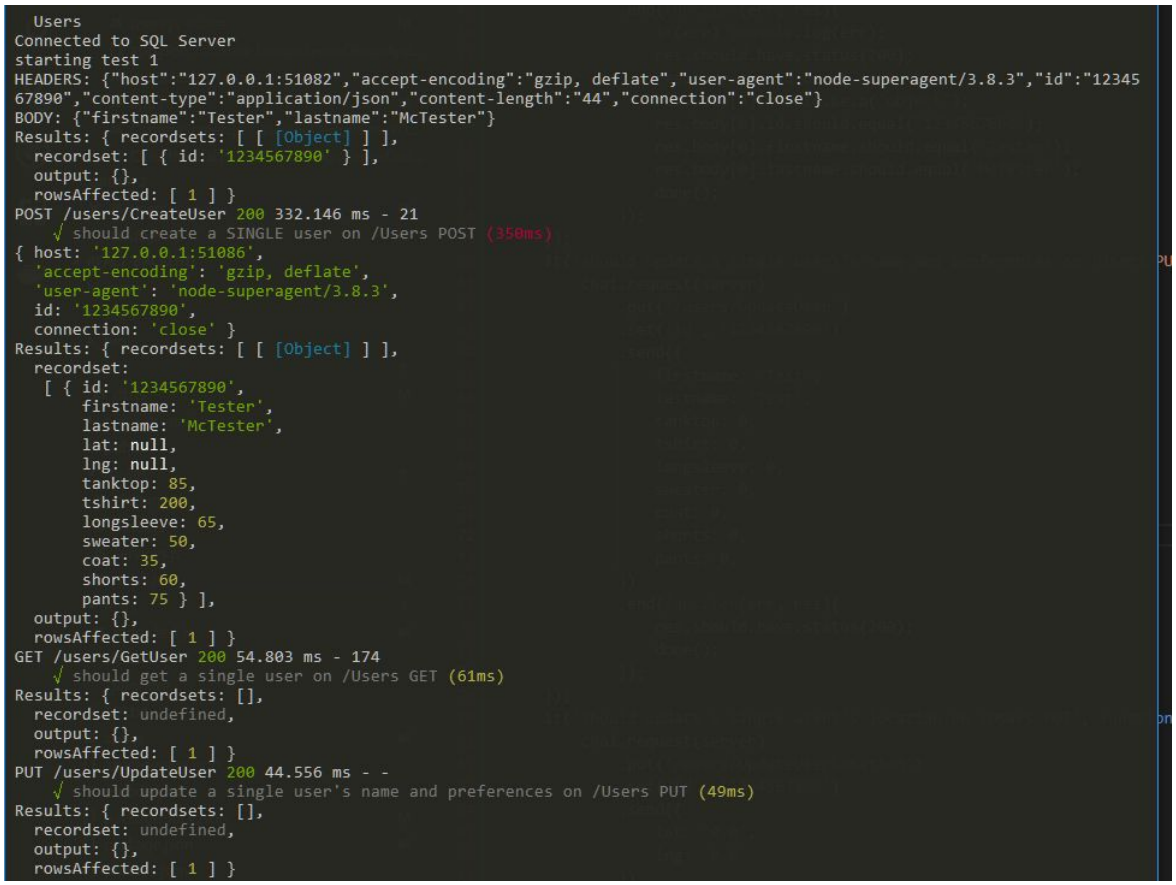
Back End Unit Testing:

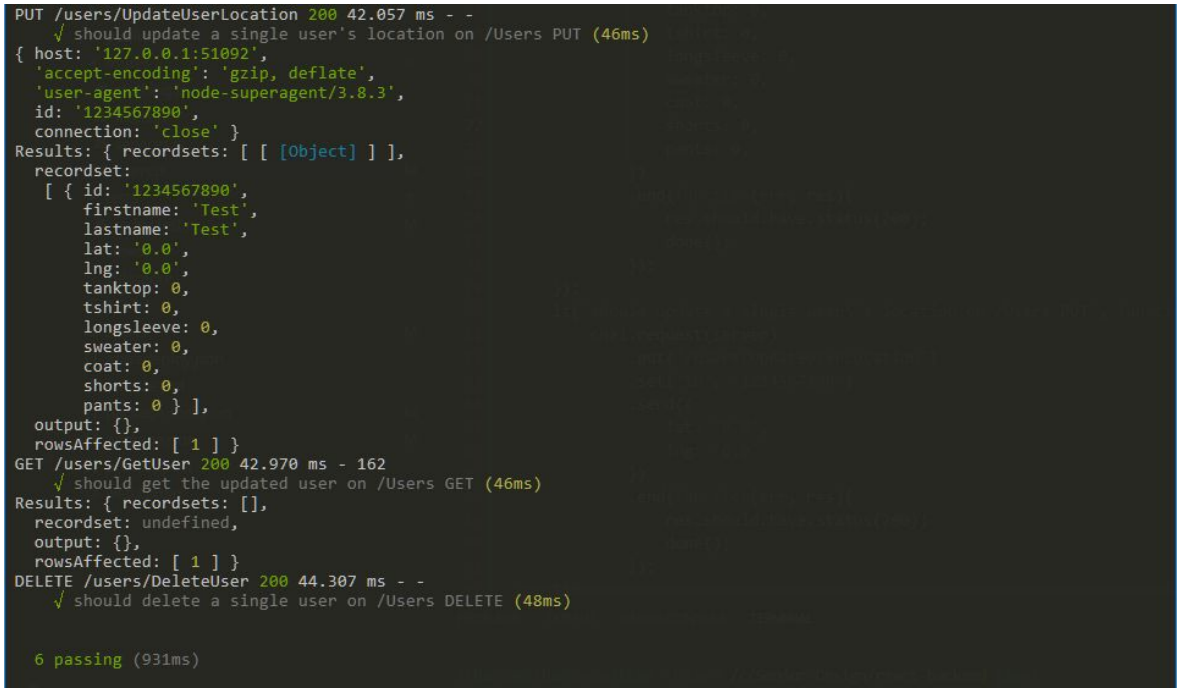
For back end unit testing we used the Mocha framework and the Chai "Should" framework to properly test every one of our API calls. Each of the tests in the screenshots below consist of a variety of *should* statements that encapsulate the proper return types and usability of our server APIs.

The following screenshot is the trip recommendation unit test and everything it is being tested for.

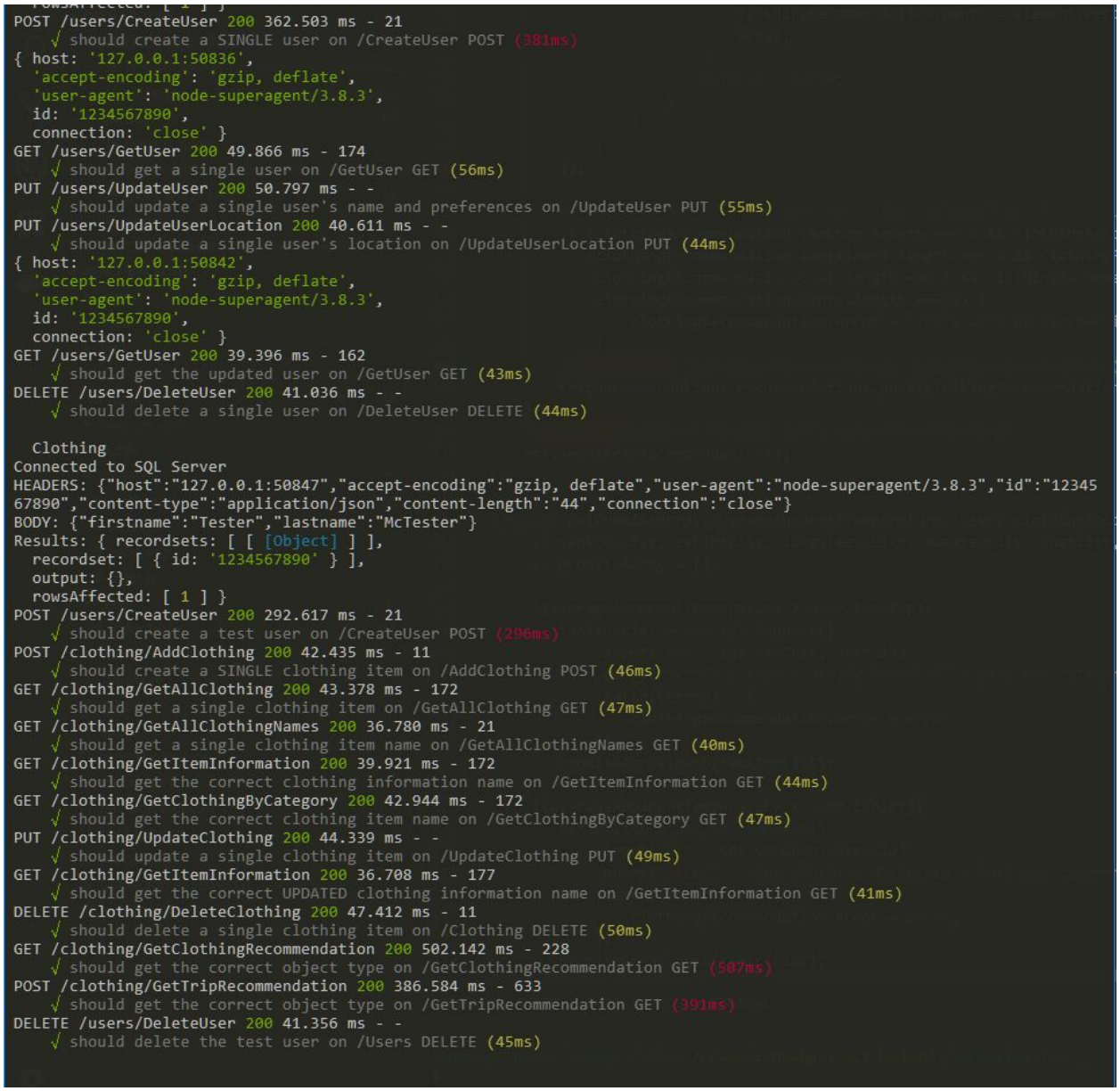
```
it('should get the correct object type on /GetTripRecommendation GET', function(done) {
    chai.request(server)
        .post('/clothing/GetTripRecommendation')
        .set('id', '1234567890')
        .send({"dates":[
                {"date": "2019-05-01T05:06:07", "dresscode": "casual", "lat": 42.0308, "lng": -93.6319},
                {"date": "2019-05-02T05:06:07", "dresscode": "casual", "lat": 42.0308, "lng": -93.6319},
            ]})
        .end(function(err, res){
            res.should.have.status(200);
            res.body.should.be.a('object');
            res.body.recommendations.should.be.a('array');
            res.body.recommendations.length.should.equal(2);
            res.body.recommendations[0].tanktop.should.be.a('array');
            res.body.recommendations[0].tshirt.should.be.a('array');
            res.body.recommendations[0].longsleeve.should.be.a('array');
            res.body.recommendations[0].sweater.should.be.a('array');
            res.body.recommendations[0].coat.should.be.a('array');
            res.body.recommendations[0].shorts.should.be.a('array');
            res.body.recommendations[0].pants.should.be.a('array');
            res.body.recommendations[0].date.should.be.a('object');
            res.body.recommendations[0].averageApparentTemperature.should.be.a('number');
            res.body.recommendations[0].error.should.be.a('string');
            res.body.recommendations[1].tanktop.should.be.a('array');
            res.body.recommendations[1].tshirt.should.be.a('array');
            res.body.recommendations[1].longsleeve.should.be.a('array');
            res.body.recommendations[1].sweater.should.be.a('array');
            res.body.recommendations[1].coat.should.be.a('array');
            res.body.recommendations[1].shorts.should.be.a('array');
            res.body.recommendations[1].pants.should.be.a('array');
            res.body.recommendations[1].date.should.be.a('object');
            res.body.recommendations[1].averageApparentTemperature.should.be.a('number');
            res.body.recommendations[1].error.should.be.a('string');
            done();
        });
});
```

The following screenshots show the results of the user API call testing including the response recordset for each call.





The following screenshot is the results of all tests running and passing without the printed recordsets.



5. Project and Risk Management

5.1 Task Decomposition & Roles and Responsibilities

Ethan Wieczorek:

Role: Lead Back-end Developer

Responsibilities:

- Working with the group to make software architectural decisions.
- Administrating and examining merge requests.

- Designing and maintaining the outfit recommendation algorithm and screen.
- Designing and maintaining the trip planning algorithm and screens.
- Dealing with API synchronicity issues and promises.
- Back end unit testing
- Working with Will to maintain working code throughout the merge process.

Nick Eaton:
Role: Product Manager
Responsibilities:

- Design of app and visuals
- Design of testing-structure and product timeline
- Documentation of product implementation decisions
- User Interface/User Experience design.

Nicus Hicks:
Role: Head of Report Development
Responsibilities:

- Create base pages to be added on, and do early research for future features
- Designing the closet view
- Responsible for any upkeep and additions to the website including but not limited to:
    - Documentation
    - Weekly Reports
    - Presentations

Will Parr:
Role: Lead Front-end Developer
Responsibilities:

- Develop user-facing features
- Working with group to set up developer environments with Expo and Node.js.
- Frontend external and internal API integration
- Translate wireframes and designs into code
- Front end unit and component testing
- Working with Ethan to maintain working code throughout the merge process

Tyler Witte:
Role: Lead Software Architect
Responsibilities:

- Designing and deploying build pipelines
- Designing database layout and database management
- System architectural decisions
- Developer support

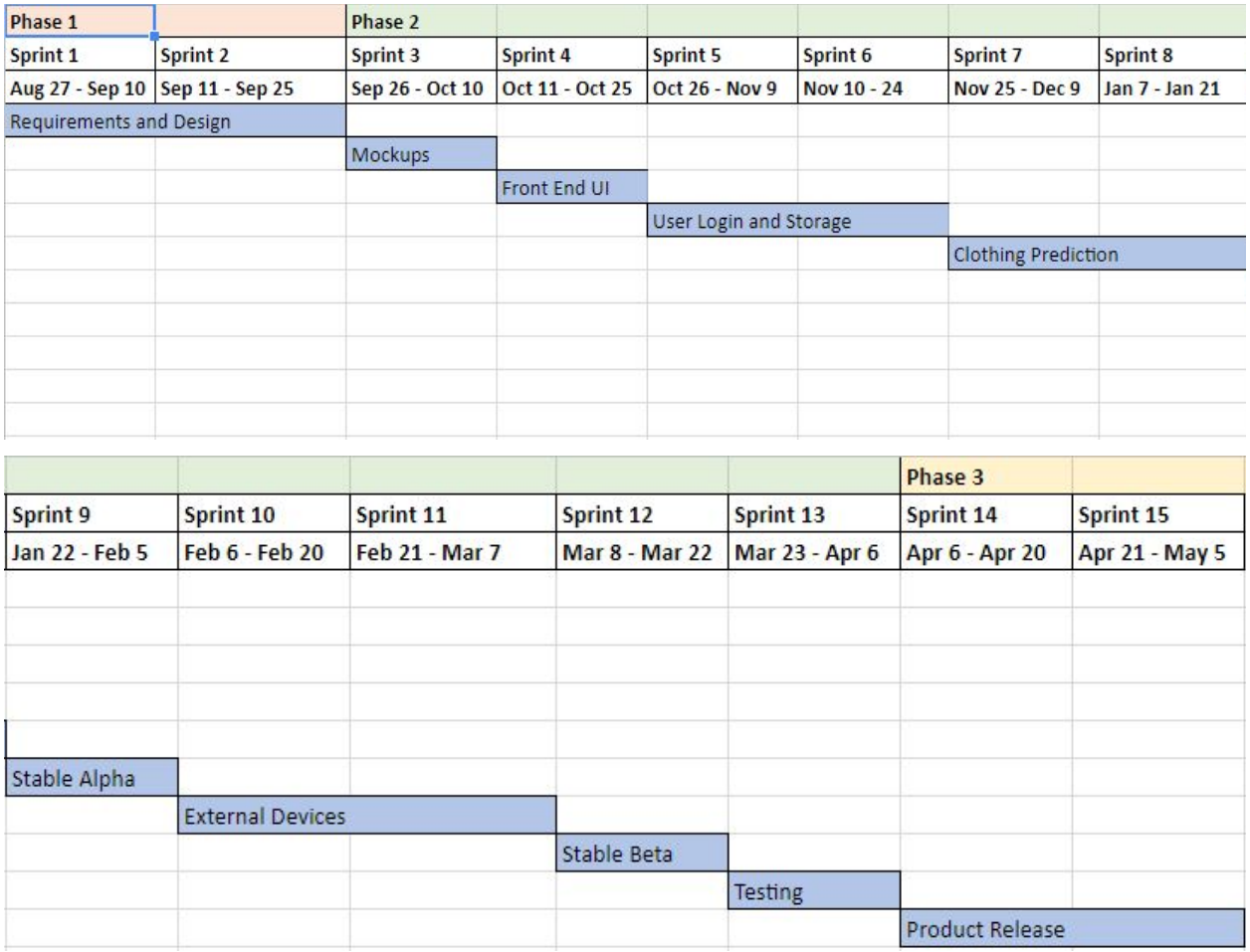
- Performance and Load Testing

Christian Ehlen:
Role: UI/UX and Quality Assurance Developer
Responsibilities:

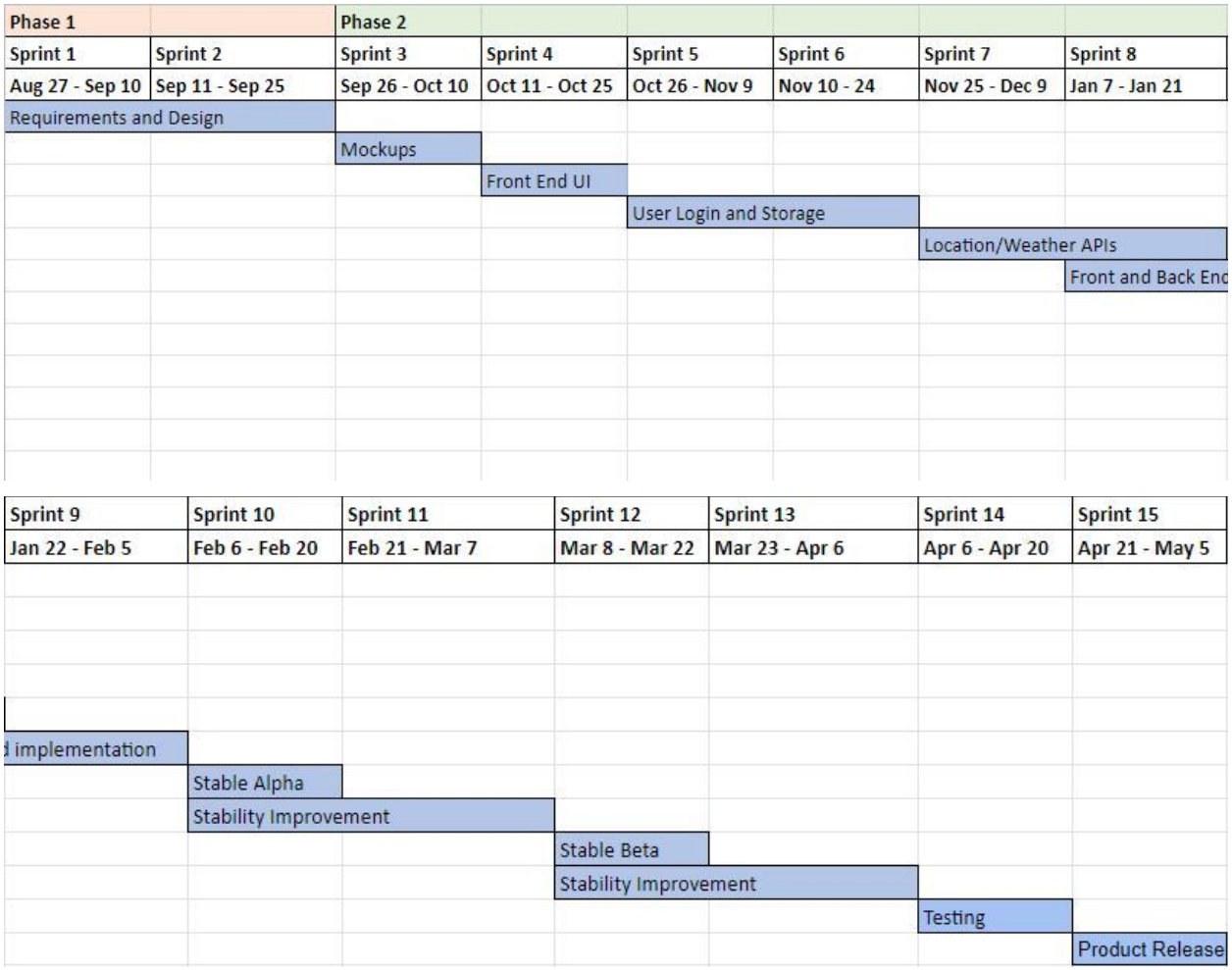
- Designing the visuals of the app and styling components created by every developer
- Testing application style and function on both iOS and Android to ensure full cross compatibility
- User Interface/User Experience design.
- Front end unit and component testing

5.2 Project Schedule

5.2.1 Proposed Gantt chart



5.2.2 Actual Gantt Chart



5.3 Project Risks

5.3.1 Anticipated Risks

- Licensing
    - Firebase
    - Azure
    - Apple developer license
    - Android developer license
- Loss of time due to various reasons
    - Issues with code that the team was not easily able to overcome
    - Immature Google Login API released from Expo
    - Changing client requirements caused loss of time rewriting code
    - Limited knowledge of technology/low previous experience

    - Change of code infrastructure/architecture
- Incorrect budget estimation
    - Expansion of project scope to increase costs
    - Increased unit testing causing us to reach daily limits against paid APIs
- Information security
    - Authorization/security issues against our back end

5.3.2 Actual Risks

- Synchronicity Issues with the API
    - Mitigation:
        - The team researched asynchronous functions and promises and used the results of the research the team completed to create a fully functional API that could be referenced synchronously or asynchronously.
- Limited knowledge of technology/low previous experience
    - Mitigation:
        - Assigned tasks to teams of developers for pair programming.
        - Using outside materials to learn the best practices of React development.
        - Working as a group to overcome difficult to solve obstacles.
- Licensing
    - Firebase
        - Mitigation: The team received a free student account and an initial $300 in free request credits through Firebase. The team did not end up using the $300 at all and stayed within the realm of the free storage throughout our development and testing.
    - Azure
        - Mitigation: The team received a free student developers license and hosted our Node.js server as well as our SQL database
    - Apple Developer License
        - Mitigation: The team made a group decision not to use the Apple app store for testflight and to release a test version through Expo
- Apple's testflight review process and release via Expo
    - Mitigation:
        - The team released our app publicly for testing via the Expo app center on Android and released the iOS version to a select few individuals with our Expo developer account.
- Change of code infrastructure/architecture
    - Mitigation:
        - The team rescoped our timeline to match our server change when the team switched to Node.js.

        - The team did not incur any additional costs and were able to cut off some of the time planned for learning the code in the back end by switching to Javascript.
- Information security
    - Mitigation:
        - The team used Google's authentication service with our login in order to authenticate user IDs against our back end
- Google login integration due to immature language properties
    - Mitigation:
        - The team temporarily created our own login system using Firebase while working with the React Native libraries and waiting for Google's authentication service to become available on React Native.
- Redux
    - Mitigation:
        - After 2 weeks of experimenting the team moved Redux to our backlog and had a group spike about it. The team ended up using a React Native global state management system by mapping the state of the login to the props of each child component and using the included Firebase redux store to check user authentication.

5.4 Lessons learned

After working as a group for an entire year the team overcame obstacles, grew as a group, and put immense effort into creating a project that the team feels exceeds the given expectations for the project. The team has learned how to mitigate issues that could not be foreseen and how those issues affect the overall lifecycle and schedule of a project. Over the course of the project the team has had to deal with issues both in and out of our control and how those issues affect a project timeline. Dealing with those issues has taught the team how to overcome issues out of the developer's control and redesign the structure of a project to still meet the client requirements within a given schedule.

Throughout the semester the team has had to grow as individuals and a group in order to succeed to the best of every member's abilities. The team assigned tasks to work on as pairs in order to grow our skill in areas where any members were not currently the most successful by allowing the developers with more experience in that area to learn from their peers.

6. Conclusions

6.1 Closing Remarks

By the end of development, the team has implemented the base functionality of the application with closet management, clothing recommendation, and trip planning outfit recommendations. Possible extensions to this project for the team and/or future developers include:

- The implementation of Google Calendar Integration for clothing recommendation based off calendar events for daily recommendation and trip planning.
- The addition of a system to take fashion into consideration when recommending clothing to users.
- The addition of a Favorite Outfits sections for users to define outfits they want to be recommended more often and as a unit.
- Saving every outfit the user selects on a daily basis in order to learn what clothes go best in each type of weather.
- The addition of a Machine Learning component to the recommendation system to keep track of trends in users' clothing selections and recommend similar clothing.
- The addition of an advertising/sponsoring program that suggests the purchase of certain clothing items when the user does not have proper clothing for the day's weather
- A push notification system that saves every user's last recorded location and watches for drastic weather changes to notify users of new daily needs

6.2 List of References

Npm, (2018). React-google-calendar-api. [online] Available at: https://www.npmjs.com/package/react-google-calendar-api [Accessed 12 Oct. 2018].

npm. (2018). *react-native-weather*. [online] Available at: https://www.npmjs.com/package/react-native-weather [Accessed 12 Oct. 2018].

Google, (2018). Firebase Database [online] Available at: https://firebase.google.com/ [Accessed 20 Oct. 2018].

DarkSky API. (n.d.). Available at: https://darksky.net/dev [Accessed 20 Oct. 2018].

Getting Started – React. (n.d.). Available at: https://reactjs.org/docs/getting-started.html [Accessed 28 Nov. 2018].