App for Microgrid Demonstration

Final Report

Team: sddec21-21

Client: Anne Kimber

Advisors: Mathew Wymore, Steve Nystrom, Nicholas David

Team: Gabriel Rueger, Michael Doyle, Micheal Thai, Patrick Shirazi, William Bronson

Email: sddec21-21@iastate.edu

Website: http://sddec21-21.sd.ece.iastate.edu

Revised: December 8, 2021 - Final

Executive Summary

Development Standards & Practices Used

- RFC 7231 - HTTP standards (Fielding)
- RFC 6455 - Websocket Design Standards (Melnikov)
- Coding Standards to maintain quality code base (Xuefen)
- Digital design standards for mobile applications (Designing)
- Git source code control
- Agile Methodologies

Summary of Requirements

- The server shall have the ability to add additional sites from the microgrid
- The database shall have the ability to add additional data sources to the database for each site
- The database shall be able to query and search different subsets of the data
- The database shall have a configurable data collection interval
- The database shall support automatic archiving of data
- The mobile application shall display data within a minute of collection
- The database size shall scale linearly with the number of data sources and with time
- The server shall reduce old data to average data over a period of time
- Frameworks, libraries, etc. for this project shall be well-supported and maintained
- Throughout this project, open and well-supported communication standards shall be used
- All decisions made throughout this project shall be well documented

Applicable Courses from Iowa State University Curriculum

SE/CPRE/EE 185
COM S 227, 228, 309, 363
S E 329, 339

New Skills/Knowledge acquired that was not taught in courses

- Apache Cassandra
- Docker
- Enzyme
- Jest
- React Native

Table of Contents

1 Introduction 6

Acknowledgment 6

Problem and Project Statement 6

Operational Environment 6

Requirements 6

Intended Users and Uses 7

Assumptions and Limitations 7

2 Design **8**

2.1 Design Thinking 8

2.2 Design 8

2.2.1 Server and Database 8

2.2.2 Data Collectors 9

2.2.3 Frontend 9

2.3 Technology Considerations 9

2.4 Development Process 13

2.5 Design Revisions 14

3 Implementation **14**

3.1 Mobile Application 14

3.1.1 Live Data Display 15

3.1.2 Data Export 15

3.2 Backend Server 16

3.2.1 Service 16

3.2.2 Controller 16

3.2.3 Websocket 18

3.3 Database 18

3.4 Data Collectors 19

3.4.1 Tesla Powerwall 20

3.4.2 Dranetz 20

3.5 Docker 22

4 Testing 23

4.1 Unit Testing 23

4.1.1 Backend Testing 23

4.1.2 Frontend Testing 23

4.2 Integration Testing 23

4.2.1 Backend Testing 23

4.2.2 Frontend Testing 23

4.3 Acceptance Testing 24

4.4 Results 24

5 Related Products 24

6 References 27

Appendix 28

I Operation Manual 28

I.a Deploying The Backend 28

I.A.1 Data Collection Configuration 28

I.A.2 Running The Application 29

I.A.3 Creating a Site 29

I.b Starting The Mobile Application 29

I.b.1 iOS/Android Setup 29

I.c Using The Mobile Application 29

I.d Running Tests 30

I.d.1 Backend Tests 30

I.d.2 Frontend Test 31

II Alternate Versions of the Design 31

List of figures/tables/symbols/definitions

(Figure 1. Component Diagram) 8

(Table 1. Comparing Backend Frameworks) 10

(Table 2. Comparing Frontend Frameworks) 11

(Table 3. Comparing Database Frameworks) 12

(Table 4. REST API Endpoints) 17

(Figure 2. Database Schema) 18

(Table 5. Tesla Powerwall Values) 20

(Table 6. Dranetz Values) 22

(Table 7. Database Schema Test Results) 24

(Figure 3. Tesla App Pages) 24

(Figure 4. Dranetz Text Meter Page) 25

(Figure 5. Our Mobile App Pages) 26

1 Introduction

1.1 Acknowledgment

We would like to bring special attention to our lead project advisor Anne Kimber and advisors Nicholas David, Steve Nystrom, Mathew Wymore. Without the support and guidance of these individuals, our project would not have even gotten its feet off the ground. We are very grateful for the help they gave us along the way.

1.2 Problem and Project Statement

The ISU Electric Power Research Center operates and conducts research on a site that has microgrid data on it from multiple data sources such as a Tesla Powerwall and Dranetz. The only current access to this data is either by driving to the site itself and manually accessing the onsite computer, or by creating a remote connection to the computer.

For our project, we will create an application to retrieve and present energy data collected by a solar site to public users and site maintainers. The user can interface with the application through a mobile app. This application will serve as an efficient way for public users and site maintainers to analyze efficiency of power collection and distribution of the solar site from data presented through the application. Site maintainers will be able to gain a better understanding of how a site collects energy over time. Site users will know what the site could be used to power given how much power is stored in the site's battery storage.

1.3 Operational Environment

The mobile application will need to be functional on iOS and Android mobile platforms. This application will be exposed to and used by the public to gain information about the sites. The backend pieces of the application, consisting of the spring boot server, cassandra database, and data collecting scripts will be containerized using docker. Thus, the operating environment for the backend needs only to support docker, as other dependencies will be provided within the container contexts.

Additionally, a vital point of the operational environment is the network environment. For our application, the device must be connected to the internet. It can be connected to any network, not solely just the site or ISU's network.

1.4 Requirements

Functional Requirements:

- Ability to add additional crates
- Ability to add additional data sources
- Query and search different subsets of the data
- Configurable data collection interval
- Support automatic archiving of data
- The mobile application's graph must continuously update over time
- The graph must be configurable to display different data sources

Non-functional Requirements:

- Data must be displayed within a minute of collection
- Database size scales linearly with number of data sources and with time
- Old data reduced to 10 minute averages
- Frameworks, libraries, etc. must be well-supported and maintained
- Must use open and well-supported communication standards
- All decisions made throughout this project must be well documented
- The system must be secure but does not require authentication

Other requirements for the system is that it must have negligible cost as there is not an existing budget for the application. Further, it must integrate into the Electric Power Research Center's existing environment.

1.5 Intended Users and Uses

This mobile application is intended for educational/informational purposes and publicity and will display the microgrid's overall performance. Additionally, researchers should be able to access the voltage, current, and frequency data readings.

1.6 Assumptions and Limitations

The following assumptions are as follows:

- The mobile app will only take data from one solar site during development.
- Students will be given a way to access the microgrid data over some api or other connection.
- The mobile application should be able to display a time range of data from a site in a graphical format.
- The mobile application only needs to use the English language.
- The team will be provided with a virtual machine to host the application.

The following limitations are as follows:

- The budget to produce the application will be negligible.
- The mobile application will be developed by the end of the 2021 Fall semester.
- Student access to the Sun Crate site is very limited, have to rely on the EPRC team

2 Design

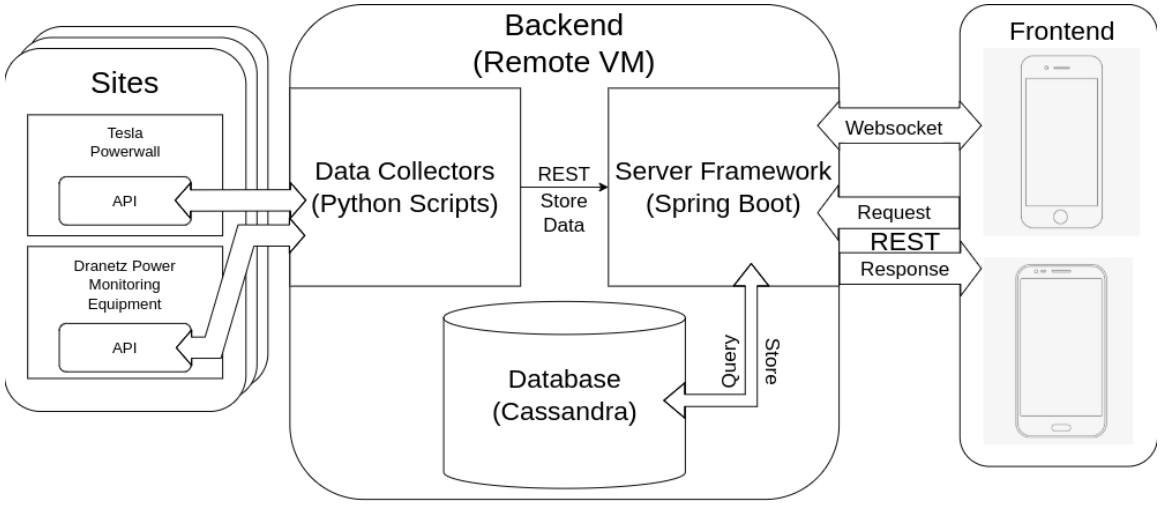
2.1 Design Thinking

One requirement that will largely shape the design of the application is the scalability requirements of the system to include new data sources and sites in the future. This leads to the idea of creating a modular application and architecture such that it is easy to add new pieces to it in the future. Our architecture will be modular by virtue of having the database and frontend being cleanly separated. The application presented to the user should be intuitive to new users and provide information in a manner that is easy to understand. The user should have accessibility to obtain the application and see certain data that the public should be able to see.

While keeping the aspects that shape our design in mind, we made numerous design choices that will address these needs. We chose a frontend framework that can provide a user-experience typical of applications native to the devices we intend to support. We also have requested a server and chosen backend and database frameworks that can scale well and are easy to modify in the future.

2.2 Design

Our system consists of 3 types of core components: server and database, data collectors, and the frontend.



(Figure 1. Component Diagram)

2.2.1 Server and Database

The server and database will run on the virtual machine provided to the team. The main functionality our server provides to the application is receiving data collected by our data collection system and diffusing it throughout the rest of the application. Specifically, it will store the data into our database and also send the data across a websocket connection to any connected frontend instances.

The backend will also provide many different REST endpoints allowing the frontend to query for any information that it needs to display. This includes information about the different sites, the data sources a site has, and the data collected from each datasource.

2.2.2 Data Collectors

The data collectors will run as python scripts on the provided virtual machine. Each data collector will be designed to collect data from a specific data source a site has. The data collector will access its data source through a network connection. Depending on the design of each data source, the data collector can do this by accessing the data source's API, scrape the data source's local webpage, or other means. After data has been collected, the data collector will then parse and transform the data into the specific format that our server can consume, and send the data via an HTTP post request to the server.

Data collectors will be designed to be as configurable as possible, with options on how frequently they should collect data from their data source and what data from the datasource to actually collect.

2.2.3 Frontend

The frontend will be a cross platform mobile application developed for iOS and Android ecosystems. Its main functionality will be to give users the ability to view data collected from the sites and datasources. To accomplish this, the frontend will have different screens for different features the app provides. This includes a screen to display graph data collected over time and a screen to allow users to export and download data.

There are two main ways the frontend will access data from the server. One way is through HTTP requests. This will be used to access existing data stored in the database. The other way is through a websocket connection. This method will be used to access live data collected from the system.

2.3 Technology Considerations

To highlight the strengths, weaknesses, trade-offs and choose the most effective backend framework out of Spring Boot, Laravel, Django, and Flask for our mobile app development, we focused on six key elements: License, programming language, age & documentation, performance, professional projects, and team experience. Below we will elaborate on importance of each elements:

License

Grants a better understanding of the framework from a financial perspective. Given monetary restrictions on the project, the framework ultimately selected should be free to use recreationally and commercially to avoid any legal concerns. Licenses of selected projects should reflect this consideration.

Programming Language

It is important for the team to select a framework that utilizes languages that are familiar. This will cut back on unnecessary learning curves that come along with learning new programming languages.

Age & Documentation

How long a project has been around is a good indicator of how well a project is supported. This support and open documentation becomes useful when researching different functionalities and setting everything up. It also can signify whether the framework will continue to be supported through the development cycle and after the final application is completed. It would be detrimental to the application's health and security to implement a framework that ends up no longer maintained.

Performance

Our project states requirements regarding response times and frequency of data collection from the site. The framework our team chooses could have a large impact on the project's ability to meet these requirements. When looking at performance, it is helpful to examine attributes such as multithreading capabilities to process multi requests at once.

Professional Projects

This category is a good reference to whether our application can be implemented using the selected framework. A framework that is used by a larger project that has similar function has a good chance to work in our project.

Team Experience

Akin to our programming language criteria, team experience is to be considered as a framework that our team has experience with will ultimately be easier to implement and time efficient.

Backend Frameworks	License/Cost	Language	Maturity	Performance	Team Exp.
Spring Boot	Apache (Free)	Java	April 2014	Large binaries, multi-thread capable	Experienced
Laravel	MIT (Free)	PHP	June 2011	Slow for large projects	None
Django	BSD 3-Clause (Free)	Python	July 2005	single thread	None
Flask	BSD 3-Clause (Free)	Python	April 2010	Limited concurrent request support	None

(Table 1. Comparing Backend Frameworks)

Spring Boot Framework:

- License: Apache License (v2.0)
- Programming Language: Java
- Age & Documentation: Released in April of 2014, and official website contains various example projects and helpful guides
- Performance: Autoconfiguration may add unnecessary dependencies making binaries larger than necessary. In addition, it can handle multiple requests.
- Professional Projects: Inuit & Zalando
- Team Experience: Software and computer engineers have experience through COM S 309

Laravel Framework:

- License: MIT License
- Programming Language: PHP
- Age & Documentation: Released in June of 2011, and official website contains documentation

- Performance: Generally slower for larger projects
- Professional Projects: 9gag & Kmong
- Team Experience: Unfamiliar

Django Framework:

- License: BSD 3-Clause
- Programming Language: Python
- Age & Documentation: Released in July of 2005, and official website contains documentation, tutories, topic guides, and installation help
- Performance: Performs well for large projects, feature-heavy, which may feel bloated for large projects, and can only handle one request at a time.
- Professional Projects: Pinterest, Instagram, and Robinhood
- Team Experience: Unfamiliar

Flask Framework:

- License: BSD 3-Clause
- Programming Language: Python
- Age & Documentation: Released in April of 2010
- Performance: It is suitable for smaller projects and doesn't handle concurrent requests as well as others.
- Professional Projects: Netflix, Reddit, and Lyft
- Team Experience: Unfamiliar

Frontend Frameworks	License/Cost	Language	Maturity	Relevant Platforms
Qt	Paid	C/C++, JS, HTML, QML	1991	Android, iOS
React Native	MIT (Free)	JS	2015	Android, iOS
Flutter	New BSD (Free)	C/C++, Dart, Skia	2017	Android, iOS

(Table 2. Comparing Frontend Frameworks)

QT Framework:

- License: Free open source license available, free educational license available, paid commercial license available, and separate distribution licensing available.
- Supported Platforms: Windows, macOS, Linux, Android, iOS
- Programming Languages: C/C++ for application framework and JavaScript, HTML, and QML for UI
- Maturity: It was first written in 1991 and had a long history of public bug reports and forums available. Also, there is extensive documentation on all available QT classes.
- Other Notable Aspects: Multithreading support

React Native:

- License: Free MIT License
- Supported Platforms: Android & iOS
- Programming Languages: JavaScript
- Maturity: It was founded in 2013 but released in 2015. It is supported by Facebook and also receives contributions from individuals and companies.
- Other Notable Aspects: It aims for a truly native feel on apps and does not support multithreading.

Flutter:

- License: Free New BSD License
- Supported Platforms: Android & iOS
- Programming Languages: C/C++, Dart, and Skia for UI
- Maturity: It was founded in 2017 and supported by Google. Because it's recent and new, there is not much support found online.
- Other Notable Aspects: It does not support true multithreading.

Additionally, we chose to pivot towards license availability and pricing, supported platforms, programming languages, maturity, and other notable aspects for the frontend frameworks. Given financial limitations on the project, selection based on free licensing was key. Maturity of the project was also a major consideration because the project will need to be supported into the future.

Database Frameworks	Storage	Schema	Team Experience
SQL	SQL Tables	Schema defined	Experienced
MongoDB	JSON Structure	No schema	None
Apache Cassandra	Tables	Schema	None

(Table 3. Comparing Database Frameworks)

SQL (MySQL & PostgreSQL):

- Data Storage: Data is stored in tables.
- Schema: The schema defines the database structure, meaning all rows must have the same structure.
- Team Experience: Familiar
- Other Notable Aspects: It can use JSON type to handle adding new data source components, but it will require more space because JSON will be stored as a string.

MongoDB:

- Data Storage: Data is stored in a JSON-like structure.
- Schema: There is no schema, and it is much easier and efficient to change.
- Team Experience: Each team member is unfamiliar with it, and there are poor online reviews about it.
- Other Notable Aspects: MongoDB query language

Apache Cassandra:

- Data Storage: Data is stored in tables.
- Schema: Must follow a schema, however allows for more complex column types including maps and lists for flexibility.
- Team Experience: None
- Other Notable Aspects: No joins - Query driven schema design, Built to be distributed and scalable.

Finally, we chose to center the database to compare data storage, schema, team experience, and other notable aspects.

2.4 Development Process

During development, the team followed agile development methodologies. At the beginning of each biweek sprint, the team planned a set of features and functionalities to work on. Issues were created using Gitlab's issue tracking system and features were developed on their own branch attached to the issue. Once a feature was certified as complete, it was then merged back into the main development branch. The team held weekly meetings in which we discussed the state of the project and what the team was working on. We also meet weekly with our clients and advisors to demo progress and gather requirements and feedback. At the end of each biweekly sprint, the team reviewed accomplishments and planned the next sprint.

2.5 Design Revisions

One major design revision we had was changing from a MySql database to an Apache Cassandra database. This change arose from feedback the team received from our initial design presentation. While in the context of a single site collecting data it was deemed that using MySql was a sufficient solution for our database, concerns were raised about the ability for it to reach the requirements of scalability the project aims for in future iterations. Thus switching to use Apache Cassandra made a lot more sense as its distributed nature allows it to be highly scalable and fault tolerant.

Another design change that we made was separating out the data collector script from a single script for each site, to a single script for each datasource in a site. This change makes updating the site to include different data sources much easier as we only have to add or remove running data collector scripts, and not change existing ones. It also allows for better reuse across sites as if multiple sites have the same datasource, the same script (although different instances configured differently) can be used for each.

3 Implementation

3.1 Mobile Application

Using React Native, we developed a mobile application for both iOS and Android which will allow us to monitor data from different sites. With the overarching goal of creating a user-friendly application which allows monitoring complex data in mind, we created several pages which allow for viewing and obtaining this data in a simple manner. The primary method of viewing data from a

site is the Live Data Display which is covered in the next section. Another method for obtaining data over an extended period of time is the Data Export page which is covered in section 3.1.2.

An important aspect to consider for every aspect of the implementation of this frontend application is that it was designed with multiple sites in mind. Given that there is only one site in operation at the time of writing this report, some parts of the interface such as where the user selects a site may seem strange, but all of the methods for selecting a site were designed so that they will automatically populate with more site options when more sites are constructed. Before the user can navigate to either the Live Data Display or the Data Export page, they will have to select which site they wish to view. At this time there will only be one site for them to select, but having this interface in place is an important part of meeting the requirements for this project to allow for additional sites in the future.

In numerous places in this app, HTTP requests using the backend server's Rest API are used to obtain information regarding sites and the data sources in them. These requests are used to present the sites available to a user and allow them to also see what data sources and data points are available for a given site. Furthemore, communication between the mobile application and server is accomplished via a websocket connection. This websocket connection is used primarily for obtaining live data from a given site as it is entered into the database.

There are many libraries which are used throughout the mobile application. One of these is react-native-vector-icons which provides a simple way to include icons throughout the project where applicable. The react-navigation library is also an essential component of the mobile application as it manages navigation between the different screens in the application.

3.1.1 Live Data Display

The Live Data Display is the method for allowing a user to see live data from a given site. This page allows for a user to select data sets from a given site and monitor them in real time. Up to two data sets can be viewed on a graph, and an additional two data sets can have their current value monitored apart from the graph.

HTTP requests to the server are used to provide the user with the data sets they may select, and they are also used to populate the graph with a period of previous data from the database when a data set is selected. A websocket connection is used to obtain live data from the site. When the Live Data Display is opened, a websocket connection is established to a channel on our server which is broadcasting all new data being stored in the database for the given site. This allows us to continue to display real time data in the graph in a timely manner which is an important aspect of meeting our requirements for this project.

The graph which shows data from a site is created using the react-native-svg-charts library. This library allows for graphing multiple data sets on a single chart which is essential for our use case. Furthermore, this library allows us to add multiple y-axes and an x-axis with data values of any format such as the time values we display on our x-axis. This library proved to be efficient and managed the dynamic thousands of data points being displayed on our graph very well.

In addition to the primary Live Data Display, the user may also access two more pages from this page which provide information relevant to the selected site. The first of these pages displays the potential solar power that may be generated by this site. This includes a monthly and annual

estimate of the power that will be produced by the site, and it provides an estimate of the power that will be produced each hour over the next 24 hours. This information is obtained via an HTTP request to the PVWatts V6 API which is made available by the Nation Renewable Energy Laboratory.

The other additional page the user may access from the Live Data Display is a page to view the weather at the selected site. This site will show the current weather conditions for a given site including information that is more relevant to solar generation such as percentage of cloud cover, sunrise time, and sunset time. A forecast for the next seven days is also included on this page. The information displayed on this page is obtained via an HTTP request to the OpenWeather One Call API.

3.1.2 Data Export

The data export use case is an integral part of the application as it allows users to download the data from a datasource in order to do their own analysis. In implementing this functionality, a number of different libraries were used.

In order to allow users to select the dates in which to download the data, we used a react-native-month-year-picker. This library allowed us to use the native iOS and Android UI components to select this information to give a more native feel.

To actually download the CSV from the server, we used the rn-fetch-blob library. This allows us to download a file from an endpoint and get a copy of it on the device. One important consideration is the size of these files, which is why we use the fileCache option to store a temporary copy in the device file system rather than hold it in memory.

Finally, we utilize the share library to handle the file after download. The share library brings up the native share capability of the device, allowing users to download the file, share through messages or email, or send the file to other applications.

3.2 Backend Server

Given our requirements, the backend component of our application provides two key aspects. The first of these being intercommunication with all other components of the system, most notably the frontend, devices at a remote site, and the database hosted by a remote virtual machine. Our implementation provides this communication via defined REST APIs and a bi-directional websocket. The APIs created for this project allow for both site devices and frontend application users to interact with data stored in the database, as long as there is an internet connection. These APIs provide important actions necessary for the functionality of our application. The other key aspect that was important when creating our backend was modularity. To achieve this, all backend components needed to be as containerized as possible to allow for future modification.

Using the Spring Boot framework, our team has defined the backend structure as sub-components. This implementation aims to segregate functionality, making simple modifications to the system more organized and impactful to only these sub-components. These sub-components will be described in more detail in the following sections.

3.2.1 Service

We implemented services as a modular way to interact with our database and models created to represent our data entries. Our application supports four different services: datasource, measurement-archive, measurement, and sites. These services implement functions that can be used and associated with controllers. Our implementation of these service components separates functionality dependent on different data models present in our project. These functions have direct interaction with database repositories in our project. Services act as a way to implement logic specific to an associated data model. This process makes modifying logic simpler as there is one location for each data model's respective logic. Our implementation of services also allowed for easier testing each of the data models.

3.2.2 Controller

Our implementation of controllers allows for our project to separate all of our API functions from the rest of our project. In this directory we also establish a generic "Data" object that represents any data that a site device can provide. This object contains important information that we need to work with, and ultimately store within our application's database.

There are two controllers used in our Spring Boot server. The first being the Site Controller that implements REST API endpoints for storing data from site sources into the Cassandra database. This controller also provides endpoints for our frontend application to access data that gives context to what is stored in the database (i.e. a measurements label, site location, etc.). The other controller implemented in our project is the Export Controller. This controller hosts a single endpoint that is used to parse and send data from our database when called. This endpoint sends this data in a form of a comma separated value file that is saved to the frontend user's device.

SiteController	Endpoint	Function	Parameters
	POST /data/store	Stores data into measurement repository, optionally store to measurement archive repository	1: data 2: archive
	POST /data/site/create	Creates a site with name “siteName”	1: String siteName
	GET /data/sites	Get all sites from the database	N/A

	GET /data/datasources	Get all data sources from site with “siteId”	1: UUID siteId
	GET /data/measurements	Get measurements of type “measurementId” after provided time “timestamp”	1: UUID measurementId 2: LocalDateTime timestamp
ExportController	Endpoint	Function	Parameters
	GET /export	Exports “date” month worth of data given datasource “id” and “name”	1: UUID id 2: String name 3: LocalDate date

(Table 4. REST API Endpoints)

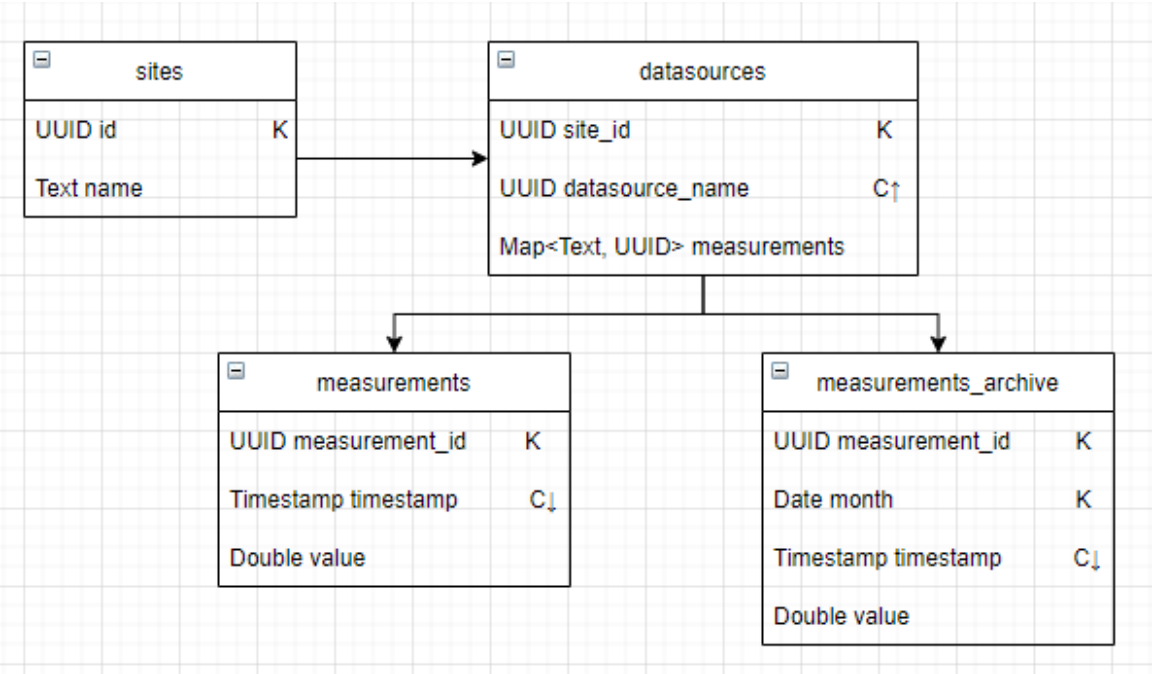
3.2.3 Websocket

The application utilizes a websocket as a way for data to be exchanged without requests originating from the frontend. This allows for frontend users to subscribe to a channel and whenever data is sent from the server to that channel all subscribed users receive that data. This is useful in scenarios like our live data display because data can be sent over the websocket as soon as an entry is added to the database. This is also particularly useful because our frontend client does not have knowledge of newly available data in the database, thus would not know when to request data. Another useful aspect of our websocket implementation is that it supports communication over a dynamic amount of concurrent channels. This is necessary to our applications functionality because of the need to support a dynamic number of sites. For each site storing data in our database there is a related websocket channel to support that site’s data transfer to the frontend.

The websocket was configured to use STOMP as a messaging protocol. This establishes a formal packet dictionary for our application to utilize and helps define the communication over the websocket channels. In practice each packet contains a JSON object pertaining to a datapoint that the frontend user will display on the live data display. Overall, this protocol allows for a concise and simple way for communication to occur over our websocket interface.

3.3 Database

As discussed in the design, the database we utilize for this project is Apache Cassandra. Our goal is the ability to keep up with a large number of frequent writes and Cassandra is excellent for this purpose. In our database we store information about each site, the different data sources that each site has, and the collected data from each data source. This can be seen in the design of our schema.



(Figure 2. Database Schema)

One important feature of this design to discuss is the measurements map that the datasources table stores. The keys of this map is the name of each measurement that a datasource stores and the value is a uuid used to identify the set of values that measurement has stored in the measurements table. This design choice allows for a number of different benefits. First, it is very easy to add another measurement or remove one as it does not require a schema update. And secondly it allows for better read and write performance as each set of data written to the measurement table is on its own partition and the data per write is only a few columns, rather than a single write with every measurement as a column.

Another detail to discuss is the two measurements tables: measurements and measurements_archive. Both these tables store the collected measurements and values for each datasource. The difference between them is that the measurements table stores all data collected while the measurements_archive table only stores a smaller subset of the data sampled from the total. After a configurable amount of time, data from the measurements table is deleted, while the data in the measurements_archive table persists indefinitely. This implementation decision was made to best handle the large amount of data that we are collecting. In a single day it is possible to collect well over a few million rows of data for just a single site. Across many months and the expansion to many sites, this large amount of data presents a storage problem. The key insight into this design is that the data granularity is much more important soon after it has been collected, while as time goes on it is sufficient to only keep a subset of the data to see larger trends. Therefore with this design we are able to keep a large amount of data soon after collection, but then store the trends data forever as this is a smaller subset.

Finally we discuss the configuration of Cassandra in regards to its distributed system. For our current implementation we rely on a single Cassandra node and do not make use of its distributed capabilities. Therefore, in our design we choose to use the SimpleStrategy for replication, with a

replication factor of 1. However, for future scaling of the project it will be necessary to make use of these features in order to maintain fault tolerance and reliability. In this eventuality, we then recommend to use the NetworkTopologyStrategy with a replication factor of 3. This then allows reliability and fault tolerance because the system will then store a total of 3 copies of all data across different nodes, and thus if nodes fail the data is less likely to be lost as it is replicated elsewhere.

3.4 Data Collectors

To collect data from a site, we implement python scripts for each datasource. While each datasource has a different api access method, the general format of these scripts is consistent. First the scripts read a config.yaml file. This file contains information about the api address, server address to send the information too, values to collect and collection rate among others. Then utilizing the twisted library to schedule the frequency, we call the datasource api to collect the data, and then send the data to our server to be processed. By having each datasource be collected by a different python script, this gives us the flexibility to easily collect data from new datasources. Simply create a script following the template design and then run it.

3.4.1 Tesla Powerwall

In order to access the local api the Tesla Powerwall provides, we utilized the 'tesla_powerall' library (https://github.com/jrester/tesla_powerwall). This library provided a robust set of tools to access the data measurements collected by the device.

The Tesla Powerwall is a fully-integrated AC battery system for residential or light commercial use. The rechargeable battery pack provides energy storage for solar self-consumption. The data collected from the powerwall includes the total energy, usable energy, max continuons real power, peak real power, apparent power max, apparent power peak, maximum supply fault current, and maximum output fault current.

The Tesla Powerwall divides its data into a set of different meters as well as general values for the entire device. The table below shows the different meters and their respective values that the system collects.

General	**Battery Meter**	**Load Meter**	**Site Meter**	**Solar Meter**
charge	instant_power	instant_power	instant_power	instant_power
	frequency	frequency	frequency	frequency
	energy_imported	energy_imported	energy_imported	energy_imported
	energy_exported	energy_exported	energy_exported	energy_exported
	average_voltage	average_voltage	average_voltage	average_voltage
	instant_total_current	instant_total_current	instant_total_current	instant_total_current

(Table 5. Tesla Powerwall Values)

3.4.2 Dranetz

To access the data from the Dranetz device, we directly made api calls to an exposed endpoint using python's built in request library. While this device's endpoints were undocumented, through our investigation of the source files of the local webpage the device provides, we were able to discover this api and add documentation for future reference.

The Dranetz meter measures power quality and records power quality events during site operation. The meter can capture fast transients which helps with solving instability that could occur during the operation of the microgrid. The data collected from the Dranetz includes line to line and line to neutral voltage, frequency, real power, reactive power, apparent power, and current.

The Dranetz device lists its data into a set of key value pairs for each value. The key is the internal register id it uses to identify the value. The following table encompasses these key value pairs for the data we collect.

Register Id	Value
0	volts_rms_a_to_neutral
2	volts_rms_b_to_neutral
4	volts_rms_c_to_neutral
6	volts_rms_d_to_neutral
8	volts_rms_a_to_b
10	volts_rms_b_to_c
12	volts_rms_c_to_a
14	volts_dc_a_to_neutral
16	volts_dc_b_to_neutral
18	volts_dc_c_to_neutral
20	volts_dc_d_to_ground
112	volts_deg_a_to_neutral
114	volts_deg_b_to_neutral
116	volts_deg_c_to_neutral
118	volts_deg_d_to_ground
120	volts_deg_a_to_b

122	volts_deg_b_to_c
124	volts_deg_c_to_a
144	frequency
1000	amps_rms_a
1002	amps_rms_b
1004	amps_rms_c
1006	amps_rms_d
1008	amps_dc_a
1010	amps_dc_b
1012	amps_dc_c
1014	amps_dc_d
1064	amps_deg_a
1066	amps_deg_b
1068	amps_deg_c
1070	amps_deg_d
2000	real_power_a
2002	real_power_b
2004	real_power_c
2006	real_power_d
2028	reactive_power_a
2030	reactive_power_b
2032	reactive_power_c
2034	reactive_power_d
2038	apparent_power_a
2040	apparent_power_b
2042	apparent_power_c
2044	apparent_power_d

4000	real_power_total
4002	react_power_total
4004	apparent_power_total

(Table 6. Dranetz Values)

3.5 Docker

By running our application on docker containers, this gave us great flexibility in locally running our application and in future deployment. For each component of our backend, we created a docker container for it to run on. Following is a list of each component of our backend and the DockerHub image we build our container on.

- Cassandra Database - cassandra:4.0.1
- SpringBoot Server - openjdk:8-jdk-alpine
- Data collection scripts - python:3.7-alpine

To combine and simplify the creation of the application across these different containers, we created a docker-compose.yaml that contained the full definition of backend container structure. We create a volume to create permanent storage for the data in our database which we attach to our Cassandra container. Then, using conditions we stand up each container in order of dependency, first standing up the Cassandra container, then the SpringBoot container, and finally the data collection script containers.

4 Testing

4.1 Unit Testing

A vital aspect of the testing process for our project is the unit tests that we created for the core functions in our application. This allowed us to easily check if individual parts of the application were working correctly after making updates to the codebase.

4.1.1 Backend Testing

Backend unit testing is being done using the JUnit framework. We deploy testing on each individual service component in the backend. This procedure is to ensure that each individual data model present in the application has functioning logic and checks for expected outputs.

4.1.2 Frontend Testing

We used the Jest testing framework to write our unit tests for our mobile application. Our strategy for testing pieces of the mobile application was to write tests that covered individual functions used by our components and screens. Jest also allowed us to mock basic functions and api requests used in order to best isolate the unit of code we were testing and allow us to control expected behavior.

4.2 Integration Testing

Another important step in our testing process was the integration tests to verify if the components of our application were interacting as expected.

4.2.1 Backend Testing

Integration testing the backend involves the Mockito framework to mock interactions with components and MockMVC to build web requests. Integration testing was used to test the web interface/API aspects of the project. These tests focus on endpoint response when making calls to our applications API, ensuring the correct methods are mapped correctly well checking to see the response body is to be expected.

4.2.2 Frontend Testing

For creating integration tests for our mobile application we used a combination of the Jest testing framework and the Enzyme library. Enzyme allowed us to write tests of a combination of components in a screen and interact with a mock render of a screen in order to check that the app was displaying what it should. This allowed us to simulate user interactions including taps and presses of the screen and make sure the app was exhibiting the proper behavior.

4.3 Acceptance Testing

After each individual feature was complete we made sure to involve our client in the acceptance process of said feature in order to guarantee that it met requirements. Because many features of the application built off one another, it was critical that our client had incremental updates and accepted these features once complete, instead of only seeing it a few times throughout the semester. Therefore, we made sure to demo completed features to our client and advisors during our weekly meetings to either gain feedback on the implementation or confirm a feature as accepted.

4.4 Results

Through our testing process we can be confident that our application met the requirements set by our client and that our application had at most a negligible amount of bugs.

An important result of our testing plan was the results of our database schema test. This test was for the space requirements of different schema formats for storing data. Our proposed schemas were storing our measurements in a map column, in a list column, or in another separate table for each measurement. To test this we created a database for each proposed schema and inserted 100,000 mock data readings containing 4 measurements per reading into each of the databases. The following table contains the results of these tests.

	Store in Map Column	Store in List Column	Store in Separate Table
Database Size	5.2 mb	5.4 mb	4.7 mb
Write Count	114488	114487	457948

(Table 7. Database Schema Test Results)

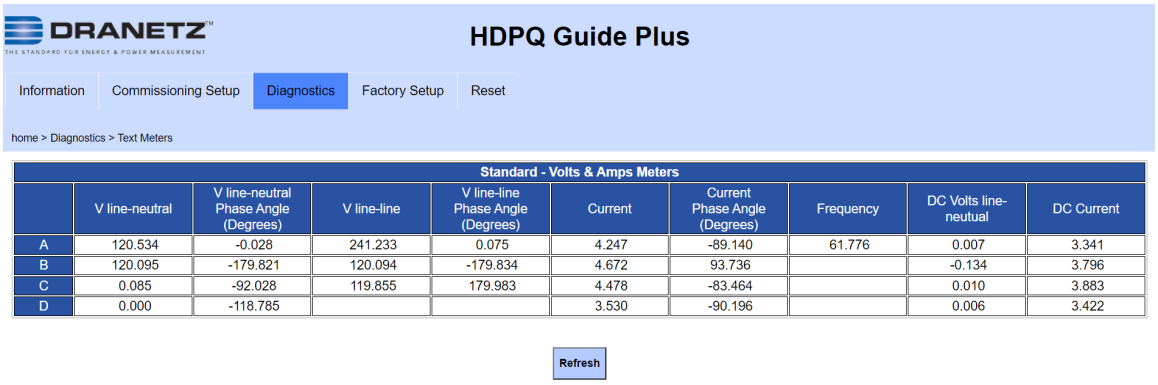
As can be seen in the results, using a map or list leads to very similar performance results. Using a separate table to store measurements resulted in a slight decrease in database size with the tradeoff of 4 times the number of writes required, in accordance with the number of measurements that were collected. Using the results gathered here as well as other considerations of the project design, it was ultimately decided by the clients and the team to go with the third schema design.

5 Related Products

To create our mobile application, we used the combination of both the Tesla app featured on iOS and Android and the Dranetz HDPQ IP address. The Tesla app gives users the ability to communicate and obtain information from their vehicles and products, but we are specifically focusing on the Powerwall, which stores and provides solar energy in case the grid goes down. From the Powerwall, users can monitor how much solar energy is being stored, used, and distributed to the grid. The images below are screenshots of what the Tesla app looks like.

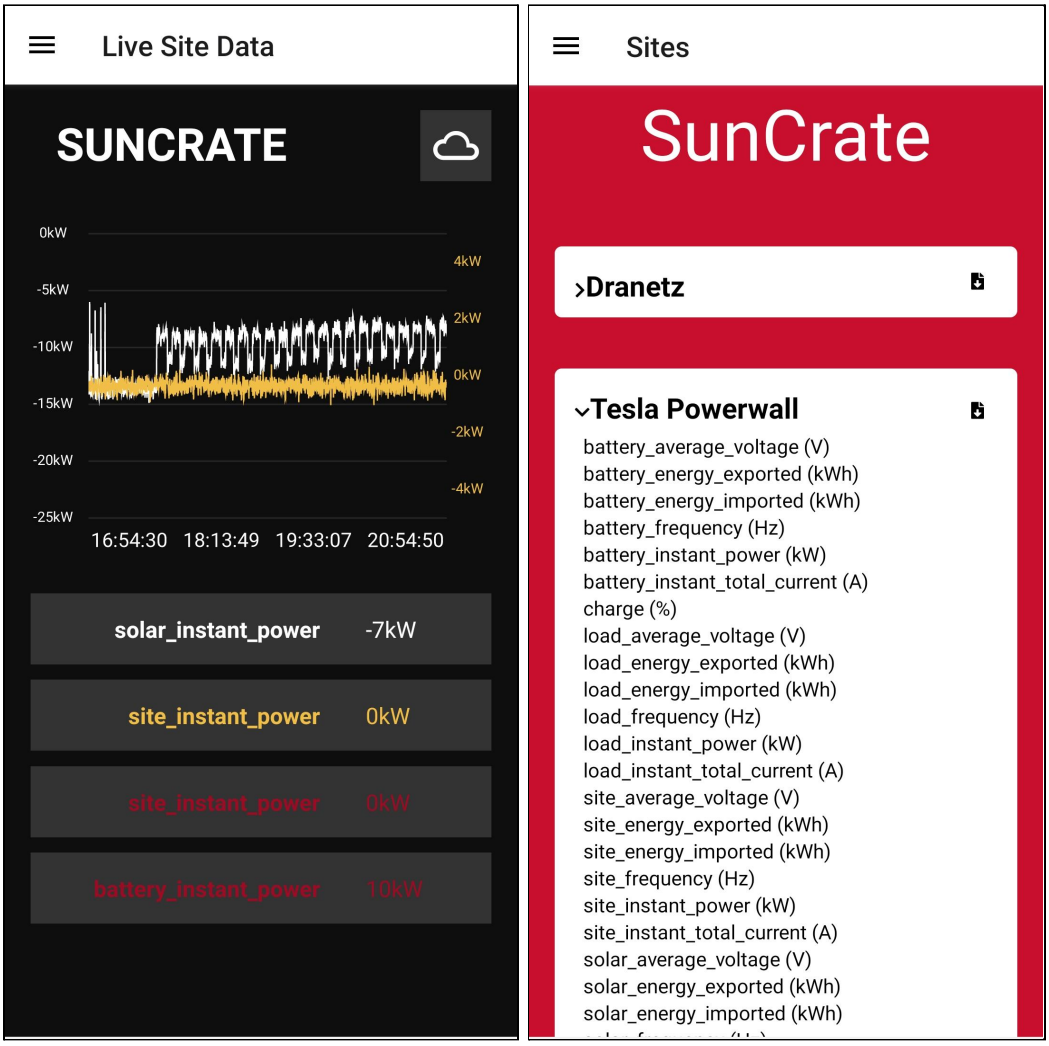
(Figure 3. Tesla App Pages)

As for the Dranetz HDPQ IP address, which is used to locate the device on the internet, users can monitor and record the power quality and energy demand from the four voltage and current channels. Below is a screenshot of the text meter page on the Dranetz HDPQ.



(Figure 4. Dranetz Text Meter Page)

Our mobile application is a similar but more advanced version of the Tesla app and the Dranetz HDPQ IP address because it incorporates all of the data from both devices into one single app, which can be displayed on different graphs. Users will no longer need to use both devices separately.



(Figure 5. Our Mobile App Pages)

6 References

"Data Replication." *Docs.datastax.com*, https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeReplication.html.

"Designing For Mobile." *Digital Design Standards*, digitaldesignstandards.com/standard-category/mobile/.

Fielding, Roy T., en Julian Reschke. "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content". Jun 2014. Web. Request for Comments.

Melnikov, Alexey, en Ian Fette. "The WebSocket Protocol". Des 2011. Web. Request for Comments.

Xuefen Fang, "Using a coding standard to improve program quality," Proceedings Second Asia-Pacific Conference on Quality Software, 2001, pp. 73-78, doi: 10.1109/APAQS.2001.990004.

Appendix

I Operation Manual

In this section, we provide a set of steps a user must follow in order to run our application. This is segmented into two major pieces. First the user will deploy the backend, which includes the server, database and data collection scripts. After that is complete, the user can then setup and run the mobile application.

I.a Deploying The Backend

I.A.1 Data Collection Configuration

To update the configuration of the Tesla Powerwall and Dranetz data collection scripts, open their config.py files located at 'Backend/tesla_powerwall/config.py' and 'Backend/dranetz/config.py' respectively. The following are descriptions of the different config options.

Tesla Powerwall

- frequency - how often to call the api, in seconds
- archive_frequency - how often to archive the data, in number of collections
- server
    - address - ip and port of the Tesla Powerwall
- datasource
    - address - ip and port of the spring server
    - site_id - uuid of the site
    - name - name of the datasource
    - password - Tesla Powerwall account password
    - user - Tesla Powerwall account user

Dranetz

- frequency - how often to call the api, in seconds
- archive_frequency - how often to archive the data, in number of collections
- server
    - address - ip and port of the Dranetz
- datasource
    - address - ip and port of the spring server
    - site_id - uuid of the site
    - name - name of the datasource
    - password - Dranetz account password
    - user - Dranetz account user
    - values - dictionary of registers and values to collect data e.g. pair '6': 'Vdg (RMS)' means register 6 has values for Vdg (RMS).

I.A.2 Running The Application

1. Install docker following the official guide here: https://docs.docker.com/get-docker/
2. In the project directory, navigate to 'Backend/microgrid_app/'
3. Run the command './mvnw clean install'
4. Navigate to 'Backend/docker/'
5. Run the command './docker compose up –build –detach'

I.A.3 Creating a Site

To create a site, use a browser or an api tool such as Postman to send the following post request to the following url: 'server_address:8080/data/site/create?name=SiteName' with 'SiteName' replaced with the name of the site to be created. The request will return the created site including the id for the site. This id can be used to configure the data collection scripts to associate the data collected with the created site.

I.B Starting The Mobile Application

I.B.1 iOS/Android Setup

1. Install React Native by following the instructions for 'React Native CLI Quickstart' here: https://reactnative.dev/docs/environment-setup .
2. In the project directory, navigate to 'Frontend'.
3. If developing for iOS, navigate to 'ios' and run the command 'pod install', then return to the 'Frontend' folder. If developing for Android, skip to step 4.
4. Install the SockJS dependency by running the command 'npm install sockjs'.
5. Install the Stomp dependency by running the command 'npm install --save react-stomp'.
6. Install React Native Vector Icons by running the command 'npm install --save react-native-vector-icons'.
7. Run the command 'react-native-link' to automatically link to the react-native-vector-icon library.
8. Install React Native SVG Charts by running the command 'npm install --save react-native-svg-charts'.
9. Install React Native Maps by running the command 'npm install react-native-maps --save-exact'.
10. Run the command 'npm audit fix' to install any other dependencies that may be needed by the newly installed dependencies.
11. If developing for Android run the command 'npx react-native run-android' to start the app. Otherwise, use the command 'npx react-native run-ios' if developing for iOS.

I.C Using The Mobile Application

When first starting the application, the home screen will be displayed. To navigate anywhere from here, use the hamburger menu at the top left of the screen to bring up the available screens to navigate to. In order to begin navigating to the Data Export screen, select 'Data Export', and in order to begin navigating to the Live Site Data screen, select 'Live Site Data'.

When navigating to the Data Export screen, the desired site from which to export data must be selected.. Select the desired site to continue. With this site selected, multiple drop down menus will be presented, one for each data source in the crate. Each of these drop down menus may be pressed to see what data sets are included as a part of the selected data source, and they may be selected again to collapse this list. In order to download data from the database for the selected source, press the download button at the right side of the drop down menu. This will present the native month-year picker for the current device. Choose the month of data that is desired to be exported, and then click done to begin downloading a .csv file with that data in it. This file will be saved in the phone's file system so that it may be viewed on the phone, email, or shared.

When navigating to the Live Site Data screen, the method for selecting a site must first be selected. A site can either be selected by name which functions similarly to the Data Export screen, or a site may be selected by location. Choose the desired option to continue. If selecting a site by name, press the name of the desired site, and the Live Site Data screen for that site will appear. If selecting a site by location, a Google map will be presented containing markers on the map where every site is located. Select the pin of the desired site to view, and then select the name of that site which will pop up in order to be taken to the Live Site Data Screen.

Once at the Live Site Data screen, data may be added to the graph on the screen by selecting any of the buttons labeled beginning with 'Graph set' below the graph, or data may simply be monitored without graphing by selecting any of the buttons labeled beginning with 'Data set' below the graph. When pressing any of these buttons, a scrollable popup will be presented that lists all of the data sources present for the selected site. Choose one of these to begin monitoring that data in real time. If selecting a data set to be graphed, the past several hours of data for that set will automatically be loaded into the graph, and new data will continue to be added as it becomes available. If selecting a data set to be observed but not graphed, the most recent value for that data set will be listed next to its label below the graph.

From this screen, there are two more screens that may be navigated to which contain information related to the selected site. Select the icon with the Sun and a bolt of electricity to view the solar potential data for the selected site. This screen will list the anticipated monthly and yearly output for the site, and hourly forecasted output for the next 24 hours can be seen by scrolling down the screen.

Back on the Live Site Data screen, select the icon with a cloud to view weather data for the selected site. This will include the current weather information for the selected crate. Also included in this screen is a forecast for the next 7 days which can be seen by scrolling down.

I.d Running Tests

In this section, we will show how to run the applications unit and integration tests for the backend and frontend.

I.d.1 Backend Tests

1. Open the project director in a shell.
2. Navigate to 'Backend/microgrid_app/'
3. Run command './mvnw test -f pom.xml'

The tests will run and results will be printed in the shell.

I.d.2 Frontend Test

1. Open the project directory in a shell
2. Navigate to 'Frontend/'
3. Run command 'npx jest'

The tests will run and results will be printed in the shell.

II Alternate Versions of the Design

At the very beginning of the project, we considered many different backend, frontend, and database frameworks highlighted in section 3.4 Technology Considerations. During this initial brainstorming phase, we decided to choose Spring Boot as our backend, React Native as our frontend, and MySQL as our database. After presenting our prototype mobile app to the faculty panels and listening to their feedback, we realized that MySQL was not the best choice but rather Cassandra was an even better option. Cassandra was the better option because it had a more appropriate schema design. Cassandra stores the measurements by datasource in a map and list. In addition, it stores measurements in a table with datasource uuid as partition key. On the other hand, MySQL has a defined schema where all rows have the same structure.