NeoCharge: Software Architecture version 2.0

The Juicerz
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA, USA

December 5, 2019

Contents

Credits

Name	Date	Role	Version
Joshua Boe	December 5, 2019		2.0
Lauren Hibbs	December 5, 2019		2.0
Casey Daly	December 5, 2019		2.0
Pranathi Guntupalli	December 5, 2019		2.0
Hannah Kwan	December 5, 2019		2.0

Revision History

Name	Date	Reason for Changes	Version
Team	November 6, 2019	Initial version	1.0
Team	December 5, 2019	Completed version	2.0

1 Introduction

This document contains the details of the software architecture of the NeoCharge application. In the following sections, there is an overview of the architecture, then several diagrams which show the functionality and design of individual use cases. The architecture contains the communication between the application and the NeoCharge device, and the application with Amazon Web Services, as well as the technologies we will use in the application.

Further details of the use cases mentioned in this document can be referenced in the Software Requirements Specification.

2 Problem Description

NeoCharge is an electric car solution that allows customers to charge their cars without paying for an electrician. Currently the car charges as soon as it is plugged in, which is not ideal for the customer who will have to pay more and create more carbon emissions by charging their car in the middle of the day. The app will allow customers to schedule charges to optimize cost and carbon emissions, as well as view analytics which users have come to expect of IOT devices. Calculating carbon emissions is a stretch goal as the service which provides this information is not free. The app shall be simple to use, as the existing process of simply plugging the car into the NeoCharge device is very straightforwards. Customers will be able to use the app without needing to adhere to rigid charging schedules, which includes being able to charge the car as soon as it is plugged in or quickly rescheduling charging for the given day.

3 Solution

3.1 Overview

3.1.1 System Overview - Dataflow Diagram

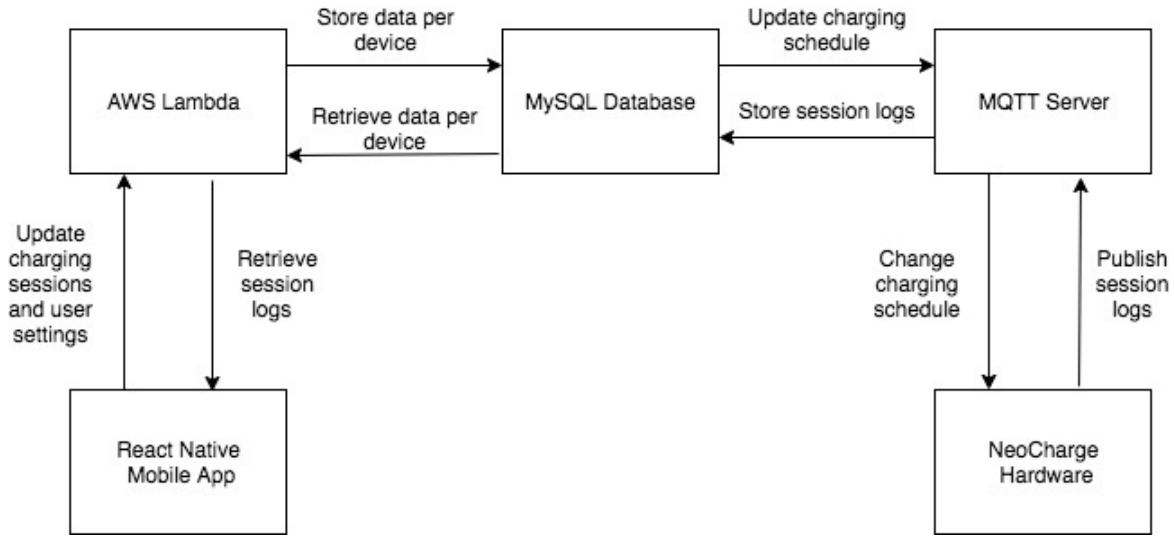

Figure 1: Dataflow Diagram

This dataflow diagram provides an overview of how data will be sent between different components of the system. The front end application servers primarily to allow the user to update charging schedules on the NeoCharge device. This data is transferred from the frontend to the MySQL database via an AWS Lambda "server" (see the section on AWS Lambda). The frontend also receives updated logs from the MySQL database. On the backend, the MQTT server manages listening to NeoCharge hardware and publishing the updates to the MySQL database, as well as writing update schedules to the devices.

3.2 Components

3.2.1 System Components - Deployment Diagram

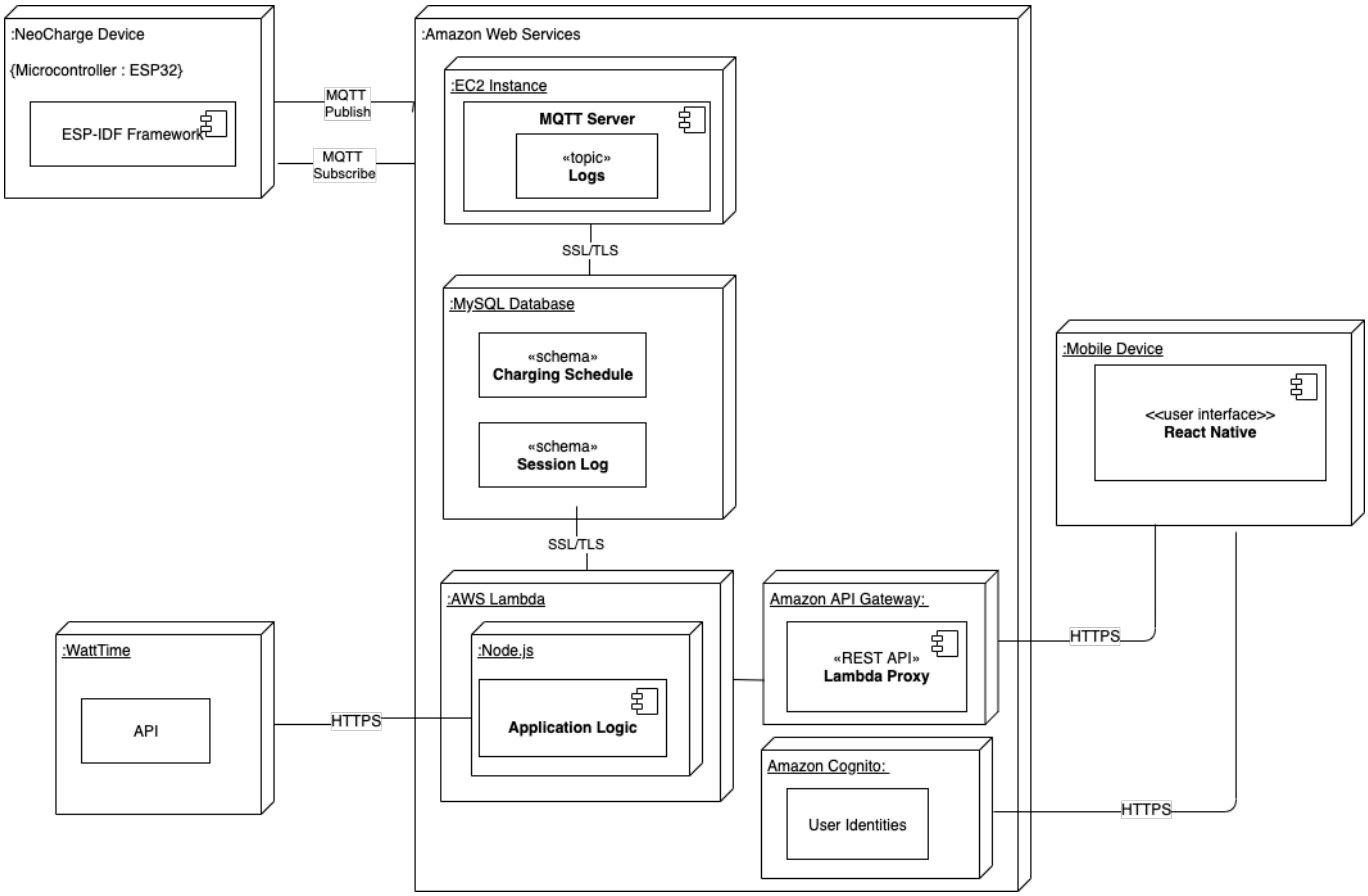

Figure 2: Deployment Diagram

This deployment diagram describes how the components of the system work together.

3.2.2 React Native Frontend

The React Native frontend will contain minimal business logic and serve exclusively as the user interface. The front end will make calls to AWS Lambda via http requests through the API Gateway, which will run functions to access information from the database and authenticate users. The front end integrates with the API Gateway via the Amplify library, which provides a simple way to make requests to REST endpoints. An important component of our front end is displaying analytics about charging history. We utilize a graphing library called VictoryNative to display this data to users.

3.2.3 AWS Cognito

Amazon Cognito is the AWS product that we plan to use in order to handle user authentication and access for the NeoCharge mobile application. The service saves and synchronizes end-user data and will manage the back-end infrastructure needed for handling new users. Cognito works by collecting the user's profile attributes into directories called user pools that a mobile app can then use to configure limited access to AWS resources. Data synchronizes with AWS when a device is online, allowing an end user to access the same information on another device. The data can also be saved locally to a SQLite database while offline before reconnecting.

3.2.4 AWS API Gateway

The API Gateway serves as a single point of access to the rest of the our Amazon services. It creates a rest layer where get, put, and post requests to each resource can in turn be connected to different lambda functions.

3.2.5 AWS Lambda

Our backend code will mostly be performing CRUD operations(Create, Read, Update Delete) on the database. It is best practice not to allow the front end of the application to communicate directly with the server, due to standard design patterns, performance issues, and security concerns. However, running an entire Node.js server as we had originally planned is more than we need. AWS Lambda is 'serverless' architecture from our perspective because we simply make calls to AWS Lambda functions where the server hosting is handled by the scenes. AWS Lambda is an appropriate solution because it handles database permissions, and abstracts backend logic away from the frontend of the app to a location in the cloud where it can be scaled appropriately.

3.2.6 MySQL Database

The SQL Database will be used to store the user's account information, charging sessions and logs per NeoCharge device. The exact schema for the data is still being decided. Logs will be written after each charging session and will be stored by device id and date. One charging session will be stored per device. The database is written to by both the mobile application and the NeoCharge device server. The server on the mobile application side will be able to update the charge schedule and user settings and be able to read device logs. The MQTT server on the NeoCharge device side will be able to read the charge schedule and update device logs.

3.2.7 MQTT Server

The MQTT Server is an AWS EC2 instance that conforms to the MQTT protocol. MQTT is a communication protocol primarily used for IOT devices. This model relies on

a publisher - subscriber architecture where each device is a publishes to the topic 'logs'. The MQTT server acts a listener for all these devices and translates the information received into messages to put the information to the SQL database.

3.3 Design

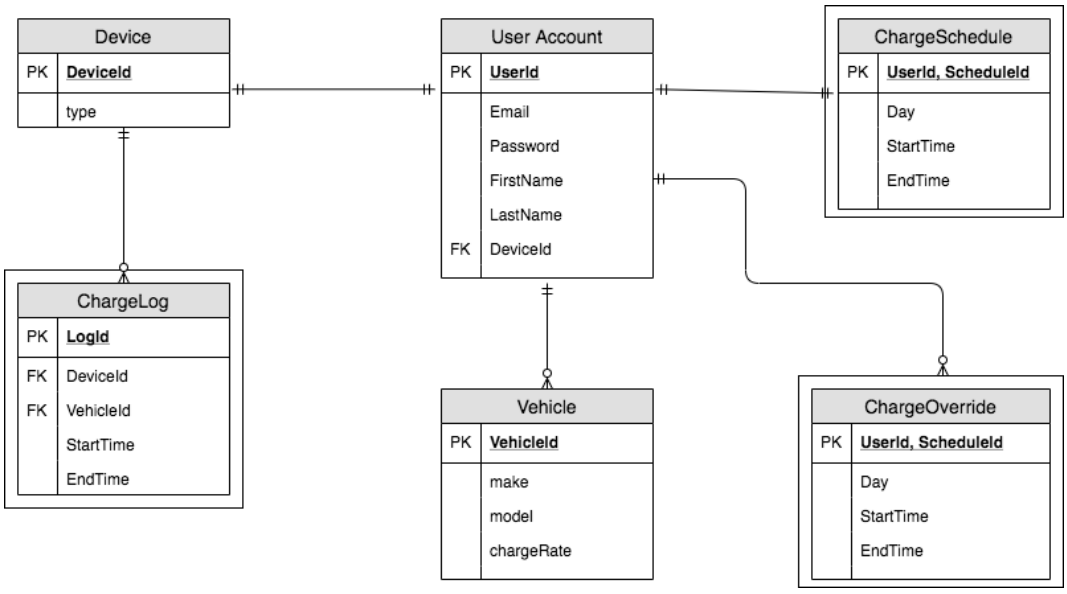

Figure 3: Entity-Relationship Diagram

The entity relationship diagram above shows the planned relationship between entities in the database. Device is its own table which serves primarily as a foreign key in other tables. Devices can have a type. UserIds and not DeviceIds are used in the case of ChargeSchedules because the user could change devices and these account level settings would still be transferred over. ChargeLogs on the other hand, are specific to the device they occurred on. A user can have a single charge schedule per device. This is represented in the database as different entries for the same userId with different ScheduleIds. It is necessary to have a ScheduleId as part of the primary key because there can be multiple start and end times for a charge on a single day. Day is a sufficient identifier for the time that the charge occurs because the schedule is on a weekly basis. It is important to note that the StartTime and EndTime in charge schedule is a Time, wheras the StartTime and EndTime in ChargeLog is a DateTime which includes both the time and the date. A user can have one to many vehicles. The vehicleId is a foreign key in the ChargeLogs table, as a user can view analytics that are different for each car. ChargeOverride is a concept developed in Usecase 6 in our SRS document. A ChargeOverride is separate than a ChargeSchdule in that it occurs one time and overrides the time for that day in the ChargeSchedule. This entry would be deleted from the table after it occurs.

3.3.1 Use Case Diagram

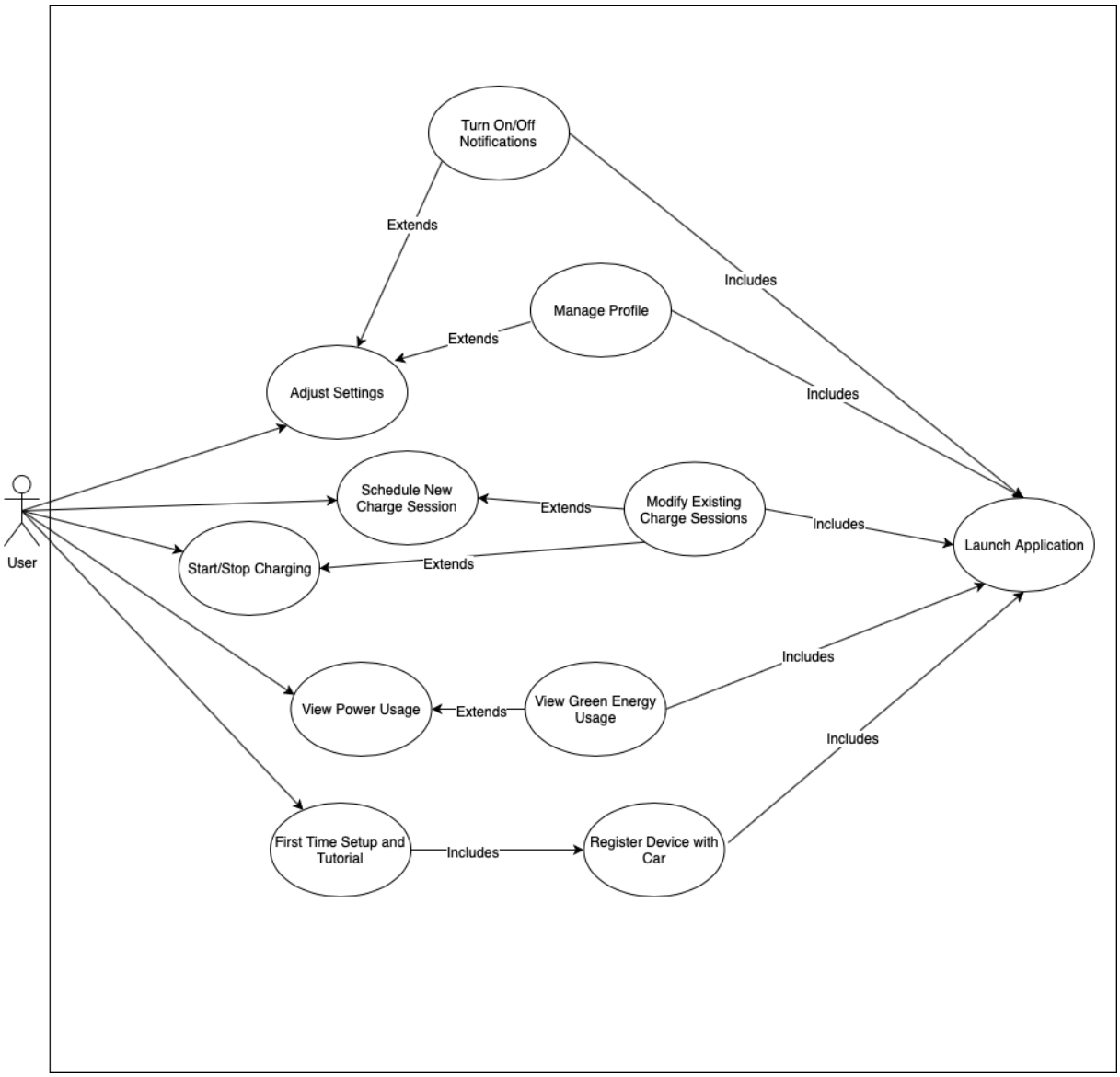

Figure 4: Use Case

This diagram is an overview of all the features the user can utilize when using the NeoCharge Application.One main part of this use case is the verification of information inputted by the user. It clarifies the main actions a user can perform while interacting with our application. The user will be able to adjust settings, schedule charge sessions, start and stop charging, view power usage, and view a first time setup and tutorial.

3.3.2 First Time Setup - Activity Diagram

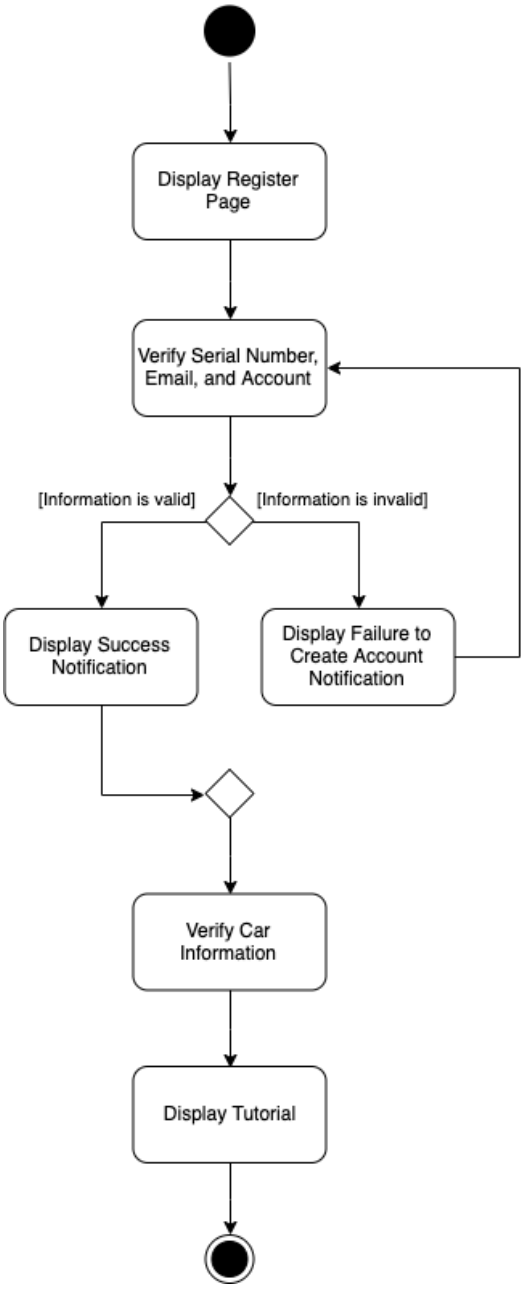

Figure 5: First Time Setup

This diagram describes the use case of a first time setup on the application. The user will enter the device's serial number along with their account information. First it is verified that the serial number is not already in use. If it is not the device is registered to that user. Once the device is registered to that user, the user should have access to only that

device's logs. After this information is entered the user will be taken to a tutorial. We are considering additional security measures when setting up the device so that devices cannot be registered to users other than the device owner.

3.3.3 Schedule a Charge Session - Activity Diagram

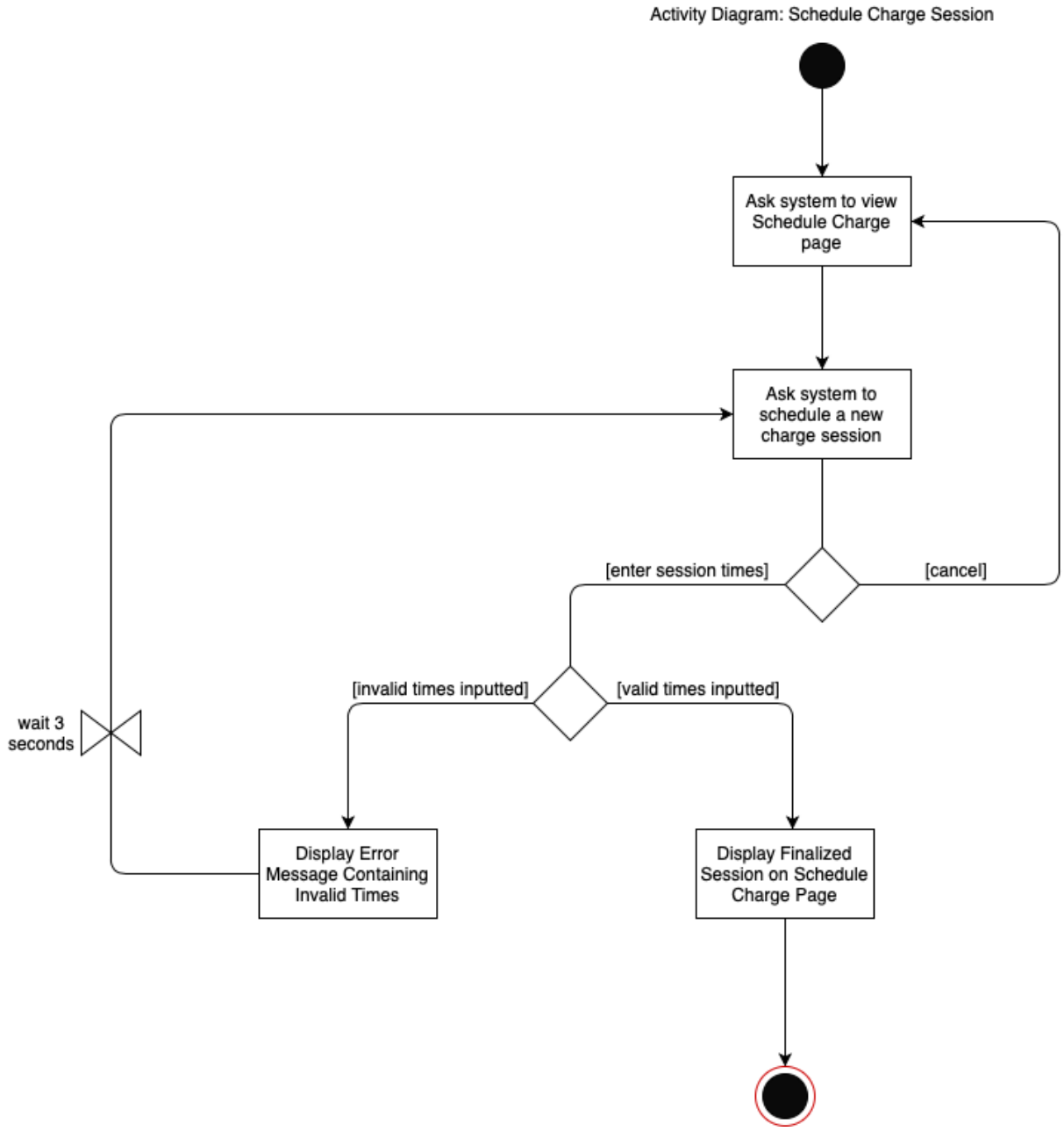

Figure 6: Schedule a Charge Session

This diagram describes the use case of a user manually scheduling a charge session. A user accesses the NeoCharge application from their mobile device. The user then navigates to the Schedule Charge page in order to schedule a charge session by inputting start and end times and the days of the week the charge should take place. The Schedule Charge page will then update to display the newly scheduled session.

3.3.4 Receiving Notifications - Activity Diagram

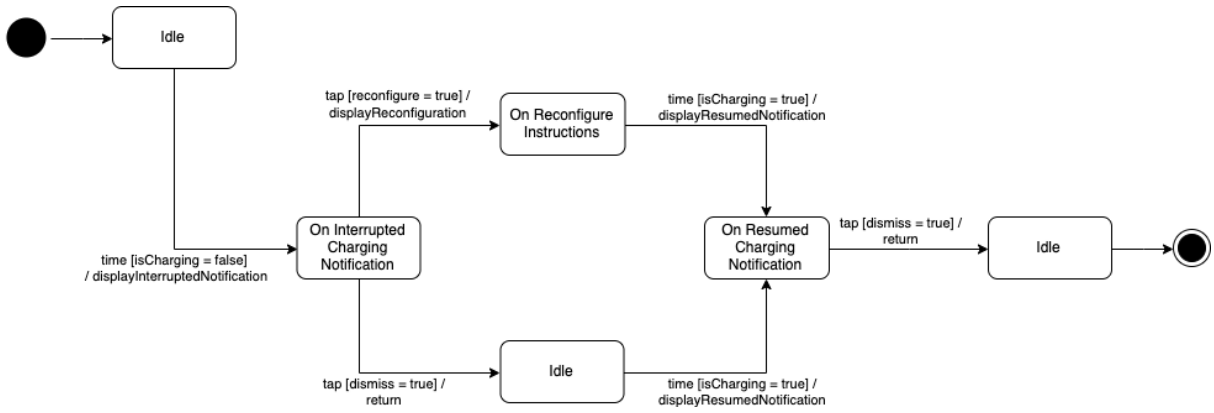

Figure 7: Receiving Notifications

This diagram shows the states of the system when receiving a notification. When the system receives the signal that the vehicle is not charging, a notification will pop up in the application. The user can either make sure their device is configured correctly, or dismiss the notification. When charging has resumed, the user will see a second notification and the application will return to an idle state.

3.3.5 Viewing Power Usage - Activity Diagram

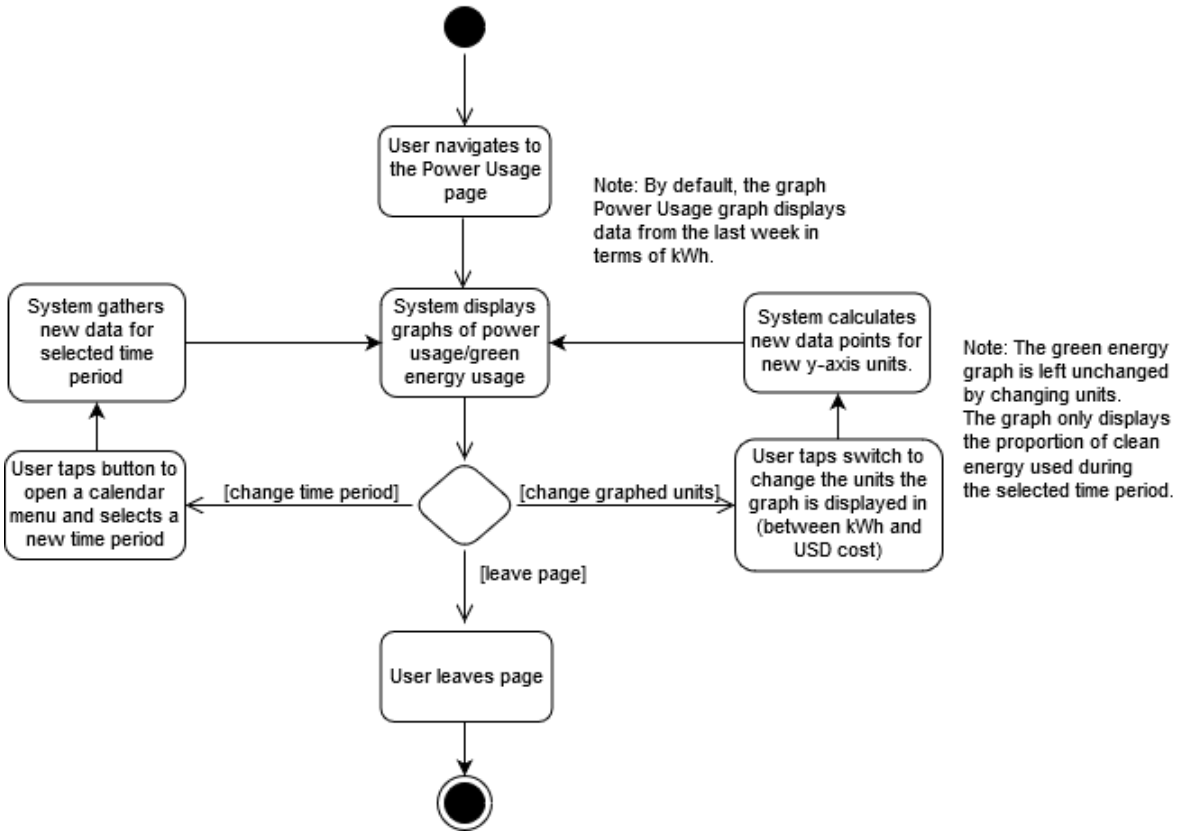

Figure 8: Viewing Power Usage

The above diagram showcases the behavior of the system when the user would like to view their past energy usage. When a user navigates to the "Power Usage" page, the system will display graphical representations of their power and clean energy uses for each day within a selected time period. By default, the power usage graph will display in units of kWh on the y-axis, and both graphs will display the last 7 days as their x-axis.

The user has 3 choices: change the time period, change the units, or leave the page.

1. The user can change the time period displayed in both graphs by tapping a "Change Time Period" button. The system will then display a calendar menu, allowing the user to select the desired time period they would like displayed. The system will then display new graphs that include the specified days.
2. The user can change the y-axis units of the power usage graph by tapping a "units" switch. The switch will alternate the units the graph displays between kWh or cost in USD. In response to a tap, the system will calculate new data points and display a new power usage graph that uses the new unit type.

3. At any point, the user can leave the Power Usage page by either exiting the app or navigating to a new page.

3.3.6 Adjust Settings - Sequence Diagram

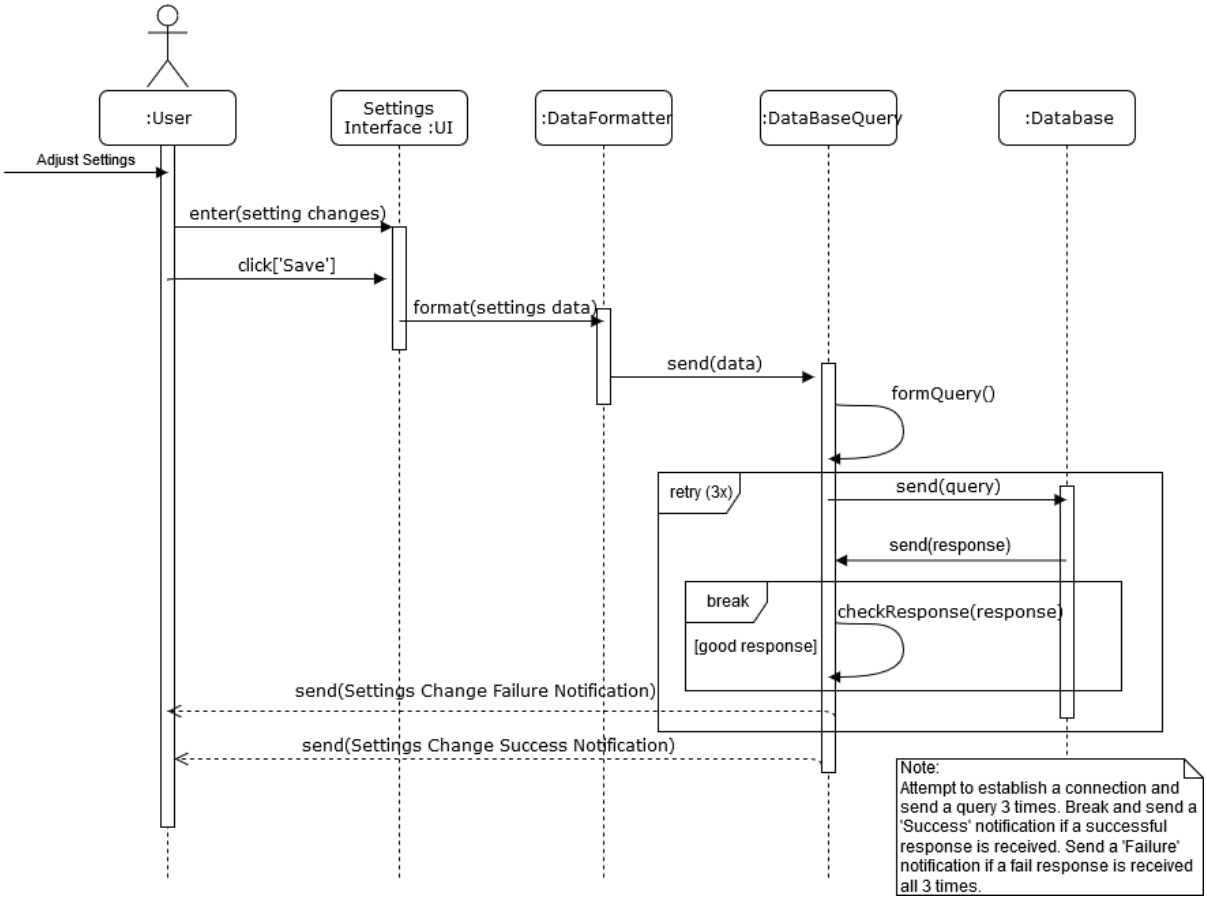

Figure 9: Adjust Settings

The above sequence diagram displays interactions between various objects and parts of the system when a user adjusts the settings within the app. The sequence begins with the user entering the changes that they would like to be made on the Settings page of the UI. After the user clicks "Save," the UI sends a snapshot of the changed settings to a DataFormatter that will format the data into an appropriate format to be sent to the database. The DataFormatter will then send the data to a DataBaseQuery object that will perform the task of sending the data to the database and awaiting a response. After receiving a response, the DataBaseQuery will check if the response signals either a query success or failure. If the query is a success, the user will receive a notification indicating that their settings changes were successful. On a failure, the query will attempt twice more to send the data to the database. If after 3 failures the data is still not able to reach

the database, the user will receive a notification indicating that their settings changes failed to be made and that they may try again later.

3.3.7 Class Diagram

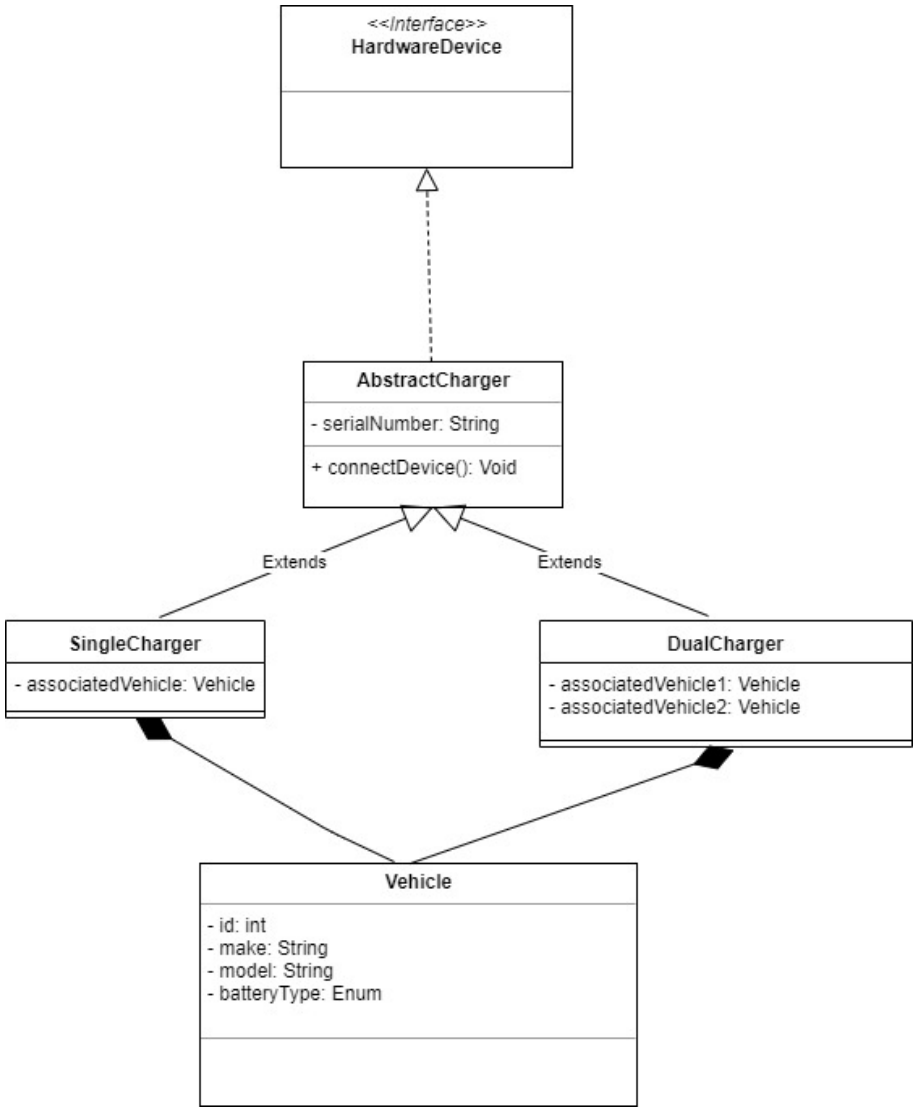

Figure 10: UML Class Diagram

The start of the Class Diagram is show in Figure 10 above. All classes representing charging hardware inherit from, at the highest level, the HardwareDevice interface, and then the concrete charger classes (SingleCharger and DualCharger) extend the AbstractCharger abstract class. This was left broad because all we know of at this point is that we are going to be dealing with charging hardware, but it could very well be possible

that our application will have to interface with new hardware in the future as requirements change. As good object oriented software should be designed, this hierarchy is left open to extension and closed to modification. For example, if a new piece of hardware is released by NeoCharge that is a supplementary piece of hardware, and not necessarily a charger, it can implement HardwareDevice. If a new charger comes out, it can extend AbstractCharger.

4 Test

The architecture will be tested on the python testing framework "pytest" through Travis CI. When pushes are made to the master repository in the team's Github, it will automatically trigger build jobs in Travis CI. The build status will be emailed to our developers or can be viewed on the Travis website.

Every developer will be responsible for writing basic unit tests as they relate to the code they have written. We will create a set of smoke tests that will be run automatically upon every build to ensure that no critical functionality has been broken with subsequent pushes. Upon the successful running of our smoke tests, we will run a set of sanity tests for each new feature or bug fix to make sure the correct functionality has been achieved.

One example of an acceptance test for our project draws from Use Case 1 of our Software Requirements Specification document. If a user tries to view power usage from a time period in the future, they will not be allowed to do so, because power usage for this time period is not possible.

A Glossary

AWS (Amazon Web Services) - a cloud computing platform provided by Amazon that offers a wide array of cloud services for creating scalable applications.

AWS Lambda - A serverless computing service provided by Amazon Web Services. It runs on pieces of code (Lambda functions) to stateless containers. Stateless means that every time a Lambda function is triggered by an event it is invoked in a completely new environment.

MQTT - A machine to machine (M2M) data transfer protocol. This protocol allows edge of network devices to publish to a broker. Clients connect to this broker which then mediates communication between the two devices.

HTTP - A communications protocol used to connect to Web servers on the Internet or on a local network (intranet). The primary function of an HTTP request is to establish a connection with the server and send HTML pages back to the user's browser.
SSL/TLS - TLS stands for Transport Layer Security. It is a web security protocol for communicating data. It is commonly used between web servers and other times security is important.

IOT Device - A piece of hardware with a sensor that transmits data from one place to another over the Internet. Types of IOT devices include wireless sensors, software, and computer devices.

B Issues List

1. The team has yet to make a concrete decision on the architecture behind push notifications. Most likely this will be integrated with the backend MQTT server and will utilize a third party free cross platform push notification service. This isn't included in our deployment diagrams as it was only recently brought to our attention that push notifications require additional architecture outside of what we have designed.

2. We have recently received substantial feedback on the design of the horizontal prototypes and the main features and use cases of the app. The NeoCharge customers are planning to conduct a user survey to get a better idea of what users want, but unfortunately this information is coming after we have already developed our use cases and user personas. Some changes we have discussed, such as to the user charging schedule, will change our representation of data in the backend and possibly more of our architecture. Additionally, we are waiting on further horizontal prototypes from a NeoCharge designer who we have been unable to get feedback from so far.

3. While Travis CI is free for open source projects, for private projects such as hours, the team collectively only has 100 builds. It was brought to our attention that there may be a cheaper option for continuous integration. Circle CI is one example of this. Depending on if our customer willing to pay for Travis CI, we will investigate other free continuous integration services.