Design Manual

Word Count: 7445

Cale Clark (H00296909)

Kyle Dick (H00301592)

Andrew McBain (H00317910)

Jake Paterson (H00323303)

Scott Valentine (H00301415)

Alexander Wickham (H00324206)

Contents

System Development Context 5

- Functional Requirements 5
- Non-Functional Requirements 7

Development Techniques 10

Frontend Components 11

- Overall Structure 11
- Tailwind CSS 11
- Font Awesome Icons 11
- Reusable Components 11
- Interactive Map 12
- Axios Requests 12

Backend Development Guide 14

- Overview 14
- Queue Backend 14
  - leave-queue 14
  - join-queue – 15
  - check-in 15
  - check-out 15
  - cancel-reservation 15
- Admin Backend API Calls 15
  - /set-permission-level 16
  - /get-admin-users 16
  - /clear-queue 16
  - /update-location 16
  - /update-charging-point 16
  - /delete-location 17
  - /delete-charging-point 17
  - /report 17
  - /validate-broken 17
  - /remove-report 17
  - /report-count 17
- Account Backend API Calls 18
  - /status 18

/login 18
/verify-user 18
/login-check 19
Email System 19
Email Host 19
Email Configuration 19
Database Management 21
Database Configuration 21
Database 21
Key Values Table: 22
Users Tables: 22
Location Table: 22
ChargingPoint Table: 22
Queue Table: 22
Report Table: 22
System Deployment 23
Background 23
Deployment Guide 24
Docker Configuration 25
Deployment Configuration Files 25
Storing Credentials 26
Sub-path Configuration 26
Changing Email Account 26
Resource Limits 26
Future Work and Known Issues 27
Future Work: Individual Charger View in Map API 27
Future Work: Three Strikes Priority System for Queue 27
Future Work: Extension of Session Timer for Users 27
Due to the time constraints this was not touched on, as it would likely require refresh tokens to be set up to ensure the login email link didn’t also remain valid for an extended period. 27
Future Work: Automate Error Reloads on Backend 27
Future Work: SMS Login Alternative 28
Future Work: Display Last Time A Charger Status Was Updated 28
Future Work: Refactored Queue System 28
Future Work: Continued User Testing 28
Issue: Users Are Given the Option To Join a Queue They Are Already In 28

Issues: Email Host Reliability. 28

System Development Context

Charge Checker is an application for mobile and desktop devices that provides electric vehicle users an interactive system to manage the use of physical charge points located at Heriot Watt University's Riccarton Campus. This document is a design manual written with the goal of providing information about the system as it is currently implemented. The information contained in this document will allow a future development team to continue the progress of the system and make improvements.

The context for this applications development was for use by electric vehicle users that visit Heriot-Watt University's Riccarton campus. A common problem for these electric vehicle users was difficulty in locating an available charge point on campus. It was often the case, as described by potential users, that the lack of an automated reporting system that would track the status of charge points would cause issues.

An overview of the information contained in this document is as follows:

- System Development Context
    - This section describes the context of the systems creation and the requirements that were defined for its implementation.
- Development Techniques
    - This section is a short description of the organisational techniques that were utilized by the team to reach the current production state.
- Frontend Components
    - This section covers the decisions that contributed to the final design of the user interface. This includes how the frontend can communicate within the backend.
- Backend Components
    - A description of backend architecture including a detailed list of API calls that interface with actions the user may take on the frontend to implement the key functionality of the system.
- Database Management
    - The system utilises a database to manage the data handled by the application.
- Deployment
    - The system is designed to be hosted remotely. The descriptions of how this was achieved including the creation of docker images can be found here.

Functional Requirements

The requirements defined because of the original requirement analysis stage have been included in this document as a guide to future developers. For each of these requirements their status in the final deployment has been classified under the following categories:

- Implemented Fully
- Implemented Partially
- Not Implemented
- Dropped From Scope

Functional Requirement #	Requirement Description	Priority	Status
FR1	A user must be able to access the website from on campus	M	Implemented Fully
FR2	A user should be able to scan a QR code to navigate the website	S	Implemented Fully
FR3	A user could be able to scan an RFID tag to navigate the website	C	Not Implemented
FR4	The user must be able to create an account using their email address	M	Implemented Fully
FR5	The user should be able to verify their account by email	S	Implemented Fully
FR6	The user must be able to delete their account	M	Implemented Fully
FR7	The user must be able to log in to the system	M	Implemented Fully
FR8	The user must be able to log out of the system	M	Implemented Fully
FR9	The user should be logged out after a certain amount of time	S	Implemented Partially
FR10	The system could remove unused accounts	C	Not Implemented
FR11	The user must be able to view the status of charging points on campus	M	Implemented Fully
FR12	The system should show a map of the campus and where charger points are located	S	Implemented Fully
FR13	The system won't get the live status from a charging point API	W	N/A
FR14	The user won't be able to connect the API of their electric vehicle to the system	W	N/A
FR15	The system must allow users	M	Implemented Fully

	to enter a queue for a specific charging location.		
FR16	The user must be able to view their queue position	M	Implemented Fully
FR17	The user must be able to leave a queue	M	Implemented Fully
FR18	The user should get a notification when a charging location that they are queuing for is available	S	Implemented Fully (email)
FR19	The user must be able to check into a specific charging place	M	Implemented Fully
FR20	The user must be able to check out of a specific charging place	M	Implemented Fully
FR21	The user should be able to report a charger that has the incorrect status (e.g. free when in use)	S	Implemented Fully
FR22	Administrator users should be able to edit the permission level of non-administrator users	S	Implemented Fully
FR23	The system must allow admin users to add, remove and update charging locations in the system	M	Implemented Fully
FR24	The system must allow admin users to add, remove and update individual chargers in the system	M	Implemented Fully
FR25	The system could allow admin users to add the location of each individual space to the map	C	Implemented Partially

Table 1: Functional Requirements

Non-Functional Requirements

Non-Functional Requirement #	Requirement Description	Priority	Requirement Met	Explanation

NFR1	The system must be usable on mobile.	M	Yes	The system uses Tailwind breakpoints to change the layout and scale for small screens.
NFR2	The system must be usable on desktop.	M	Yes	The system uses Tailwind breakpoints to change the layout and scale for larger screens.
NFR3	The system must meet the WCAG 2 AA accessibility guidelines.	M	Yes	The system uses a Tailwind colour palette to ensure colourblind accessibility
NFR4	The system could accommodate a range of languages.	C	No	There was not time to implement this
NFR5	The system should accommodate different screen sizes.	S	Yes	The system uses Tailwind's predefined breakpoints to make sure that UI elements scale correctly
NFR6	The system should be hosted on the university system instead of a cloud provider.	S	Partially	Although the main system is hosted on the MACS webserver, we are using a gmail account for our emailing system
NFR7	The system should be portable to allow for easy migration.	S	Yes	The main part of the system runs inside of Docker containers
NFR8	The system could use less than 1 gigabyte of RAM and less than 1 CPU (while under a realistic)	C	Yes	The 3 Docker containers have resource limits on them that ensure that they cannot use more than 500MB of RAM, or 1 CPU.
NFR9	The system should be configurable from a	S	No	The system also requires a config

	single Docker-Compose YAML file			file for the Nginx reverse proxy
NFR10	The user should be able to log into the system while giving only their email address.	S	Yes	The system uses password-less sign in.
NFR11	The system should have password-less login using email authorisation.	S	Yes	The system uses email only sign in
NFR12	The user must not be able to see the data of other users (e.g. email address).	M	Yes	Users of the site cannot see all other user emails for privacy reasons.

Table 2: Non-Functional Requirements

Development Techniques

This system was designed and developed in an Agile development process. The development team used the SCRUM methodology to split work into week-long sprints in which tasks were worked on.

The tasks for each of these sprints were created by reviewing initial requirements and create task specifications from them. These tasks were designated and organised mainly on a feature-by-feature basis.

These sprints were followed by weekly stand-up meetings where the team would address outstanding actions from the previous meeting, discuss the work carried out in the sprint and identified the tasks / actions to complete over the next sprint.

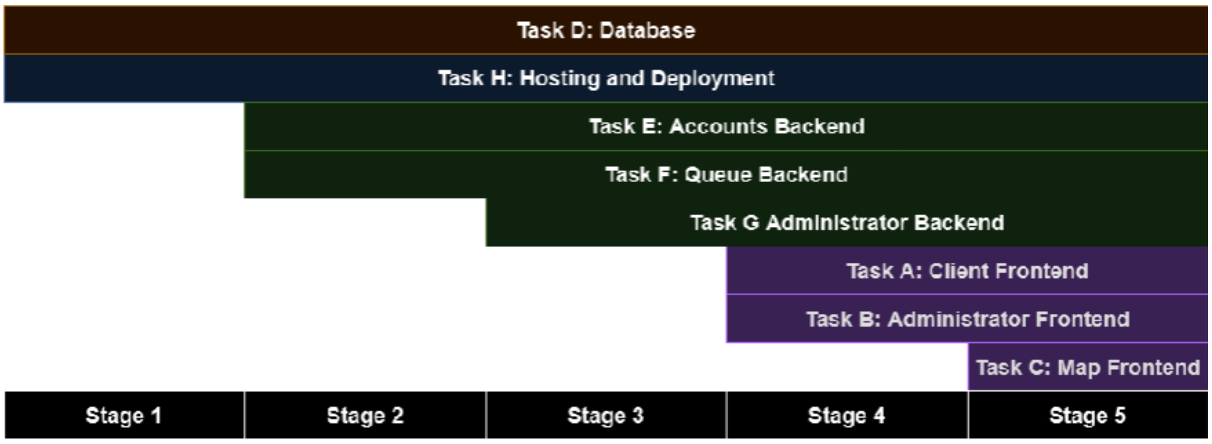


Figure 1: Sprint Structure from Initial Phases

Frontend Components

Overall Structure

The frontend was all written using React with JavaScript, in the root App.jsx file the routes are defined with each URL route given an associated page component.
The outer page components are all stored inside a folder called ‘pages’, each of these components correspond to a full page on the site. These page components make use of the smaller child components defined in the ‘components’ folder.
The ‘data’ folder is where the functions which perform the majority of CRUD requests to the backend are stored. The ‘public’ folder is where all image files are stored which are used throughout the site.
The last folder in the frontend is ‘dummyData’ in which Json’s replicating the prisma output from the database are stored for testing purposes.

Tailwind CSS

Throughout the front end we use tailwind CSS for all our styling, this meant that we could create a restricted colour palette in the `tailwind.config.js` file to ensure consistency throughout the site as no colours out with the predefined palette could be used. Defining this clear colour palette also helped ensure the site met the WCAG 2.0 at Level AA accessibility standards for colour blindness.
Also defined in the `tailwind.config.js` file were two themes for the navigation bar colours, to define the colour of different menu elements for each state (hover, selected & standard). This meant that the same navigation bar component could be used on admin and client site despite using different colours, by passing in “admin-theme” or “client-theme” into the class name.
Throughout the site we tried to make use of tailwinds predefined breakpoints to change the styling based on the screen size, this ensures that all objects scale correctly on different devices. Some features are also hidden completely on small screens to avoid making the user interface become cluttered.

Font Awesome Icons

Rather than use image files on the site for all icons, we decided to use Font Awesome icons as they allow icons to be embedded and treated as text which keeps styling consistent. Font Awesome also has better general performance than using images and had good browser compatibility.

Reusable Components

To make use of the modular nature of React, we created a many basic reusable components which ensures that code clean and that the user interface is consistent throughout.

- `Card.jsx` is a reusable component, which wraps code into a ‘card’ element which is a stylised white div with a shadow, these are used throughout the site and form the basis of the charger dashboard for the client side.
- `Button.jsx` is a reusable component for a prestyled button, this can be used to ensure all buttons are consistent throughout the site. This component takes a

background colour parameter which can have values ‘WHITE’, ‘GREEN’, ‘BLUE’ and ‘RED’ which ensures that all buttons of a given colour use the same text colour.

- `Modal.jsx` is a component which can be used to wrap content inside of a popup window with the surrounding background greyed out. This window takes a ‘setOpen’ parameter can be used to close the modal when the close buttons are clicked/ the user clicks off the modal. This modal window also expands to full screen on phone/small screens.
- `Navbar.jsx` is a component which can be reused for both admin and client as it is set up to take in a JSON (stored in the data folder), which contains all links along with their name and icon. As mentioned above it also makes use of tailwind themes to change the styling for client and admin sites.

Interactive Map

The interactive map was created using the open-source Leaflet js library and the OpenStreetMap geographical database. All map features were held inside the ‘Map.jsx’ component, at the start of this file a <Helmet> tag is used to append a necessary stylesheet and external script to the head of the document. The main map component is created using the <MapContainer> tag which takes in the initial center coordinates, zoom level, and controls information as parameters.
The <TileLayer> tag inside of this gets the map data from OpenStreetMap and the charge locations are mapped through and returned as <LocationMarker> components on the map. Inside the <LocationMarker> component the availability of chargers is used to choose which icon to display.

Axios Requests

The following files in the ‘data’ folder perform the backend API requests using the Axios library:

- ‘getAPIData.js’ was used as a reusable function for performing general **get** requests, it takes the end of the URL path for the request as a parameter.
- ‘joinQueue.js’ contains the functions for the:
  - **post** request for joining queue.
  - **patch** request for checking in to a reserved charger when at the front of the queue.
  - **patch** request for cancelling a reservation when the front of the queue.
  - **patch** request for checking out of a space you are currently in.
- ‘leaveQueue.js’ contains the **post** request for when a user selects to leave a queue.
- ‘login.js’ contains the:
  - **post** request for logging in
  - **get** request for checking if a user is already logged in when viewing login page (with redirect if they are)
  - **get** request for checking if a user if admin and outputting a Boolean result.
  - **patch** request for logging a user out by clearing their cookies and redirecting them to the login page.

- **patch** request for deleting an account and redirecting the user to the login page.
- ‘reportUpdate.js’ contains a **post** request which can take one of two paths, one for discarding a charger report and the other for validating the report.
- ‘updateCharger.js’ performs the **patch** request to update a charger status.

All functions above apart from the logging in function, required credentials to be passed in to show they are a logged in user and so the backend can retrieve the userID.

Backend Development Guide

Overview

The backend of the system has been organised to use routes for API calls to communicate with the frontend. The frontend makes requests which the backend that implements the key functionalities of our system. We have implemented this using express to handle our routing.

Our database interactions have been handled by the prisma package which has helped to cut down on the raw SQL querying across our system. It has also streamlined the readability of our code on the whole and created an easier database interaction format.

A basis of key functionalities across the system require the use of an email service to send email notifications to our users. We have used the nodemailer package to enable this across many backend functions.

Below is a more in depth description of the major functionalities implemented in the backend system of the product. Each section describes a key feature and lists the API calls used alongside a description of what specifically achieve.

Queue Backend

The Queue.js file implements the backend logic and API calls for the charger point queuing functionality of the product.

API Calls

/

Name	Required Data	Output	Type
/	User ID	ChargingPoint.chargingPointID, ChargingPoint.status, ChargingPoint.location, ChargingPoint.chargerType, List of QueueLengths	POST

Description: This call implements the functionality for returning data from the queues of each charging location. The function getQueues is used to return the charger point location details and current availability of each location on the system.

leave-queue

Name	Required Data	Output	Type
/leave-queue	User ID, List of available charge points		POST

Description: This call implements the functionality for removing a user from a charge point queue using the function leaveQueue. The function simply removes a valid userID value from the queue location list.

join-queue –

Name	Required Data	Output	Type
/join-queue	User ID, List of available charge points	An available charge point	POST

Description: This call implements the functionality for a user joining the queue for a specific charge point using the function joinQueue. This function first validates the existence of a valid userID value and ensures the user isn't already part of a queue or in a space. If the there is an available charger point at the user's location, the status of the user is updated to "PENDING" and the charge point is updated to "RESERVED" to indicate the charge point has been reserved by the. If not, then the user is added to the queue in a "WAITING" state to be notified for the next available charger point at that location.

check-in

Name	Required Data	Output	Type
/check-in	UserID, ChargingPointID		PATCH

Description: This call implements the functionality for a user to check into a specific charge point using the checkIn function. This function updates the chargePoint and user's status to "CHARGING" and removes them from the queue.

check-out

Name	Required Data	Output	Type
/check-out	UserID		PATCH

Description: This call implements the functionality for a user to check out of a charge point using the checkout function supported by the removeUserFromPoint and freeChargingPoint helper functions. removeUserFromPoint finds the User profile and updates the status to "IDLE" before calling freeChargingPoint. This function searches for the next user in the queue and updates the charger point as "RESERVED" for that next user or updates the charger point as "IDLE" if the queue is empty.

cancel-reservation

Name	Required Data	Output	Type
/cancel-reservation	UserID		PATCH

Description: This call implements the functionality for a user's reservation of a charge point space to be cancelled using the cancelReservation function and helper function freeChargingPoint. The function updates the user profile status to "WAITING" before calling freeChargerPoint to search for the next user in the queue and update the charger point as "RESERVED" for that next user or update the charger point as "IDLE" if the queue is empty.

Admin Backend API Calls

The Admin.js file implements the logic and API calls for the system specific administration actions.

/set-permission-level

Name	Required Data	Output	Type
/set-permission-level	Email, permissionLevel		PATCH

Description: This call implements the functionality to raise a user's permission level. The email and new permission level fields are taken from the request body before extracting the user record from the users table and changing the permissionLevel field to the new permission level value.

/get-admin-users

Name	Required Data	Output	Type
/get-admin-users		List of Admin users	GET

Description: This call implements the functionality to return all the users with admin permissions. The user's table is searched for all users with "ADMIN" or "SUPERADMIN" permission levels and added to a list which is returned by the call.

/clear-queue

Name	Required Data	Output	Type
/clear-queue	LocationID		DELETE

Description: This call implements the functionality to clear a queue for a specific charge point location. The locationID is used to ensure it is a valid location before the system retrieves all entries with that location and removes them from the queue table.

/update-location

Name	Required Data	Output	Type
/	chargingPoint, locationID, wattage, latitude, longitude, name, noChargers		PATCH

Description: This call implements the functionality to update the location details of a charging point location. First the fields for locationID, wattage, longitude and latitude are checked to have valid values. The locationID value is then used to search for the charger point in the location table and the data is updated with the information values passed in.

/update-charging-point

Name	Required Data	Output	Type
/update-charging-point	chargingPointID, status, locationID	Notification of new charger point	PATCH

Description: This call implements the functionality to update the details of a specific charger point. The chargerPointID and newLocationID is checked for validity before the chargingPoint record is update in the chargingPoint table.

/delete-location

Name	Required Data	Output	Type
/delete-location	locationID		DELETE

Description: This call implements the functionality to remove a charging point location from the system. First the locationID value is validated against the location table to ensure it exists and then the actual location record is deleted from the table.

/delete-charging-point

Name	Required Data	Output	Type
/delete-charging-point	chargingPointID		DELETE

Description: This call implements the functionality to remove a specific charging point from a charging location. The charging point is checked to exist within the chargingPoint table and then removed if valid.

/report

Name	Required Data	Output	Type
/report		List of Reported charging points	GET

Description: This call implements the functionality to retrieve a list of chargers that have been reported as broken. The getReport function extracts all the records in the report table and returns this information.

/validate-broken

Name		Required Data	Output	Type
/validate-broken		chargingPointID		PATCH

Description: This call implements the functionality to update a valid broken charger point. The validateBroken function uses the chargingPointID value to update the status of the chargingPoint record status to “BROKEN” and then removes the chargingPoint specific record in the report table.

/remove-report

Name	Required Data	Output	Type
/remove-report	reportID		PATCH

Description: This call implements the functionality to remove a charging point report. The function removeReport uses the provided reportID to delete the report record from the report table.

/report-count

Name	Required Data	Output	Type
/report-count			GET

Description: This call implements the functionality to get a count of reported charging points. The function getReportCount retrieves all the records from the report table and returns a count of these.

Account Backend API Calls

The Account.js file implements the logic and API calls for the system specific actions relating to account creation and login for users.

/status

Name	Required Data	Output	Type
/status	UserID	User status, chargingPointID, chargerLocationID, pendingStartTime, location	GET

Description: This is call implements the functionality for checking the current status of a user and what charger point location (if any) they are charging or about to charge at. This call uses the checkUserStatus function to first retrieve the status from the user table and then, if the status is "PENDING" or "CHARGING", return the details of the charging location and the start time of the charge.

/login

Name	Required Data	Output	Type
/login	Email		POST

Description: This call implements the functionality to both create a user account and allow a user to access an existing account on the system. This call uses the CheckUser, CreateUser and Login functions to do this.

The CheckUser function is used to check whether the email retrieved from the body of the request exists already within the user's table.

CreateUser is called if the email has not been found within the users table. It then creates a new record in the users table with the provided email address. This is to support seamless account creation and cut down on the steps a user needs to take to use the system effectively

The Login function is then called authenticate the user. This function fetches the UserID value from the users table using the provided email and then creates a signed Json Web Token(JWT) from this plus a JWT secret value. The function then creates a new email to send to the address of the user with a unique link using the JWT which they can click to continue signing in. This email is created using the Nodemailer package and uses the loginEmail template.

/verify-user

Name	Required Data	Output	Type

/verify-user	UserID		GET

Description: This call implements the functionality for the system to verify a user attempting to access the site from a login email. It verifies the JWT token received on request and, if a valid userID can be matched to a user in the users table, will redirect the user to main site to join the queue for a charge point at the location they accessed the site from.

/login-check

Name	Required Data	Output	Type
/login-check	UserID		GET

Description: This call implements the functionality for to check that user is in fact a registered user before requesting information. This is implemented by the middleware verifyLogin.js which checks the user login in is still a registered user in the users table and clears their cookies if not.

Email System

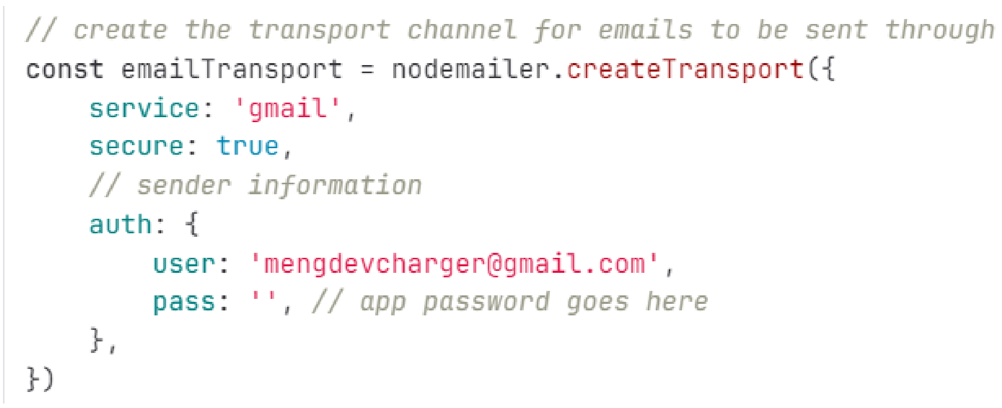
A source of significant concern for potential users of the system was based in a lack of available knowledge regarding the real-world status of charge points. An email-based notification system the solution to this problem. The advantages associated with this approach is that it can be integrated with the lightweight password-less sign in system to further minimize the required labour on the end-user.

Email Host

In the current implementation the source of system emails is a google account hosted at MengDevCharger@gmail.com. The motivations behind the decision to use this as the host provider of outgoing emails was due to the ease at which the email could be integrated within the node.js environment and concerns of security. This email host should only be used to produce outgoing emails to users because of API calls and is not intended to be the recipient of any emails itself. It is possible for the system to produce emails that may contain information that could identify individual users such as their name and email address. The email can benefit from an additional layer of security due to native two-factor authentication provided to all google accounts.

Email Configuration

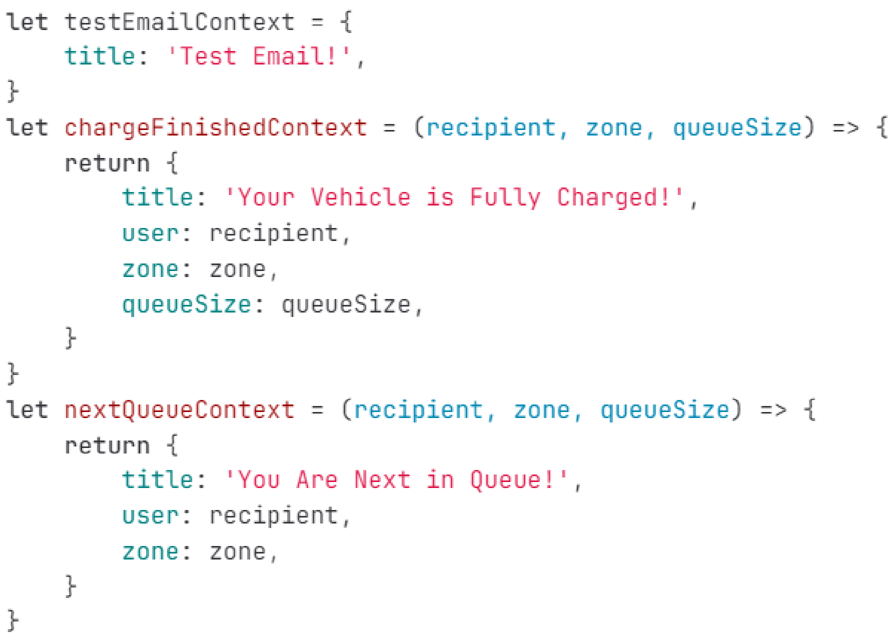
The email configuration utilises the nodeMailer module to manage the production and transmission of emails from the system to the users. Emails are sent through the Simple Mail Transport Protocol as is supported by nodeMailer which allows for support of all email providers that also utilise SMTP for transport.



```javascript
// create the transport channel for emails to be sent through
const emailTransport = nodemailer.createTransport({
    service: 'gmail',
    secure: true,
    // sender information
    auth: {
        user: 'mengdevcharger@gmail.com',
        pass: '', // app password goes here
    },
})
```

Figure 2: Email Transport Setup for Google Account (App Password Omitted)

Semantic templates for outgoing emails were achieved by utilising the handlebars JavaScript extension. This would allow for parameter passing to translate into the creating of DOM elements within an email. The data that each email should manage can be formatted and passed through the email transport as a context field.



```javascript
let testEmailContext = {
    title: 'Test Email!',
}
let chargeFinishedContext = (recipient, zone, queueSize) => {
    return {
        title: 'Your Vehicle is Fully Charged!',
        user: recipient,
        zone: zone,
        queueSize: queueSize,
    }
}
let nextQueueContext = (recipient, zone, queueSize) => {
    return {
        title: 'You Are Next in Queue!',
        user: recipient,
        zone: zone,
    }
}
```

Figure 3: Context Templates for Emails

Database Management

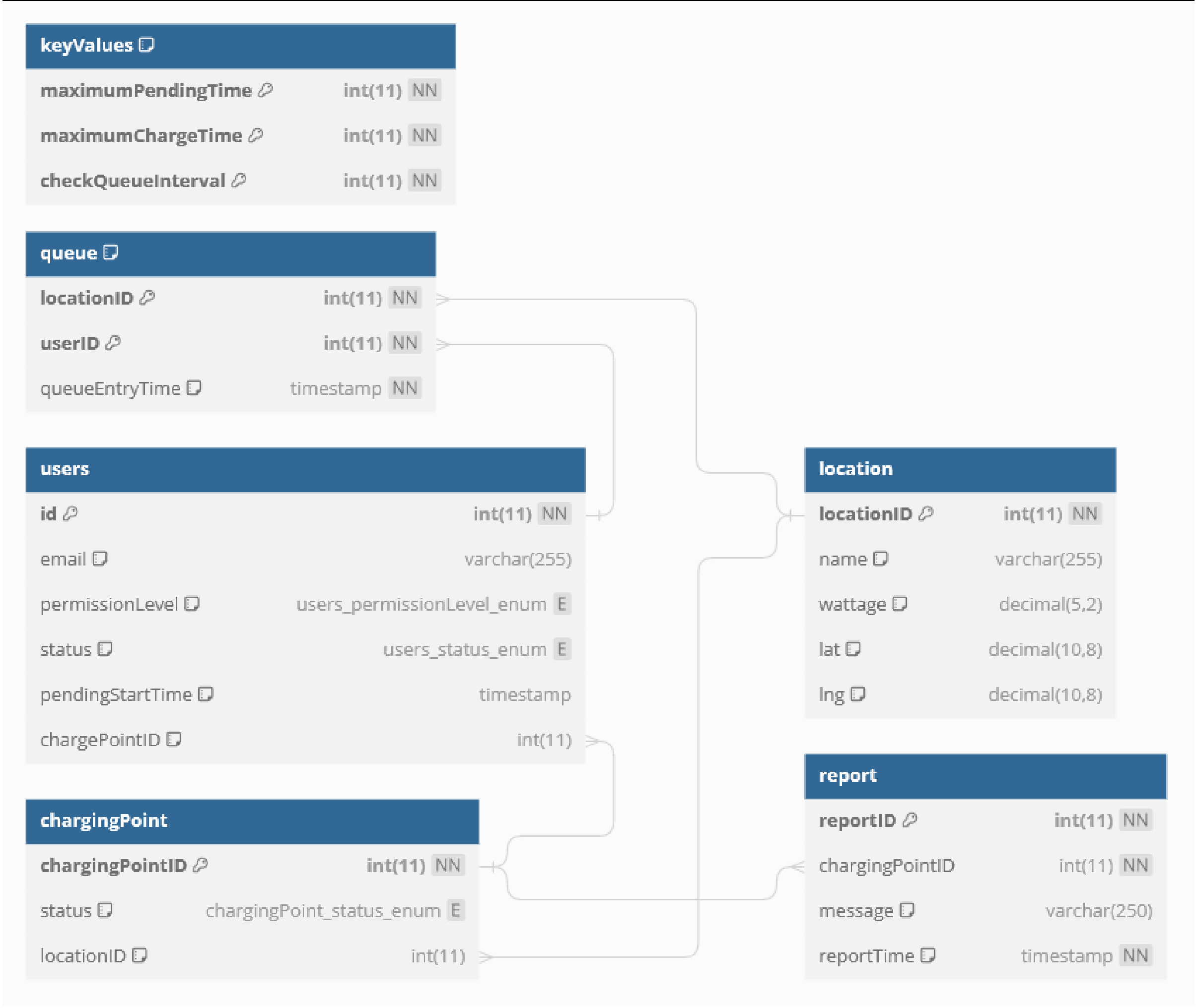
Database Configuration

The database can be created using the SQL file in the GitLab repository at /Implementation/database/hwcharging.sql. Please see the Database section above for details on the schema.

If you do not wish to use a managed database service, the system also supports using a self-hosted MariaDB instance. This can be done easily by simply adding a MariaDB Docker container to the docker-compose.yaml file. This is how the testing environment is configured to run, however for a production system it is recommended to use increased password security (the testing configuration has the password in plaintext in the docker-compose.yaml file) and to configure the database container to use a Docker volume to persist its data between restarts. A complete production configuration for a self-hosted database is out of scope for this guide.

Database

A diagram of the database schema is shown below.



keyValues		
maximumPendingTime	**int(11)**	NN
maximumChargeTime	**int(11)**	NN
checkQueueInterval	**int(11)**	NN

queue		
locationID	**int(11)**	NN
userID	**int(11)**	NN
queueEntryTime	timestamp	NN

users		
id	**int(11)**	NN
email	varchar(255)	
permissionLevel	users_permissionLevel_enum	E
status	users_status_enum	E
pendingStartTime	timestamp	
chargePointID	int(11)	

chargingPoint		
chargingPointID	**int(11)**	NN
status	chargingPoint_status_enum	E
locationID	int(11)	

location		
locationID	**int(11)**	NN
name	varchar(255)	
wattage	decimal(5,2)	
lat	decimal(10,8)	
lng	decimal(10,8)	

report		
reportID	**int(11)**	NN
chargingPointID	int(11)	NN
message	varchar(250)	
reportTime	timestamp	NN

Figure 4: System Database Schema

Key Values Table:

This table contains all the constant values that are needed throughout the backend functionality. This is done, as it will make it easier to alter the information without accessing the backend code and makes it easier change these key values. These key values are all intervals that are used to check how long someone has been checked in, how long someone has been inside the queue, and how long they have had access to the charger and haven't collected it. If the time has exceeded, then they are reset or removed from the queue to prevent the system does not get clogged.

Users Tables:

The user table contains all the key information for each user. The ID is an auto-incremented for each user that signs up for the website that is kept separate from the email so more of their data is kept secure while being a unique value. The email is the email address of the user that they have signed up with that is used for email verification. The user_permissionLevel is the enumerated value that will assign a value determining whether they are an admin. The status field is another enumerated value that will state what status the user is in, whether they are charging, idle or pending as shown in the enumerated charts. The pendingStartTime is the timestamp that is created when they enter a queue and the chargePointID is a charge point that is being currently used by the user if they are using it.

Location Table:

The location ID is an auto-incremented value that is used to specificality when referring to the location in other tables that is also the primary key. The name is a varchar, that contains the name of the location which is used whenever to the name of charger locations. The wattage is a decimal which stores the wattage that is used whenever the wattage is needed to show the users. The lat and lng are two decimals that are used for the map to show where the location is on the map.

ChargingPoint Table:

The chargingPointID is a auto-incremented value that is the primary key that is used when referring to the unquie id of a charger. The status is an enumerated value that is used to show what state the charger is. The location id is a foreign key from the location table to show which location it is referring to.

Queue Table:

The queue table acts as the link between the users table and the location table. The primary key is a compound key with the locationID and the userID forgien keys. The instance of someone queuing is created with the reference of the location they are queuing for. The queueStartTime is a timestamp that is created when the entry is created, representing when the user entered the queue.

Report Table:

The report table is where the reports are sent about broken chargers. First the reportID which is the primary auto-increment, the message that came with the report is stored in the message varchar field, the reportTime is the timestamp when the report was made. The chargingPointID is the foreign key to the charger table so make sure the report is referring to the charger that the report was made about.

A table of the enum types used in the database are shown below.

Enum name	Types
users_permissionLevel_enum	ADMIN USER SUPERADMIN
users_status_enum	WAITING CHARGING IDLE PENDING
chargingPoint_status_enum	IDLE BROKEN CHARGING RESERVED

Figure 5: Enum Types Used in Database

System Deployment

This section describes how the system can be deployed in a production environment. This section assumes that the reader has some knowledge of Docker and webservers.

Background

The system was designed to be run on the Heriot-Watt MACS webserver. This server runs a copy of Apache webserver, which is configured to redirect requests on the /hwcharging/ sub-path to our system. All these requests are redirected to http://127.0.0.1 (localhost), on port 8285 which is not exposed externally. We use a Dockerized Nginx reverse proxy to route these incoming requests to the two separate parts of our application: the NodeJS backend and the React frontend, each of which run in a container of their own. All requests to the /api/ sub-path are directed to the backend and everything else goes to the frontend. From the webserver, the backend of our system can communicate with a university hosted instance of MariaDB, which we use as our database.

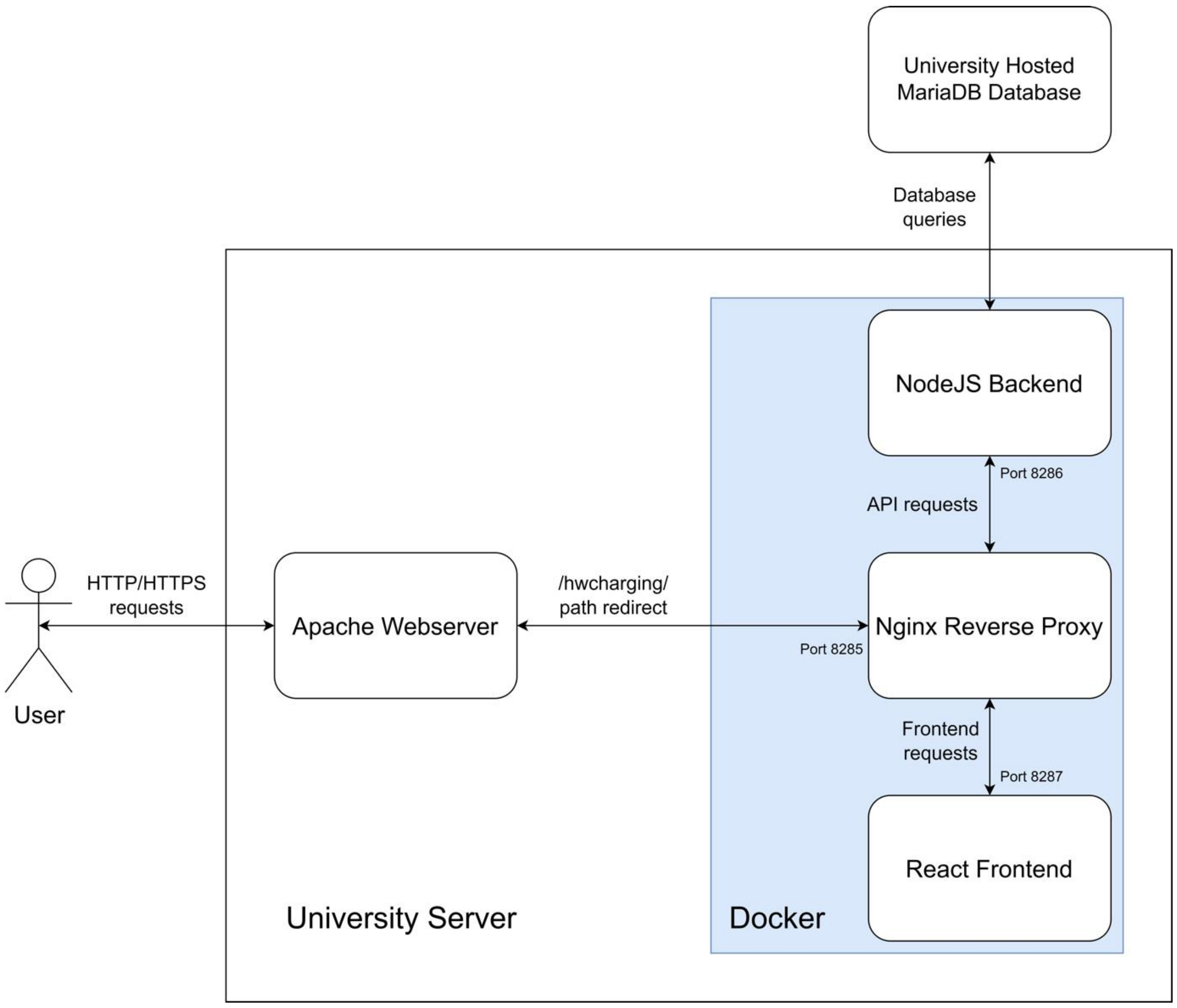


Figure 6: Deployment Architecture Diagram

Although the system is currently running in the above configuration, it is designed to be portable, so the Docker containers can be easily deployed on another webserver or a cloud environment. As long as the URLs are set correctly in the configuration, it should be theoretically possible to host the system on any platform running Docker. Before this project, the Apache webserver was already running on the MACS server, listening on the HTTP and HTTPS ports. This means that it was necessary for us to use it to redirect the requests on the /hwcharging/ sub-path to our system, since our application cannot also listen on the server's HTTP or HTTPS ports. When hosting on another platform, it is theoretically possible to have the Nginx reverse proxy listen directly on the HTTP and HTTPS ports, although it would need to be configured accordingly in the nginx-rproxy.conf file.

The system can also be run in a testing configuration, which is also Dockerized. Please see the Backend API Testing section of the test report.

Deployment Guide

The steps for deploying the system are outlined below.

1. Pull the "prod" branch of the GitLab repository

2. Ensure that the program is configured to be hosted on your desired sub-path (by default this is set to /hwcharging/). See the Sub-path Configuration section below for details.
3. Build the frontend Docker image by navigating to the /Implementation/client directory and running a Docker build command.
4. Build the backend Docker image by navigating to the /Implementation/server directory and running a Docker build command.
5. If you are building on a separate machine from the machine that you are going to host on, push the frontend and backend docker images to a container registry (e.g. Docker Hub or GitLab container registry) that can be accessed by the hosting machine. Additionally, the MACS webserver was having problems pulling the official Nginx image, so we opted to push a copy to our container registry as well to work around this. Then on the hosting machine, pull all the required images.
6. Configure the docker-compose.yaml and nginx-rproxy.conf files accordingly (see the Deployment Configuration Files section below for details).
7. Set up the database (see the Database Configuration section for details)
8. Set the required docker secrets. These are the database connection URL, JWT secret and the email password (see the Storing Credentials below for details)
9. Ensure that Docker is running in Swarm mode
10. Configure the docker-compose.yaml and nginx-rproxy.conf files accordingly (see Deployment Configuration Files below for details)
11. Deploy the stack using a "docker stack deploy" command. By default, the nginx-rproxy.conf is set up to route requests to a stack called "hw-charging". This will automatically pull all of the required images that are not already present on the machine.

Docker Configuration

On the webserver, Docker is set up to run in Swarm mode with only a single node. This enables certain useful features (e.g. Docker secrets). To read more about Docker Swarm, please see the official documentation. We also use Docker Compose files to simplify the configuration and deployment process significantly.

Deployment Configuration Files

The deployment configuration files are available on the deployment branch of the GitLab repository. This consists of two files:

1. docker-compose.yaml - This is the primary configuration file for deploying the system. This file sets the images that are used for each container, exposes ports, sets environment variables, mounts files and secrets into containers and sets resource limits for the system.
2. nginx-rproxy.conf - This file is mounted into the Nginx reverse proxy container to overwrite the default Nginx configuration. By default, this is set to receive requests from the Apache webserver and redirect them to the frontend and backend containers. The Apache webserver performs rewrites on the request URLs, so the Nginx reverse proxy expects incoming requests to have been rewritten in this manner.

Because of this, this file will most likely have to be adjusted for new hosting configurations. Please see the official documentation for details.

Storing Credentials
The credentials required by the backend of our system are the database connection URL (containing password), the JSON web token secret and the password for the email account used by NodeMailer. These are stored using Docker secrets, which are then mounted into the backend container. Docker must be running in Swarm mode to use Docker secrets, see the Docker Configuration section for details. To read more about creating and managing Docker secrets, please see the official documentation.

Sub-path Configuration
Since the system is hosted at https://www.macs.hw.ac.uk/hwcharging/, the system is configured to be hosted on the /hwcharging/ sub-path. This is "hardcoded" into certain sections of the system; however, it should be relatively simple to change this. A list of places in the source code where it is hardcoded are listed below, although these can also be found by using a search function (e.g. grep, or the visual studio code search tab):

- /Implementation/client/vite.config.js
- /Implementation/client/src/Env.js
- /Implementation/client/src/main.jsx
- /Implementation/client/src/data/getApiData.js
- /Implementation/client/src/data/login.js
- /Implementation/client/src/data/navbars.js

The deployment configuration files also must be changed in order to host the application on another sub-path, see the Deployment Configuration Files section for details.

Changing Email Account
The email account used by NodeMailer is set to be "MengDevChargers@Gmail.com" by default. The password that the backend uses to try to log in to this email can be set by using Docker Secrets when in production (or environment variable when in development mode). To change the email account, the email address must be changed in the source code.

Resource Limits
The system is also configured to limit the maximum CPU and memory usage of each of the three containers. This is currently set to:

- Nginx Container: 0.2 CPUs, 100MB RAM
- NodeJS Backend Container: 0.4 CPUs, 200MB RAM
- React Frontend Container: 0.4 CPUs, 200MB RAM

Future Work and Known Issues

At the end of the development there were serval know issues that remained. These issues have been left for future teams to work on to help improve the system. There are many future possibilities for this project that could be worked upon to improve its functionality.

Future Work: Individual Charger View in Map API

Description: In maps view admins would be able to draw out individual charger parking spaces to clearly show where chargers are in each location and the status of each.

A similar situation exists for the admins being able to add charging points without having to access the database directly as some of the functionality already, however given our scope we did not deem it as high a priority as the queue system as there already exists a map and there are no immediate plans for adding new charge point that we are aware of. It is also worth noting that we did add the ability for admin to label charge points as broken meaning the admins still have ability help show what is available to the users of the site.

Future Work: Three Strikes Priority System for Queue

Description: When a user fails to reach a parking space in the allotted time, when they get put back into a 'waiting' state they would also have a variable incremented for 'missed openings.' If a user has too many missed openings, they will be kicked from the queue, so they don't keep getting put to the top.

This was almost completely implemented however the main issue was implementing the incremental part of the program. For the future team that wishes to work on this they should make this a high priority as most of the functionality is completed. This would mean that you would not have someone who has signed up for the queue clogging up the system by keeping the top spaces up. This could be problematic if multiple users end up doing this so it should become a key priory for teams going forward.

Future Work: Extension of Session Timer for Users

Description: The log in cookies could be modified so the log in time could be extended if the user remains active, rather than exiting as soon as a given period has passed since logging on.

Due to the time constraints this was not touched on, as it would likely require refresh tokens to be set up to ensure the login email link didn't also remain valid for an extended period.

Future Work: Automate Error Reloads on Backend

Description: If tasks such as checking out a user from a space do not work, rather than sending them an error message saying to try again later (which if checking out they likely won't do), we could send it to a pub/sub server to keep retrying in the background

This wasn't touched upon as it does not specifically add features but rather to help improve the already existing system. This has nothing written so far so it does not lend itself to be continued upon by other teams without additional work. It would be ideal but ultimately not necessary and was arguably out of scope for our current prototype. However, it is still a key feature that should be attach to something when a final product is realised as it enables the system to have a greater degree of validation.

Future Work: SMS Login Alternative

It has been requested by one of our end users that we have an SMS alternative to the email-based login system. Many of the users will be on mobile phones, meaning that an SMS based login may be more popular than an email login overall.

Future Work: Display Last Time A Charger Status Was Updated

Currently, the system does not display when the last time a charger's status was updated. This could be a problem, since it may lead to charger information being out of date, which could lead to the users being frustrated that the system is telling them to go to charging points that are not free.

Future Work: Refactored Queue System

Another area for potential future development would be a slight overhaul for the queuing system within the product. This would be with a particular focus on transforming the system to more of a notification system which instead of placing user's in a queue would instead allow them to subscribe to a particular charge location and wait alongside other users for a general notification of a new space becoming available. This would transform the system to more of a "first come, first served" format.

Future Work: Continued User Testing

In order to gain an even more realistic idea of the suitability of this system, further potential user interviews and feedback sessions would be a positive course of action. The system would likely benefit with a more in-depth process of user testing to highlight more areas of future development and new features to improve it overall.

Issue: Users Are Given the Option To Join a Queue They Are Already In

A small bug in the current implementation that needs to be rectified in future development. Current users can be given the option to join a queue that they are already a part of. This is a fix to be implemented in the next stage of development om the system.

Issues: Email Host Reliability.

Description: The current emailing system has seen positive results however it is not without its issues. In the early phases of testing a situation arose which caused the backend to become locked out from access to the email account that would send notifications to the user.

This meant that new emails could not be transmitted. The cause of this issue was determined to be a combination of multiple simultaneous logins attempts to the google account that the email provider is associated with and the erratic nature of which the system sends its emails. This resulted in the security measures on the account triggering and closing the account down from outside access until a security check could be completed by an administrator. This problem has since been addressed by implementing a two-factor authentication security measure on the google account which has eliminated the risk of the system locking out the backend communication. The two-factor authentication security measure introduces its own complications however as it is tied to the mobile phone number of one of the developers which can cause issues in development if that individual cannot be notified of their needed assistance.

Due to the complications with the current mailing system described above the course of action agreed upon that would make most sense in future development is to implement an original email server that the system team could manage. This would address the limitations of the google account implementation without sacrificing security which is an overall benefit to the end users.