

Contents

I Literature Review - Cloud Services	2
1 Hypertext Transfer Protocol (HTTP)	2
2 Development Language	2
3 Cloud Computing Services	2
3.1 Compute Services	2
3.1.1 Serverless Architecture	3
3.1.2 PaaS	3
3.1.3 Google App Engine	3
3.2 Database	3
3.3 Mobile Notifications	4
3.4 Cloud Pub/Sub	4
3.5 Scheduling	4
II Cloud Services	5
1 Infrastructure Architecture	5
2 Data Model	6
2.1 User	6
2.2 Charging Structure	7
2.2.1 Location	7
2.2.2 Charger	7
2.2.3 Session	7
3 Flask Server	7
3.1 RESTful Services	8
3.2 Blueprints	8
4 Security	8
4.1 Passwords	9
4.2 Authentication	9
4.2.1 Bearer Token	9
5 Pricing	10
6 Results	10
7 Future Work	10
8 Summary and Future Work	11
III Discussion and Conclusions	12
1 Discussions	12
1.1 Environmental Impacts	12
1.2 The Future of Mobility Grand Challenge	12
1.3 User Behaviour	12
1.4 Group Work	12
2 Conclusions	13
References	13
A Results	14

Part I. Literature Review - Cloud Services

1 Hypertext Transfer Protocol (HTTP)

The core of the backend services in the context of this project is a HTTP (Hypertext Transfer Protocol) server. When running, a server is available at a hostname (<http://google.com>) which can be queried over the internet. HTTP servers are ubiquitous on the internet and most internet communications from retrieving websites to using web connected mobile apps and streaming video takes place between a client and an HTTP server. An HTTP server can be run locally on a PC but requires proper hosting in order to allow communications over the public internet.

Apart from their direct use by the user in websites and video streaming, HTTP can also be used to form APIs (Application Programming Interface) where structured text is instead sent back and forth to represent actions in a service.

At a high level HTTP APIs typically act as a layer of logic between users and databases for data persistence.

2 Development Language

There are many languages well suited to writing HTTP servers from well established ecosystems like PHP, Java and Ruby to newer environments such as Python and Javascript using Node.js. An understanding of the language combined with high speed of prototyping led Python to be chosen.

Libraries for creating HTTP applications in Python typically implement the Web Server Gateway Interface (WSGI), an interface for passing web requests from the web server which hosts the application to the application itself. This theoretically allows any WSGI server to run any application written using a WSGI framework.

Two of the most popular frameworks are Flask[1] and Django[2].

Django has a strong database abstraction layer and is “somewhat-opinionated” about the structure of web applications. Flask however has a more minimalist structure, functioning as a micro-framework that lightly wraps around both the Werkzeug library for a WSGI toolkit and Jinja for template engine support. It has no database layer or form validation, however it’s un-opinionated structure and flexibility proves to be agile and very powerful. For these reasons Flask was chosen for implementing the WSGI interface.

3 Cloud Computing Services

Google Cloud Platform[3] (GCP) is a platform of cloud computing products from Google offering services from compute, storage and databases to networking, AI and IOT. Built upon the same infrastructure that hosts the public facing Google services, GCP is one of the largest and most feature rich platforms of it’s type operating in the same space as Amazon Web Services and Microsoft’s Azure platform.

With effective documentation and compelling free quotas combined with previous experience with the platform, GCP was used to provide all the hosting needs for the project.

3.1 Compute Services

Deploying an HTTP server to the public internet can be done in various ways. One of the more established but manual approaches is to create a Linux virtual machine on which WSGI applications can be started using a WSGI server like Gunicorn. This requires careful Linux administration to create a secure public facing server, the growth of cloud computing platforms has removed the requirement for such overhead.

3.1.1 Serverless Architecture

Serverless computing describes a new type of cloud service architecture that abstracts away much of the required administration, aiming to make the user experience feel “serverless”. There are many different types of service using the architecture with the most popular being function-as-a-service or FaaS, branded Cloud Functions on GCP. FaaS services allow users to write and deploy individual functions or methods of source code to be run by a trigger.

These triggers can include database state changes, asynchronous message queues or HTTP requests providing single functions with unique URLs to trigger them.

3.1.2 PaaS

Platform as a service or PaaS is a similar architecture retaining many of the advantages of serverless products. It is a step towards the more manual approaches and implements the serverless ethos in a different way. To compare to FaaS which provides serverless hosting for individual methods of source code, PaaS can be seen as serverless hosting of entire WSGI applications when working in Python. In doing so the advantages of serverless hosting such as reduced configuration and high scalability are retained while allowing developers to create APIs in the industry standard WSGI format.

One of the main advantages of serverless (including PaaS) architecture is it’s ability to horizontally scale on demand. Horizontal scaling describes a service’s ability to handle concurrent traffic, typically scaling horizontally is achieved by running more instances of an application concurrently. This is in contrast to vertical scaling which instead describes a services ability to handle more traffic with the same resources, typically by making each instance faster.

3.1.3 Google App Engine

Google App Engine (GAE) is a PaaS service by GCP providing application hosting for many different languages including Python. GAE provides many powerful features that either require complex configuration or are unfeasible with bare-metal or virtual machine based solutions.

For example GAE provides automatic horizontal scaling allowing the number of application instances to dynamically increase in response to increased traffic. The scope of the GCP infrastructure gives effectively unlimited horizontal scale to smaller services that would otherwise be unable to handle such demand.

GAE also makes using TLS security extremely easy. With a manual hosting set-up, TLS security can be included by attaching a certificate to the hosting web server, either purchased from a certificate authority such as Verisign or for free generated by Let’s Encrypt. It is also advisable to include a rule within the web server in order to redirect unencrypted HTTP traffic on port 80 to encrypted HTTPS traffic on port 443.

App Engine includes TLS security with a Let’s Encrypt certificate as standard and within the deployment configuration automatic redirects from HTTP to HTTPS can be added with one line. These two features can make development and deployment fast and secure.

3.2 Database

GCP has many options for database services depending on the requirements. For both SQL and NoSQL, GCP has databases designed to operate at different scales and with different features. For example, Cloud SQL offers managed MySQL, PostgreSQL and SQL server instances with ultra low latency for smaller scale use but Cloud Spanner offers scalable SQL instances designed to be highly consistent at a global scale.

Equally, for NoSQL there are two main options. Cloud Bigtable offers global scale NoSQL database instances able to handle petabyte sized datasets designed for analytical big data workloads. Alternatively, Cloud Firestore is a new service from GCP under the Firebase brand banner.

Being marketed as the new flagship NoSQL product, Firestore has key integrations with the rest of the GCP services, for example Cloud Functions can be triggered directly by changes within the database and Firebase provides authenticated access to the data from within mobile applications.

Firestore is also highly scalable and consistent as a result of its serverless architecture.

3.3 Mobile Notifications

Mobile notifications are a useful tool for providing instant messages and updates to a user having superseded the previously accepted methods of email and SMS messages.

While mobile applications can locally generate notifications, a more useful method of using notifications is for them to be generated and delivered by a cloud based service and transmitted over the internet. This allows changes in the service's state such as changes in a user's car battery level to be relayed to the user.

Due to the architecture of delivery and display when pushed over the internet, notification systems are usually part of the mobile operating system such as Firebase Cloud Messaging for Android and Apple Push Notification Service for iOS. Using Firebase Cloud Messaging messages can be sent to individual devices or groups of devices, the former being identified using a device specific key.

3.4 Cloud Pub/Sub

Within a cloud system, the ability to orchestrate different aspects of the architecture can be facilitated by sending messages between services. This allows asynchronous updates and the spawning of tasks while maintaining independence of different parts of the architecture. A widely useful way to do this is by creating a communication channel within which messages can be published and subscribed to. Cloud Pub/Sub is a GCP service facilitating this form of communication. While it has many applications it is particularly useful for starting long running tasks triggered by HTTP requests or for background tasks within a service.

In order to fully utilise a serverless architecture as described above, FaaS Cloud Functions can be configured as subscribers to such messages, increasing the speed with which messages can be processed.

3.5 Scheduling

When operating a system of larger scale, typically there will be tasks required to run repeatedly at predefined time intervals. Cloud Scheduler is a service within the Google Cloud Platform taking inspiration from the Unix tool, Cron, allowing commands to be run in the background at flexibly defined times. Cloud Scheduler uses the same string format defining time schedules and has many different actions including Google App Engine URL endpoints and publishing messages to Pub/Sub message channels.

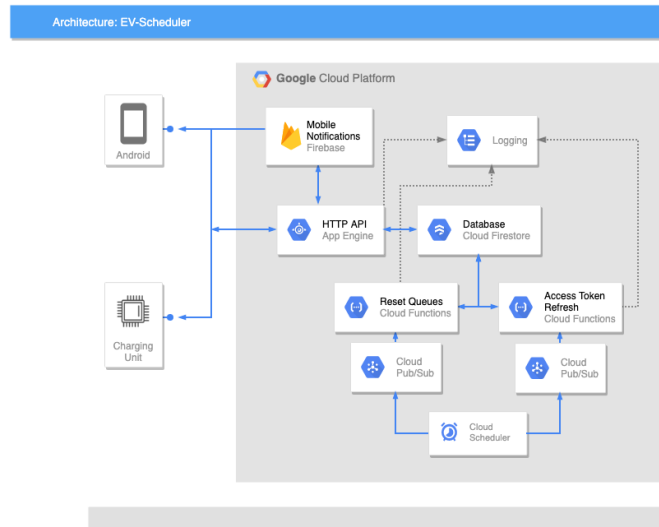


Fig. 1: Architecture of Google Cloud Platform services

Part II. Cloud Services

When designing a centralised commercial product with services available over the internet, the code to run those services cannot solely be run in individual phone apps or on IOT-esque pieces of hardware. The need for an authoritative store of information such as user details and service specific information (location and charge station details) requires a central instance of code to store information and make that data available when properly authorised.

The relationship between the code that the user interacts with directly (mobile app, charging unit) and that which centrally controls the service is described as the separation between the front-end and back-end of an application.

More precisely in the front-end/back-end model the back-end acts as the data access layer (DAL), while the front-end is the presentation layer for a service.

The source code for the cloud services module can be seen at <https://github.com/Sarsoo/ee3035-ev-scheduler>.

1 Infrastructure Architecture

The architecture of cloud services utilised on GCP can be seen in figure 1.

Google App Engine was used to host the Flask application in the standard environment which proves to be more than suitable for these requirements. This GAE application provides the public web hostname to interface with the mobile application and charging unit.

The need for a database was fulfilled by Firestore. Although a propriety SDK must be used to interface with the database, the library does have many benefits such as automatically authenticating database actions when code is running on deployed GCP services.

Cloud Scheduler is used to run utility tasks for the service at regular intervals, these tasks can be in the form of messages published to Cloud Pub/Sub queues. Each task is encapsulated within a serverless Cloud Function which is triggered via subscription to separate Cloud Pub/Sub queues. Cloud Scheduler can then publish messages to these queues for consumption.

The access token refresh task is part of the authentication workflow, described further in section 4.2. In summary the confidential keys used to authenticate with the API should be refreshed periodically to reduce the value of any leaked data, this function performs that refresh action for all required users.

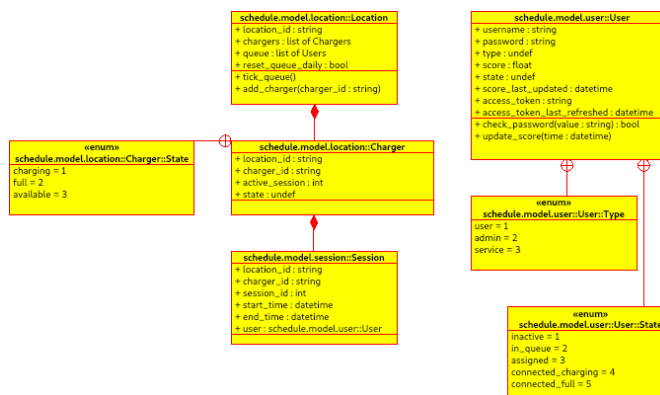


Fig. 2: Class diagram for data model

The reset queue tasks performs actions related to the semantics of the service. In the context of a workplace, the queue of users waiting for charging sessions should not necessarily roll over to the next day. Electric vehicle owners will typically charge their cars overnight leaving charging at work to function as a top up for the return commute. As a result, if a user is not able to receive a session that day, it does not necessarily mean they would still want an available session the next day. To allow this, locations can be configured to have the queue of users reset at midnight.

Each of the compute services being used, namely the GAE application and the serverless functions can be configured to use the centralised logging provided by GCP allowing intuitive review.

Finally the Firebase SDK is used in order to deliver mobile push notifications to users using the Firebase Cloud Messaging API.

2 Data Model

In designing the data model for this project objects were defined for users and the charging structure, a UML diagram for the architecture can be seen in figure 2.

2.1 User

Users have associated usernames and passwords for identification and authentication. Usernames are the primary key differentiating objects and are case insensitive. Passwords are salted and hashed using the Werkzeug security functionality, for more information see section 4.1.

Defining user types allowed the implementation of access controls between different use cases. End-users are classed as 'user' and have limited access to much of the resource types while 'admin' accounts are able to make every types of request, naturally this set of users would be heavily restricted. Service accounts are used by the charging units in order to make changes on behalf of the system. This includes changing the state of the charging structure throughout the life cycle of a charging session, end-users cannot make any changes of this sort.

The state of the user is defined by the life-cycle of a charging session as described in the associated enumerated type. Using Python properties, specific state changes for a user have associated actions, this is where mobile notifications are triggered in order to inform the user of changes in the charging state.

Users have an access token field and associated last updated time in order to secure the HTTP API, the authentication methods used are covered in section 4.2.

2.2 Charging Structure

The charging architecture is defined by two objects modelling physical structure and a session object. A Location defines a physical location at which are found charging stations. An office building with electric vehicle parking spaces would be a Location as would be a multi-storey car park. Functionally it is any group of charging spots.

A Charger is an individual charging station which will have an associated charging unit to relay session information.

A Session defines a discrete charging window completed by a user. It begins when the queue starts a session for a user who is informed as such. The user begins charging their car and upon completion is notified that they can remove their car. When a user unplugs the car the session ends and ticks the location's queue to select the next user.

Locations and chargers have string IDs while session IDs are integers. This is done so sessions can be stored for logging purposes following completion and new session IDs can be generated easier by iterating the max value of the saved IDs.

2.2.1 Location

A Location has a `location_id` to act as the primary key, they are case insensitive. A location has a list of contained chargers and a list of users representing the queue waiting to charge. The `tick_queue()` function updates the score for each user in the queue and selects the lowest scoring to receive the next session if available. The queue is ticked both when a user is added to the queue and when a session ends to ensure the time is used effectively.

2.2.2 Charger

A Charger's associated `charger_id` differentiates chargers of a location, they are unique to that location's namespace. The active session field is for the integer `session_id` of the currently running session if one is available. A null value indicates no session is running. The state field functions similarly to the same for the User object and defines the charger's position within the life cycle of a session.

2.2.3 Session

Sessions are retained after their end for logging purposes, as such integer session IDs were selected in order to allow easier retrieval of new IDs. To do so the maximum value of all available IDs is incremented. Similarly to the charger key, session IDs are unique within their charger's namespace, this allows a single session to be identified by a fully qualified ID, separated by colons for example,

```
location1:charger1:5
```

Due to the use of IDs in request URLs and the fully qualified ID, colons, spaces and forward slashes are illegal characters for all three IDs. Sessions have date-time objects for both their start and end time and a reference to the User object who owns the session.

3 Flask Server

Flask[1] is a lean Python web framework that provides flexibility and extensibility for web server development. The framework is used by instantiating a Flask application and registering routes with the app object.

Routes are registered using a function decorator defining that method with a URL string pattern to match requests. The return value of the method is passed back to the HTTP requester as the response. This decorator takes both the intended route string pattern and the available HTTP methods as parameters.

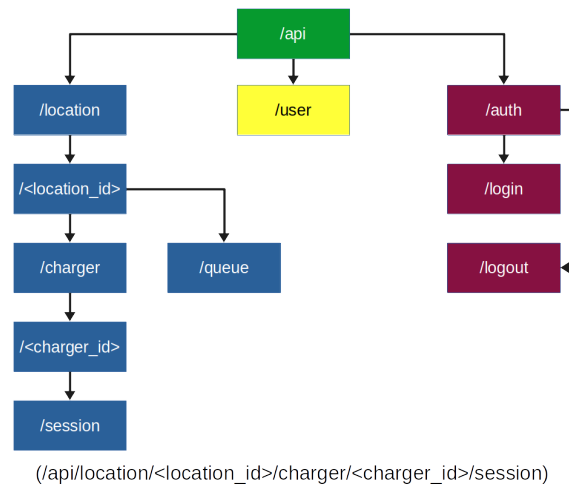


Fig. 3: Visualisation of API endpoints with example fully qualified route

3.1 RESTful Services

When designing a web server a modern philosophy for doing so is the REST architecture. REST or Representational State Transfer is a methodology for defining the semantics of what each URL means. Each URL is treated as an identifier for a resource it describes, making requests with different HTTP methods at this URL represents different actions to be taken on the object that URL identifies. The most common methods are GET, POST, PUT and DELETE.

GET is typically used as a read only method who's result is a text serialization of the object.

The POST method is used to create an object while PUT is used to replace the object. For this API PUT is used to replace individual fields of some objects making the definition of PUT closer to update.

DELETE is for deleting the object the URL denotes.

The aim of a RESTful web service is to make interacting with APIs consistent and intuitive both intrinsically and when comparing different APIs. It is similar to a design pattern.

A diagram laying out the structure of the API URL routes can be seen in figure 3.

3.2 Blueprints

A useful tool for encapsulating discrete parts of a Flask API that should still be part of a single application is to use blueprints.

Blueprints group a set of routes allowing them to be mounted on a Flask application object together. Blueprints have routes registered in a similar fashion as the Flask application, blueprints are then individually hung off of a prefix URL string by the application following instantiation.

Blueprints were used to separate routes for authentication, users and the charging structure in order to closely follow the data model, a visualisation of the final URL route map can be seen in figure 3.

4 Security

When operating a cloud-based commercial service arguably the most important aspect of it's design is that of security, breaches increasingly occur in large public companies allowing unauthorized access to customer's private information.

Fortunately, using a serverless architecture abstracts away much of the required boilerplate security administration such as managing complex firewall setups and intrusion detection and prevention systems (IDS/IPS), instead much of this low-level security is provided by the infrastructure itself.

The three remaining areas of this service requiring considerations from a security perspective are those of access controls (section 2.1), password handling and authentication. Access controls have been discussed above in the data model, it involves considering which users should be able to conduct what actions and what the implications of such access are. For example ensuring that only system owned accounts, admin accounts and the user who owns a session can manipulate it.

4.1 Passwords

Passwords should never be stored at rest in the cleartext form they are provided by users as any breaches of the database will expose sensitive user information. The impact of such a breach would not only cover this service but could prove devastating for users who use the same password across websites (however unadvisable that is), this proves to be valuable attack vector for bad actors.

To secure passwords they should undergo obfuscation processes such as salting and hashing. Hashing is a one-way function that takes a variable length string and produces a reproducible fixed length output, the same input string always provides the same output string. Importantly for password storage is the one way characteristic of the function, this means the hashed password cannot be inversely “un-hashed” back into the clear text string, even with knowledge as to what type of hashing was used. Cracking hashed passwords must instead be done with large tables of cleartext passwords and the associated precomputed hashes, this is known as a rainbow table attack. In order to protect against brute force attacks of this fashion, salts are additionally used. A salt is a random string of data that is appended to the user password before hashing and then stored with it. This reduces the effectiveness of a rainbow table attack as rainbow tables cannot practically be pre-calculated for input passwords when mangled by a random string in this way. For this implementation passwords are salted using a 8 character long salt before being hashed using the pbkdf2:sha256 algorithm.

4.2 Authentication

Authentication is a crucial concern when designing web services. Access controls and other security measures are important facets of a system however they are useless if weak authentication allows them to be circumvented. There are many ways to secure a HTTP API that range in security and complexity, finding a balance for the context of the service is important. Authentication describes the methods by which a user interacting with the API proves they are who they say they are. This is separate to the process of authorization which describes the process of proving that a user is able to do what they are trying to do. This difference is relevant when considering the intricacies of more complex security architectures however for these purposes both are done at the same time.

Most methods of authentication eventually authenticate each request using a HTTP header with the “Authorization” key of key-value pairs. The value of this header prescribes which method is being used.

During development basic authentication was used, this involves sending the username and password for the requesting user in the header of each request. This however requires the client to keep unprotected forms of the user credentials to be ready for inclusion in each request and must be used in conjunction with HTTPS to protect them from interception. For release a more secure form of authentication was used, bearer tokens.

4.2.1 Bearer Token

A bearer token delegates the authentication authority from the user credentials to a separate piece of secret information. In doing so, the need to repeatably share user passwords or the derivatives of is removed. This additional piece of information is a string of characters typically referred to as a key or token, it is provided with each request in the authorization header in the form

Bearer <token>

Bearer tokens in one form or another are the most common form of authorisation and authentication for web based APIs with standards such as OAuth (2.0) and JSON web tokens (JWT) being well regarded technologies in this space. OAuth describes an architecture for API authorization however while being a proven structure, the system is fairly complex and has an emphasis on allowing third party services delegated access to user information, something that wouldn't be required for a service that expects only first party access.

This service uses a simpler implementation of bearer tokens than seen in larger architectures like OAuth 2.0 with each user having a unique and randomly generated key stored with their user information. Tokens are refreshed on a weekly basis, in doing so any compromised key has its value reduced as a result of its limited life time.

5 Pricing

An important aspect of using a third-party hosting platform for all the required services is the prices for each service, both currently and when anticipating future growth. When looking back to figure 1 each of the listed services have free tiers and prices following these quotas. Using estimations of possible system loads and the Google Cloud Platform pricing calculator[4] an estimation of running costs can be obtained.

When considering the reasonable usage of the service and the free quotas available for each used product, It is not expected that any of the services would require payment at low usage. As interaction increases the first service that would likely require payment would be the GAE app. While the 28 instance hour per day limit is a healthy quota, high simultaneous usage at commuting times and throughout the work day could require payment for more instances. Past this point, instances cost \$0.06 per hour.

6 Results

An example JSON output from the API can be seen in appendix A. The Flask server proved extremely flexible in designing and implementing the required URL routes. The un-opinionated environment was a strength as Firestore requires a proprietary SDK for database actions, a simple database layer was written to match much of what would be achieved with a traditional Object-Relation Mapping (ORM).

Worth noting however is that this was not as fast as industry standard libraries such as SQL-alchemy could provide, reading and writing to the database were the slowest aspects of the program and could definitely be optimised further. This architecture was achieved while remaining within the free quotas for all services.

7 Future Work

The back-end has room for development, future work could develop both the infrastructure and the source code itself. The infrastructure could be extended by increasing the utilisation of serverless functions and by considering the AI and IOT products provided by the platform.

The source code has more room for improvement, specifically the database layer could be optimised as it is currently the bottleneck for typical use. Retrieving a session, for example, involves first retrieving its parent charger and location which slows down processing.

Additionally the authentication scheme could be developed. While the bearer tokens as implemented are suitable, more complex systems have significant advantages. Specifically the use of JWTs would decrease processing time and provide more authentication information from a debugging and security perspective.

8 Summary and Future Work

An array of cloud computing services have been orchestrated into a system functioning as the back-end for the scheduling project.

An HTTP server provides the public gateway to data contained within a database. At each point services built with modern architectures provided by the serverless philosophy have been used effectively to keep the system simple and the cost free.

The security of the architecture has also been considered, both in the context of those aspects abstracted away through the infrastructure choices and those which are part of the service itself.

Costs for running the service were shown to be free for small scale deployment and reasonably priced if usage were to increase, the services most likely to begin costing money were laid out.

Finally the space for development is presented, highlighting the investigations that could be made both into the infrastructure and the source code itself.

Despite these areas the final product successfully fulfils the specifications for cloud data storage in a stable and self-descriptive manner.

Part III. Discussion and Conclusions

1 Discussions

1.1 Environmental Impacts

The subject of the project being electric vehicles has obvious links with the environment and as such the wider environmental effects should be considered.

Initially it could be considered that the wider use of a vehicle charge management system would be positive in this regard. This would be as a result of the incentive provided by being able to charge a user's car while at locations such as a workplace with limited spaces. As a result, the members of the workplace could be more likely to use electric vehicles over the alternative and the use of fossil fuelled cars would decrease.

However in practice, a management system of the type presented here encourages sustained usage of charging stations throughout the day, possibly leading to an overall increase in power usage.

At a time when the majority of power generated in the UK still comes from fossil fuels the overall effect could be negative. It is worth defining, then, the point of such a service and how it responds to the grand challenge it aims to solve.

1.2 The Future of Mobility Grand Challenge

This service aims to improve the future of mobility by easing the adoption of a new form of travel in electric vehicles. As the amount of energy generated through renewable means increases, the negative effects of increased usage is reduced.

The main drawbacks of EVs in comparison to their traditional fossil fuel counterparts are battery range, charging times and the availability of charging stations This service aims to allow the most efficient use of charging infrastructure such that these limitations can be reduced and the technology further encouraged.

1.3 User Behaviour

Another aspect of the service worth considering is the way in which bad faith interaction such as delaying the removal of a full car is reacted to and punished by the service.

The aim of a user's score is to punish bad actors by reducing their ability to get further sessions, this leaves the service to be utilised by users more willing to properly engage with the service's environment.

However, overrunning meetings or time sensitive work are both common and when considering how the service is ultimately a utility for the company owning the charging spots, few if any would prioritise the interactions with the car charging structure over the work being completed by the employee for that company.

This highlights both how reliant the service is on users engaging in good faith and how the system could be susceptible to abuse.

1.4 Group Work

Development of the three facets of the service were, in general, developed separately with a process of integration and final testing to assemble the final product.

This, in combination with a group structure that loosely divided between the software and hardware for the most frequent communication led to some integration issues that, with hindsight, could have benefited from a more collaborative approach or project management style. Each of the three aspects, however, did achieve their individual aims and provided the best components with which to complete the integrations.

2 Conclusions

As part of this project a service has been designed and implemented allowing the dynamic scheduling of electric vehicle charging time. In doing so, the shared use of limited spaces with charging units can be systematically managed and used more efficiently than the alternative.

Integrating the hardware unit with the native J1772 protocol of the existing charging infrastructure allows intuitive sessions to be defined by the subject vehicle as opposed to predefined time windows, further improving the efficacy of the service.

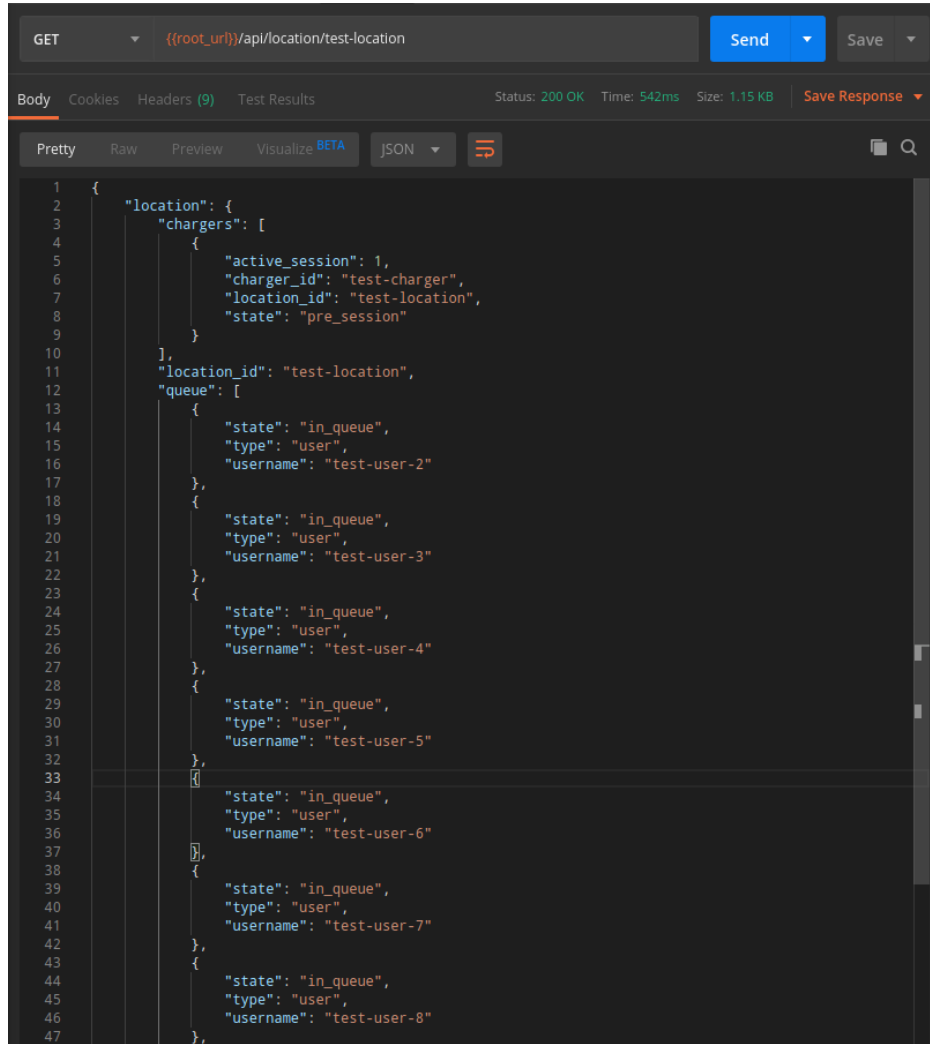
The use of industry leading cloud platforms allowed a back-end service to be created that is highly available, can scale automatically with demand and could naturally operate at a global scale were it required.

The Android mobile application provides a natural user interface for that back-end while the use of mobile notifications makes the entire experience responsive. Discussion's above highlight the relevance of this work to the grand challenge it aims to contribute to whilst also demonstrating the limitations of the design.

References

- [1] A. Ronacher, *Flask*, Pallets Projects. [Online]. Available: <https://github.com/pallets/flask>.
- [2] A. Holovaty and S. Willison, *Django*, Django Software Foundation. [Online]. Available: <https://github.com/django/django>.
- [3] Google, *Google cloud platform*, Google Cloud Platform. [Online]. Available: <https://cloud.google.com>.
- [4] —, *Pricing calculator*, Google Cloud Platform. [Online]. Available: <https://cloud.google.com/products/calculator>.

A Results



```
1 {
2   "location": {
3     "chargers": [
4       {
5         "active_session": 1,
6         "charger_id": "test-charger",
7         "location_id": "test-location",
8         "state": "pre_session"
9       }
10    ],
11    "location_id": "test-location",
12    "queue": [
13      {
14        "state": "in_queue",
15        "type": "user",
16        "username": "test-user-2"
17      },
18      {
19        "state": "in_queue",
20        "type": "user",
21        "username": "test-user-3"
22      },
23      {
24        "state": "in_queue",
25        "type": "user",
26        "username": "test-user-4"
27      },
28      {
29        "state": "in_queue",
30        "type": "user",
31        "username": "test-user-5"
32      },
33      {
34        "state": "in_queue",
35        "type": "user",
36        "username": "test-user-6"
37      },
38      {
39        "state": "in_queue",
40        "type": "user",
41        "username": "test-user-7"
42      },
43      {
44        "state": "in_queue",
45        "type": "user",
46        "username": "test-user-8"
47      }
48    ]
49  }
```