NATIONAL AND TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF TECHNOLOGY, INFORMATION AND COMPUTERS

Creation of a complete cloud system for the management of second life battery systems

Thesis Report

of

Ioannis E. Mytis

Supervisor: Panayiotis D. Tsanakas,
Professor of Computer Engineering NTUA

Athens, March 2025

National and Technical University of Athens
School of Electrical and Computer Engineering
Division of Information Technology and Computer

Creation of a complete cloud system for the management of second life battery systems

Thesis Report

of

Ioannis E. Mytis

Supervisor: Panayiotis D. Tsanakas,
Professor of Computer Engineering NTUA

Approved by the examination committee on 3rd March 2025:

.....	.....	.....
Panayiotis Tsanakas	Angelos Amditis	George Korres
Professor, NTUA	Reseacher A', ICCS-NTUA	Professor, NTUA

Athens, March 2025

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών

Δημιουργία ολοκληρωμένου συστήματος νέφους για τη διαχείριση μπαταριών

Διπλωματική Εργασία
του

Ιωάννη Ε. Μύτη

Επιβλέπων: Παναγιώτης Δ. Τσανάκας,
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Μαρτίου 2025:

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Παναγιώτης Τσανάκας Καθηγητής, Ε.Μ.Π.	Άγγελος Αμδίτης Ερευνητής Α΄, ΕΠΙΣΕΥ-Ε.Μ.Π.	Γεώργιος Κορρές Καθηγητής, Ε.Μ.Π.

Αθήνα, Μάρτιος 2025

(Signature)

.....
Ioannis E. Mytis
Graduate Electrical and Computer Engineer NTUA

Περίληψη

Η χρήση των μπαταριών για δεύτερη ζωή θα έχει έναν ολοένα και πιο σημαντικό αντίκτυπο στη ζωή μας, όπως αναφέρει η Ευρωπαϊκή Ένωση. Εν τω μεταξύ, η ζήτηση για συστήματα μπαταριών αυξάνεται ραγδαία και αρκετά συστήματα μπαταριών βρίσκονται κοντά ή έχουν ήδη φτάσει στο τέλος της πρώτης τους ζωής, ενώ εξακολουθούν να διατηρούν την ικανότητα να επαναδιαμορφωθούν για άλλες χρήσεις. Ταυτόχρονα, δεν υπάρχει ένα ισχυρό πλαίσιο που να καθιστά την επαναχρησιμοποίηση αυτών των μπαταριών εύκολη, ασφαλή και αποδοτική.

Αυτό οδηγεί σε προβλήματα όπως υψηλοί παράγοντες κινδύνου κατά την αποσυναρμολόγηση των μπαταριών και δυσκολίες στον σχεδιασμό, την πρόβλεψη και την παρακολούθηση της απόδοσης των συστημάτων μπαταριών δεύτερης ζωής. Κατά συνέπεια, αυτό δημιουργεί σημαντικά εμπόδια για τη συμμετοχή εξωτερικών ενδιαφερόμενων επιχειρήσεων της αλυσίδας αξίας της ανακύκλωσης μπαταριών. Ως αποτέλεσμα, σε μια περίοδο που χαρακτηρίζεται από τη ραγδαία αύξηση της παραγωγής αποβλήτων μπαταριών, το δυναμικό για την επαναχρησιμοποίησή τους μειώνεται, επηρεάζοντας τόσο την επαναχρησιμοποίηση των μπαταριών όσο και τη βιωσιμότητα του περιβάλλοντος.

Αυτή η διπλωματική εργασία ασχολείται με τη δημιουργία μιας cloud πλατφόρμας για τη σωστή διαχείριση και επικοινωνία πληροφοριών σχετικά με συστήματα μπαταριών δεύτερης ζωής, στο πλαίσιο του ευρωπαϊκού έργου Horizon, Battery2Life. Θα υλοποιήσει δομές δεδομένων σύμφωνα με το Ψηφιακό Διαβατήριο Μπαταρίας, αποδοτικούς μηχανισμούς αποθήκευσης δεδομένων και θα παρέχει ασφαλείς και ευέλικτους τρόπους επικοινωνίας αυτού του ευρέος συνόλου δεδομένων σε όλους τους ωφελούμενους της αλυσίδας αξίας της δεύτερης ζωής των μπαταριών.

Στο Κεφάλαιο 1, θα εξετάσουμε το ζήτημα της διάρκειας ζωής των μπαταριών πέρα από την αρχική τους χρήση και θα παρουσιάσουμε το τμήμα μας στο έργο Horizon Battery2Life. Στο Κεφάλαιο 2, θα καλύψουμε τις συγκεκριμένες απαιτήσεις της πλατφόρμας, θα παρέχουμε λεπτομερή παραδείγματα των API endpoints και θα περιγράψουμε την αρχιτεκτονική της λύσης,συμπεριλαμβανομένων των στοιχείων της και των αλληλεπιδράσεών τους. Στο Κεφάλαιο 3, θα συζητήσουμε πώς υλοποιήσαμε την πλατφόρμα σύμφωνα με τις προδιαγραφές και την αρχιτεκτονική που περιγράφηκαν στο προηγούμενο κεφάλαιο, και θα εξηγήσουμε γιατί επιλέξαμε τις συγκεκριμένες τεχνολογίες καθώς και πώς τις διαμορφώσαμε. Στο Κεφάλαιο 4, θα καθορίσουμε τη μεθοδολογία δοκιμών της πλατφόρμας μας, θα διεξάγουμε τις δοκιμές, θα παρουσιάσουμε τα αποτελέσματα και θα αναλύσουμε τα βασικά σημεία ενδιαφέροντος. Στο Κεφάλαιο 5, θα ολοκληρώσουμε αυτήν τη διπλωματική εργασία, επισημαίνοντας τα κύρια συμπεράσματα και προτείνοντας μελλοντικές βελτιώσεις για την cloud πλατφόρμα Battery2Life.

Λέξεις Κλειδιά — Διαχείρηση Μπαταριών, Ψηφιακό Διαβατήριο Μπαταρίας, Ανακύκλωση Μπαταριών, API, Cloud, Django, Docker

Abstract

The act of utilizing batteries beyond their first life will have an increasingly important impact on our lives, as stated by the European Union. Meanwhile, the demand for battery systems is skyrocketing and several battery systems are close or have reached the end of their first lives while still maintaining the capacity to be reconfigured for other use cases. At the same time, there is no robust framework that makes the task of reusing these batteries easy, safe and efficient.

This leads to problems including high risk factors for the disassembly of batteries and difficulty in designing, predicting/monitoring the performance of 2nd life battery systems. Consequentially, this poses significant obstacles for the participation of outside stakeholders in the recycling value chain. As a result, during a time marked by a rapid rise in battery waste production, the potential for reusing batteries is reduced, affecting both the reuse aspect of batteries and environmental sustainability.

This thesis takes up the matter of building a cloud platform for the correct management and communication of information of second life battery systems within the scope of the European Horizon project Battery2Life. It will implement data structures according to the Digital Battery Passport, efficient data storing mechanisms and provide secure and flexible ways of communicating this wide set of data to all the benefactors of the second battery life value chain.

In Chapter 1, we will explore the issue of battery life beyond initial use and introduce our segment of the Horizon Battery2Life project. In Chapter 2, we will cover the platform's specific requirements, provide detailed examples of the API endpoints, and outline the solution's architecture, including the solution components and their interactions. In Chapter 3, we will discuss how we implemented the cloud platform according to the specifications and architecture outlined in the previous chapter, and explain why we selected the technologies we did, as well as how we configured them. In Chapter 4, we will establish the testing methodology for our platform, carry out the tests, present the results, and discuss key focus areas. In Chapter 5, we will conclude this thesis by highlighting key takeaways and offering suggestions for future work on the Battery2Life cloud platform.

KEYWORDS — Battery Management, Digital Battery Passport, Cloud, Django, Docker, Battery 2nd Life, API

Contents

List of Figures

List of Tables

Εκτενής Περίληψη

Η χρήση των μπαταριών για δεύτερη ζωή θα έχει έναν ολοένα και πιο σημαντικό αντίκτυπο στη ζωή μας, όπως αναφέρει η Ευρωπαϊκή Ένωση. Εν τω μεταξύ, η ζήτηση για συστήματα μπαταριών αυξάνεται ραγδαία και αρκετά συστήματα μπαταριών βρίσκονται κοντά ή έχουν ήδη φτάσει στο τέλος της πρώτης τους ζωής, ενώ εξακολουθούν να διατηρούν την ικανότητα να επαναδιαμορφωθούν για άλλες χρήσεις. Ταυτόχρονα, δεν υπάρχει ένα ισχυρό πλαίσιο που να καθιστά την επαναχρησιμοποίηση αυτών των μπαταριών εύκολη, ασφαλή και αποδοτική.

Αυτό οδηγεί σε προβλήματα όπως υψηλοί παράγοντες κινδύνου κατά την αποσυναρμολόγηση των μπαταριών και δυσκολίες στον σχεδιασμό, την πρόβλεψη και την παρακολούθηση της απόδοσης των συστημάτων μπαταριών δεύτερης ζωής. Κατά συνέπεια, αυτό δημιουργεί σημαντικά εμπόδια για τη συμμετοχή εξωτερικών ενδιαφερόμενων επιχειρήσεων της αλυσίδας αξίας της ανακύκλωσης μπαταριών. Ως αποτέλεσμα, σε μια περίοδο που χαρακτηρίζεται από τη ραγδαία αύξηση της παραγωγής αποβλήτων μπαταριών, το δυναμικό για την επαναχρησιμοποίησή τους μειώνεται, επηρεάζοντας τόσο την επαναχρησιμοποίηση των μπαταριών όσο και τη βιωσιμότητα του περιβάλλοντος.

Αυτή η διπλωματική εργασία ασχολείται με τη δημιουργία μιας cloud πλατφόρμας για τη σωστή διαχείριση και επικοινωνία πληροφοριών σχετικά με συστήματα μπαταριών δεύτερης ζωής, στο πλαίσιο του ευρωπαϊκού έργου Horizon, Battery2Life. Θα υλοποιήσει δομές δεδομένων σύμφωνα με το Ψηφιακό Διαβατήριο Μπαταρίας, αποδοτικούς μηχανισμούς αποθήκευσης δεδομένων και θα παρέχει ασφαλείς και ευέλικτους τρόπους επικοινωνίας αυτού του ευρέος συνόλου δεδομένων σε όλους τους ωφελούμενους της αλυσίδας αξίας της δεύτερης ζωής των μπαταριών.

Στο Κεφάλαιο 1, θα εξετάσουμε συνοπτικά το ζήτημα της διάρκειας ζωής των μπαταριών πέρα από την αρχική τους χρήση και θα παρουσιάσουμε το τμήμα μας στο έργο Horizon Battery2Life.

Στο Κεφάλαιο 2, θα καλύψουμε τις συγκεκριμένες απαιτήσεις της πλατφόρμας σε σχέση με την αποθήκευση των δεδομένων. Ύστερα, θα παρέχουμε λεπτομερή περιγραφή των API endpoints μαζί με αναλυτικά παραδείγματα που θα περιέρχουν μία γενική περιγραφή του endpoint, τη μέθοδο που χρησιμοποιήθηκε, την είσοδο/ κορμό του αιτήματος εάν υπάρχει και την αναμενόμενη απάντηση του endpoint.τέλος, θα σταθούμε στην αρχιτεκτονική της λύσης, συμπεριλαμβανομένων των στοιχείων, των αλληλεπιδράσεών τους και της γενικότερης χρήσης του συστήματος με συγκεκριμένα σενάρια χρήσης.

Στο Κεφάλαιο 3, θα περιγράψουμε το πώς υλοποιήσαμε την πλατφόρμα σύμφωνα με τις προδιαγραφές και την αρχιτεκτονική που περιγράφηκαν στο προηγούμενο κεφάλαιο. Πιο συγκεκριμένα θα αναλύσουμε τη δομή του κάθε μέρους του συστήματος και θα εξηγήσουμε γιατί επιλέξαμε τις συγκεκριμένες τεχνολογίες καθώς και πώς υλοποιήσαμε την λύση μας. Αυτά τα μέρη περιλαμβάνουν:

- τα πρωτόκολλα επικοινωνίας (REST, MQTT)
- την ανάπτυξη των endpoints (Django, Django Rest Framework)
- την καταγραφή των endpoints (Swagger UI)
- τη βάση δεδομένων και την παραμετροποίηση της (PostgreSQL)
- το σύστημα καταγραφής γεγονώτων (Python Loggers)
- βασικές οδηγίες χρήσης του συστήματος

Στο Κεφάλαιο 4, θα μιλήσουμε για το πως υλοποιήσαμε τον έλεγχο της πλατφόρμας και ποιες μετρικές χρησιμοποιήσαμε προκειμένου να διασφαλίσουμε την ποιοτική του λειτουργία. Πιο συγκεκριμένα, θα καθορίσουμε τη μεθοδολογία δοκιμών της πλατφόρμας μας σε ότι αφορά τη λειτουργικότητα και την ποιοτική λειτουργία της πλατφόρμας. Στην συνέχεια, θα διεξάγουμε τις δοκιμές και θα συλλέξουμε τα δεδομένα. Τέλος, θα παρουσιάσουμε τα αποτελέσματα των δοκιμών και θα αναλύσουμε τα βασικά σημεία ενδιαφέροντος με έμφαση στους ποιοτικούς περιορισμούς που έχουμε θέσει παραπάνω.

Στο Κεφάλαιο 5, θα ολοκληρώσουμε αυτήν τη διπλωματική εργασία κάνοντας μία ανασκόπησή σχετικά με το τι θέλαμε να πετύχουμε, τι καταφέραμε και πως το καταφέραμε επισημαίνοντας τα κύρια συμπεράσματα. Τέλος, θα μιλήσουμε για την ετοιμότητά της πλατφόρμας μας για την παραγωγή και θα αναλύσουμε κάποια σημεία ενδιαφέροντος για μελλοντικές βελτιώσεις για την cloud πλατφόρμα Battery2Life.

Λέξεις Κλειδιά — Διαχείρηση Μπαταριών, Ψηφιακό Διαβατήριο Μπαταρίας, Ανακύκλωση Μπαταριών, API, Cloud, Django, Docker

Chapter 1

Introduction

1.1 Problem Statement

Battery utilization has become increasingly significant in today's world. Batteries vary in application, shape, and material composition with unique traits especially evident in larger battery packs.

Currently, about 54 million electric vehicles (EVs) are in use globally[1]. In 2024, EV sales reached 16 million units, with projections rising up to 20 million by 2025. Furthermore, the electric truck market is predicted to surpass 1 million units within five years, up from 100,000 units three years ago. Typically, these battery packs carry a warranty ranging from 5 to 10 years, with an annual capacity decline of approximately 2.3%. Consequently, by 2030, around 5 million metric tons of battery modules will have reached the end of their lifecycle, retaining 70-80% of their original capacity.

Coincidentally, the demand for motive batteries of any kind is set to reach around 50GWh by 2025. This situation presents a substantial opportunity for reusing these modules in a second life, albeit with challenges. Notably, the absence of standardized packaging hampers disassembly efforts posing high costs, safety risks, and potential large scale cell defects. Additionally, the lack of uniform battery state monitoring coupled with the inherent inability of knowing how a battery has been used, poses the seemingly insurmountable challenge of not knowing how a battery can behave in second-use applications. This inevitably will make its use potentially inefficient and even dangerous. Finally, transferring the Battery Management System (BMS) to a secondary application is seldom quite complicated, due to varied BMS specifications.

In this part of the thesis diploma, we are going to attempt to give the reader a general overview of the aims, goals and objectives of the Battery2Life (B2L) initiative. Then we are going to do the same with our particular given task (which is the implementation of the Battery2Life cloud solution) and try to explain how it interacts with all the other components of B2L. Then we are going to give a brief and abstract synopsis of the architecture of our solution with a more detailed description of the RESTful Application Programming Interface (API) and the Data Management Schema (DMS). We are going to continue with the testing of the platform and presentation of the results and key points of interest. Finally, we are going to conclude by summarizing what we have achieved and referring future areas of improvement.

1.2 The Battery2Life (B2L) Horizon Project

The Battery2Life project aims to impact the battery industry significantly by introducing innovative solutions for the second-life of batteries, with a stronger focus on the stationary energy storage systems (ESS)[1]. The project will contribute to several key outcomes, including the development of an open and adaptable cloud-based interface that enables effortless communication of battery data between 3rd parties and the BMSs, improved safety and reliability of battery system by monitoring through embedded sensing and State-of-X (SoX) estimation algorithms, and new system designs that facilitate the disassembly and reconfiguration of batteries for second-life applications.

The Battery2Life project is designed to have a substantial impact in a variety of fields. Notably, these advancements are expected to reduce the time and cost of repurposing EV batteries for second-life use, extend the lifespan of batteries, and enhance their environmental and economic sustainability. The project also aims to support promoting circularity and recycling in the battery industry, thereby reducing the carbon footprint of Europe.

1.3 The Battery2Life Cloud Platform

As mentioned above, the challenges that jeopardize the extension of battery use to a second life are immense. One of the most important ones being the niche properties that current BMS design present, tailored to the specific needs of the application in use. Therefore, within the vision of perfect reuse of batteries, an absolutely and unequivocally vital step towards the right direction is to design data-driven and application-agnostic Battery Management Systems.

The emerging and long-lasting goal of Battery2Life is to provide the enablers that foster an ecosystem of solutions which makes the extension of battery life attractive to 3rd parties. Some components of these solutions that manage to achieve just that and need to be given special attention, as they directly influence our system design and implementation, include the embedded sensors that provide data from the battery system (BS), advanced State-of-X algorithms that provide key analytics for monitoring and assessments, and a new Electrochemical Impedance Spectroscopy (EIS) implementation (which is integrated straight into the BMS) that provides some key data about the state of the BS.

We observe that all aspects of the project share a fundamental requirement: data. Consequently, one of the key elements that will fast-track our vision is to work towards holistic and flexible data formats and models. Moreover, the storage of the data produced by the aforementioned processes and logistical management of all the levels of abstraction of a battery ecosystem (from a single cell to a whole module) should become an intuitive and seamless process. Finally, the importance of well-defined communication protocols for the coherent interconnection of the systems that will implement the solutions described above becomes apparent and is of vital importance.

The Battery2Life Cloud Platform comes to fill in that gap by delivering an open and adaptable solution to potential beneficiaries. Special focus is given on precisely defining frameworks and their characteristics. Effective operation and communication with third party systems is key. This, in turn, will enhance our ability to unambiguously get a better understanding of the elements that make up the model of our system and precisely pinpoint areas of improvement that further help the monitoring and analytics of these systems.

Now we will offer an overview of the way that our application will satisfy these requirements. Firstly, and most importantly, by taking into account the existing battery passport models. our app will formulate a well-defined yet flexible data model. Secondly, our system will provide a User Interface (UI) for the bookkeeping of the battery systems. Thirdly, our

system will provide a RESTful API with several endpoints to be used by algorithms for the analytics of the system as well as endpoints for the retrieval of data of EIS experiments and real time measurements. Lastly, the system will provide a publish-subscribe communication functionality for the insertion of data to the Data Management System (implements the DMS).

For the scope of this diploma thesis, we will focus on the implementation of the backend aspect of the system for displaying and updating information to third-party users. Next, we will develop the backend system for storing data to a DMS of our fabrication. These data will come as a result of EIS experiments and live timing measurement of system characteristics. Additionally, we will ensure the implementation of user management, authorization, and authentication functionalities that will be utilized by the User Interface (UI). Finally, all of these systems will be implemented, and the appropriate tools will be selected according to the specifications and requirements set by the project's earlier work packages.

Chapter 2

System Model

In this chapter, we are going to get into the specifics of the data management system where we present the schema of the database as well as the rationale behind it. Then we are going to continue with the demonstration of the endpoints structure including the expected behavior and examples. We are going to finish by providing a high-level overview of the architecture of our system and its components linking it with the use cases for our cloud platform.

2.1 Data Management System

2.1.1 Digital Battery Passport

The Digital Battery Passport is defined as an electronic record of an individual battery [2] (Article 77) and essentially acts as a digital identity. The European Union introduced the concept of the Digital Battery Passport as part of its new Batteries Regulation, which was adopted in July 2023[3]. This regulation mandates that, starting from February 18, 2027, all light means of transport (LMT) batteries, industrial batteries with a capacity greater than 2kWh, and electric vehicle batteries placed on the EU market must have an electronic record known as a "battery passport". Therefore, the need to accurately define the final blueprint of the DBP becomes of vital importance and is a work in progress.

The Battery Pass Consortium(BPC)[1] is actively working on developing the necessary frameworks and technical standards for the implementation of the battery passport. The BPC passport content guidance provides us with a wealth of useful information for the design of the data schema. In particular, the data can be divided into static and dynamic categories[4], reflecting how frequently they are updated. Static data are updated infrequently, while dynamic data are updated more regularly. In addition, essential attributes such as battery and manufacturer information, compliance and carbon footprint, battery composition, performance, and durability are systematically detailed and utilized in our data model.

The battery passport is designed to enhance transparency and sustainability throughout the battery value chain. It will document a battery's entire lifecycle—from raw material extraction and production to use, reuse, and recycling—by recording data such as carbon footprint, material composition, information pertinent to recycling/ repurposing and more. This directive was introduced in order to enhance the sustainability, traceability and accountability in the battery sector.

The description of the DBP though doesn't come without its restrictions and challenges. As described in the DBP value assessment[5] several setbacks and obstacles could include:

- The complexity and scope of the data leads to difficulties in collecting, validating and managing large amounts of data
- The integration of the DBP into existing systems will be a tough challenge due to the variety in battery management systems
- Coordinating the various stakeholders involved, given the diverse roles and responsibilities they hold across the battery value chain
- The additional cost and resource allocation that are required for its implementation
- The intricate nature of data systems needed to handle the extensive data volume, while meeting all specifications

Additionally, the standardized data model for the battery passport[6] offers valuable insights for the structure of our model. Specifically, it outlines a tree structure of nodes consisting of battery attributes, manufacturing information, usage attributes, and environmental impact attributes. Furthermore, it includes part nodes that provide detailed insights into the various components of a battery. The key takeaways of this model are, the modularity and flexibility that the abstraction of nodes imparts as well as some basic attributes our model could exploit regarding the use and performance.

The precise framework and technical details of the DBP are still under development. The

[1]The Battery Pass Consortium is a collaborative effort involving multiple stakeholders, including companies like BMW, AUDI, and Umicore. It is co-funded by the German Federal Ministry for Economic Affairs and Climate Action

Battery Pass Consortium, as well as several 3rd party research, are working on technical specifications and testing systems, which are anticipated to be finalized by the end of 2025 to ensure readiness for implementation in 2027. The aims and objectives of the current effort to describe the DBP though, are in line with the aims and objectives of the B2L project. More specifically, there is a need to enhance reusability, serve the ever-increasing demands for the motive power delivery and give consistent, standardized and modular frameworks for the monitoring and assessment of second life battery systems. The DBP provides well-organized, crucial information to stakeholders in the battery value chain to support these objectives. As a result, the assimilation of the DBP to our project (even at its early stage) by utilizing it to define data formats that are as uniform and meaningful according to the use cases and specifications of our cloud-platform (that have been extensively described in section 2.3), is projected to have significant impact and will foster our goals to provide an open and interoperable second life battery management system.

2.1.2 Database Design Logic

The goals of our data management system, as far as the data structure model is concerned, is to provide a standardized framework of structured data that minimizes data volume, finds the right balance of data granularity, facilitates easy and real time data retrieval (for all the stakeholders of the value chain) and provides flexibility for future extensions.

On the other hand, the goals of our data management system as a whole is to render communication with any kind of 3rd party system feasible and efficient, ensure the security and access control of the data, streamline data integrity, control backup and failure recovery protocols and ensure high percentage of service availability.

For all the reasons that are mentioned above the use of a Database Management System (DBMS) is the most fitting solution. More on the specifics of it will be laid out in Chapter 3, where we describe the implementation of our solution. Following this, we introduce the data model we developed based on the specified requirements mentioned above and provide an explanation of our rationale behind the crucial data design decisions.

Data Model

The developed data model schema is represented by the relational diagram below. Marked as "PK" are the primary keys of each entity.

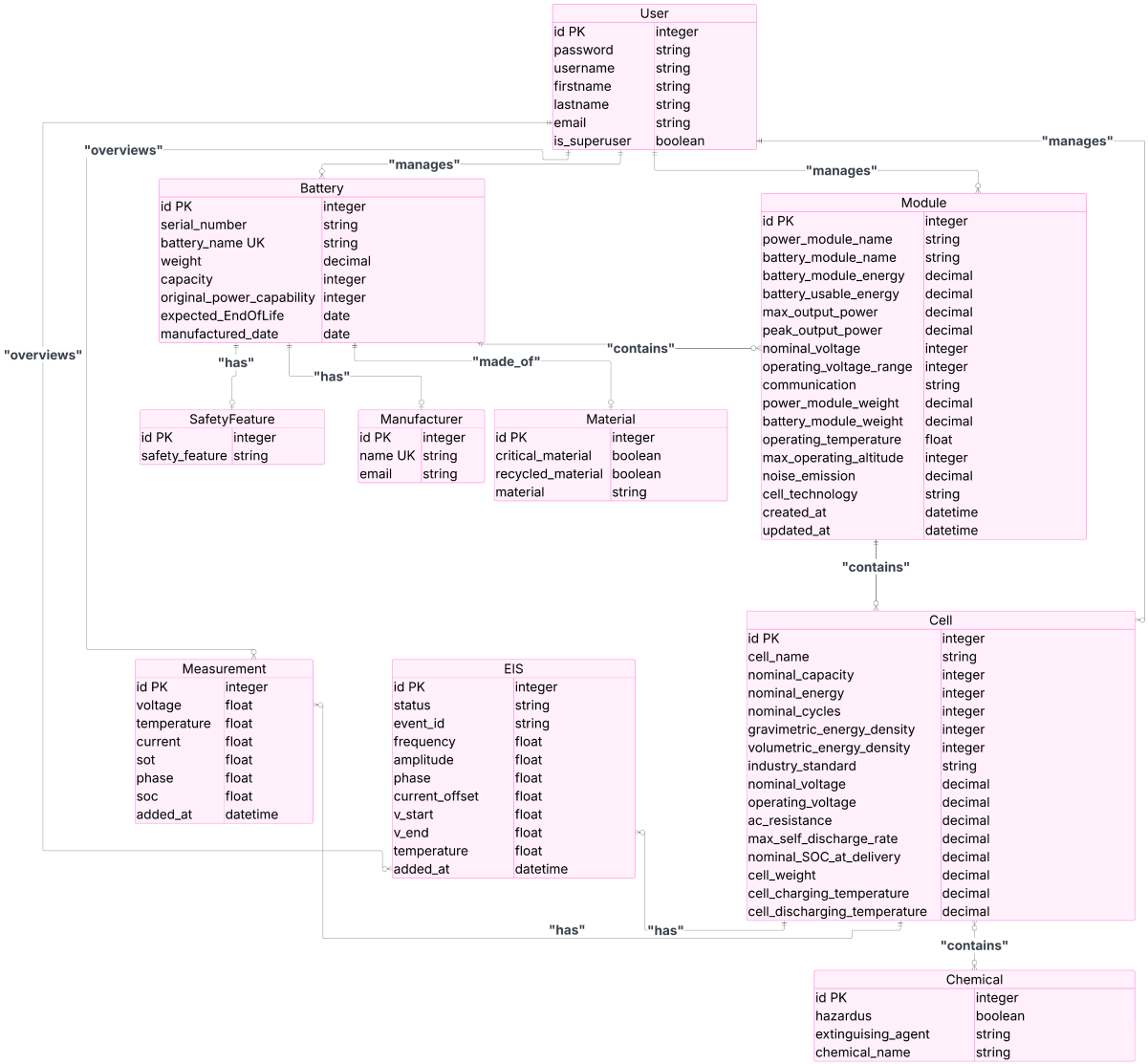


Figure 2.1: Database Entity-Relationship Diagram

Table logic

Now follows a short description and explanation of the design logic and functionality of each table.

User:
A User table will be necessary in order to ensure user authentication. Having groups of users with different privileges will help us give specific authorization to different parts of the data. As a result, the system administrator will have exclusive authority over data monitoring and modification rights, therefore maintaining data privacy and consistency.

Battery:
This is the highest level of abstraction of our power module. This table contains aggregate information such as the weight, dimensions and capacity of the battery. Also, it provides key values regarding its identification, material composition and manufacturing information which are essential for the correct monitoring of the Battery System.

Module:
Then we have the model for the individual 2nd hand battery modules that comprise the battery pack. In this model we identify a wide variety of attributes for the identification and management of the component and then most attributes regarding its operation.

Cell:
This table focuses on the most fundamental level of abstraction for a functional battery: the cell. Cells are the basic building blocks, which are grouped together to form modules, and multiple modules are combined to create a complete battery. The model primarily includes attributes that define the specifications of the cell, such as *nominal_energy* and *volumetric_energy_density*. Additionally, it incorporates attributes related to the physical dimensions of the cell and operational characteristics, such as *operating_voltage*.

In our work we are following a modular approach, as suggested in the *Standardized Data Model for the Battery Passport*[6], we will try to make separate tables that contain all the attributes that are common from the previous relationships.

Manufacturer:
To enhance the recycling and traceback processes, a dedicated manufacturer table was created to store essential information. The manufacturer table is optional and can be utilized by the model's Battery, Module and Cell.

Chemical:
This table contains information for the chemicals used by the battery module as a whole and contains attributes like information about hazardous substances, type of chemical and extinguishing agent.

SafetyFeature:
The Battery relationship leverages this connection to store critical safety insights specific to each battery pack.

Material:
This relationship is also leveraged by the battery model to provide users with a comprehensive overview of the materials used in their batteries, including details about their state—such as whether they are recycled or pose any potential hazards[2].

Our data management system also accommodates the storage and retrieval of information generated through experiments and live measurements. This includes enabling the seamless insertion of experimental data and ensuring efficient access for analysis and further use.

Measurement:
This model defines a structure for the insertion of real time data from a battery cell like *voltage*, *temperature* and *state_of_charge*.

EIS:
This model defines a data structure to insert Equivalent Impedance Spectroscopy (EIS) data into the system. Attributes include *status*, *frequency*, *amplitude*, *temperature* and others.

The exact attributes for each table were selected from the technical specification for the EU battery passport[7] where more information regarding the description of the fields can be found.

[2]These hazards include environmental contamination and human poisoning

2.2 REST API Endpoint Structure

Our Application Programming Interface(API) is designed to serve as an interoperable platform for managing and analyzing Battery Management System (BMS) data. It is structured into two core functional categories: Analytics Endpoints and Battery Management Endpoints.

The Analytics Endpoints are responsible for retrieving real-time data and experimental results, as well as storing this information in the database for further analysis. The structure of those endpoints, is illustrated by a specification document we received about the operation of the onboard BS Wi-Fi modules [8]. On the other hand, the Battery Management Endpoints focus on logging and managing operational data, ensuring all relevant information is captured in alignment with the Data Model outlined in the previous section.

In our system, user authentication/authorization is required to access the API resources at all times. So, an authentication mechanism and an inclusion of the produced authentication token is demanded to grant access rights to the API. All the objects in the examples shown below conform to the data model design.

In the following sections, we will provide a detailed description of each endpoint, including its purpose, input parameters/body, output and expected behavior with example requests/responses.

2.2.1 Analytics

EIS

API Endpoint

GET `/api/eis/`

Description: Retrieves all the available information about the requested model.

Path Parameters:

- No parameters required

Response: Returns an array of objects for the required resource, with HTTP status code 200 (OK). If no objects are found it returns an empty array, also with HTTP status code 200 (OK).

Example Request:

```
HEADERS

GET /api/eis/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Authorization: Bearer <token>
Content-Type: application/json
```

Example Response:

```
HEADERS:

allow: GET,POST,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
[
    resource_object1,
    resource_object2,
            ...       ,
    resource_objectN
]
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

API Endpoint

POST /api/eis/

Description: Creates a new database entry for the required resource according the values of the body of the request.

Path Parameters:

- No parameters required

Response: Returns an object with HTTP status code 201 (Created)

Example Request:

```
HEADERS:

POST /api/eis/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```


```
{
    "status": "success",
    "event_id": "207_1",
    "cell_id": 1,
    "frequency": 25.11886432,
    "amplitude": 0.00033255133178403194,
    "phase": 0.085914587975807,
    "current_offset": 2,
    "v_start": 3.5,
    "v_end": 3.5,
    "temperature": 25
}
```


Example Response:

```
HEADERS:

allow: GET,POST,HEAD,OPTIONS
connection: keep-alive
content-length: 149552
content-type: application/json
cross-origin-opener-policy: same-origin
date: Fri,21 Feb 2025 11:07:01 GMT
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY

STATUS CODE:
201
```


```
{
  "id": 1001,
  "status": "success",
  "event_id": "207_1",
  "frequency": 25.11886432,
  "amplitude": 0.00033255133178403194,
```

```
            "phase": 0.085914587975807,
            "current_offset": 2,
            "v_start": 3.5,
            "v_end": 3.5,
            "temperature": 25,
            "added_at": "2025-02-26T07:48:08.704086Z",
            "cell_id": 1
        }
```


Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

GET `/api/eis/{id}/`
Description: Retrieves a specifc object from the requested resource.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested resource object, with HTTP status code 200 (OK)

Example Request:

```
HEADERS

GET /api/eis/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value2",
            ...
    "fieldN": "valueN",
}
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PUT `/api/eis/{id}/`
Description: Replaces the required database entry, with specific id, with the information provided by the body JSON object. Doesn't replace the optional ommited fields but instead replaces the values that sees that has changed.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the request resource changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PUT /api/eis/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "cell_id": 1,
    "status": "fail",
    "event_id": "207223_1",
    "frequency": 0,
    "amplitude": 0,
    "phase": 0,
    "current_offset": 0,
    "v_start": 0,
    "v_end": 0,
    "temperature": 0
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
```

```
connection: keep-alive
content-length: 1000
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```


```
            {
              "id": 1001,
              "status": "fail",
              "event_id": "207223_1",
              "frequency": 0,
              "amplitude": 0,
              "phase": 0,
              "current_offset": 0,
              "v_start": 0,
              "v_end": 0,
              "temperature": 0,
              "added_at": "2025-02-26T07:48:08.704086Z",
              "cell_id": 1
            }
```


Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PATCH `/api/eis/{id}/`

Description: Provides some fields to be changed for a specific database entry. The rest of the fields of this entry are not changed in any way. Request body contains only the fields that we want to change and no more.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PATCH /api/eis/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
            {
                "cell_id": 1,
                "status": "success",
                "event_id": "207_1",
                "v_end": 3.5,
                "temperature": 25
            }
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 1000
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
            {
              "id": 1001,
              "status": "success",
              "event_id": "207_1",
              "frequency": 0,
              "amplitude": 0,
              "phase": 0,
              "current_offset": 0,
              "v_start": 0,
              "v_end": 3.5,
              "temperature": 25,
              "added_at": "2025-02-26T07:48:08.704086Z",
              "cell_id": 1
            }
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

DELETE `/api/eis/{id}/`
Description: Deletes the database entry with the id given by the parameter id in the url.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns HTTP status code 204 (No Content) if successful.

Example Request:

```
HEADERS

DELETE /api/eis/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 0
cross-origin-opener-policy: same-origin
date:  Wed,26 Feb 2025 07:59:08 GMT
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

Measurements

API Endpoint

GET `/api/measurements/`

Description: Retrieves all of the available information about the requested model.

Path Parameters:

- No parameters required

Response: Returns an array of objects for the required resource, with HTTP status code 200 (OK). If no objects are found it returns an empty array, also with HTTP status code 200 (OK).

Example Request:

```
HEADERS

GET /api/measurements/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,POST,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
[
    resource_object1,
    resource_object2,
            ...       ,
    resource_objectN
]
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

POST `/api/measurements/`

Description: Creates a new database entry for the required resource according the values of the body of the request.

Path Parameters:

- No parameters required

Response: Returns an object with HTTP status code 201 (Created)

Example Request:

```
HEADERS

POST /api/measurements/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "cell_ids": [1, 2, 3, 4],
    "voltage": [3.934999943, 3.950999975,
        3.816999912, 3.948999882],
    "temperature": [18.5, 19.39999962,
        17.29999924, 16.60000038],
    "sot": [1, 0, 1, 1],
    "phase": [ 45, 21, 211, 0],
    "current": [0, 0, 0, 0],
    "soc": [69.66569519, 70.92323303,
        59.74949265, 70.6905365]
}
```

Example Response:

```
HEADERS

allow: GET,POST,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY

STATUS CODE:
200
```

```
{
  "cell_ids": [
    1,
    2,
    3,
    4
  ],
  "voltage": [
    3.934999943,
    3.950999975,
    3.816999912,
```

```
            3.948999882
          ],
          "temperature": [
            18.5,
            19.39999962,
            17.29999924,
            16.60000038
          ],
          "current": [
            0,
            0,
            0,
            0
          ],
          "sot": [
            1,
            0,
            1,
            1
          ],
          "phase": [
            45,
            21,
            211,
            0
          ],
          "soc": [
            69.66569519,
            70.92323303,
            59.74949265,
            70.6905365
          ]
        }
```


Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

GET `/api/measurements/{id}/`
Description: Retrieves a specifc object from the requested resource.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested resource object, with HTTP status code 200 (OK)

Example Request:

```
HEADERS

GET /api/measurements/{id}/ HTTP/1.1
```

```
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value2",
            ...
    "fieldN": "valueN",
}
```

Possible Errors:

- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PUT `/api/measurements/{id}/`

Description: Replaces the required database entry, with specific id, with the information provided by the body JSON object. Doesn't replace the optional ommited fields but instead replaces the values that sees that has changed.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the request resource changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PUT /api/measurements/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
  "voltage": 0,
  "temperature": 0,
  "current": 0,
  "sot": 0,
  "phase": 0,
  "soc": 0,
  "cell_id": 42
}
```


Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY

STATUS CODE:
200
```


```
{
  "id": 1001,
  "voltage": 0,
  "temperature": 0,
  "current": 0,
  "sot": 0,
  "phase": 0,
  "soc": 0,
  "added_at": "2025-02-21T11:16:37.051142Z",
  "cell_id": 42
}
```


Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PATCH `/api/measurements/{id}/`

Description: Provides some fields to be changed for a specific database entry. The rest of the fields of this entry are not changed in any way. Request body contains only the fields that we want to change and no more.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PATCH /api/measurements/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}

STATUS CODE:
200
```

```
{
  "cell_id": 3
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
  "id": 1001,
  "voltage": 0,
  "temperature": 0,
  "current": 0,
  "sot": 0,
  "phase": 0,
  "soc": 0,
  "added_at": "2025-02-21T11:16:37.051142Z",
  "cell_id": 3
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

DELETE `/api/measurements/{id}/`

Description: Deletes the database entry with the id given by the parameter id in the url.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns HTTP status code 204 (No Content) if successful.

Example Request:

```
HEADERS

DELETE /api/measurements/1001/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 0
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY

STATUS CODE:
204
```

Possible Errors:

- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

2.2.2 Battery Management

Battery

API Endpoint

GET `/api/batteries/`

Description: Retrieves all of the available information about the requested model.

Path Parameters:

- No parameters required

Response: Returns an array of objects for the required resource, with HTTP status code 200 (OK). If no objects are found it returns an empty array, also with HTTP status code 200 (OK).

Example Request:

```
HEADERS

GET /api/batteries/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr
Authorization: Bearer {token}
```

Example Response:

```
 HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
[
    resource_object1,
    resource_object2,
            ...       ,
    resource_objectN
]
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

API Endpoint

POST /api/batteries/

Description: Creates a new database entry for the required resource according the values of the body of the request.

Path Parameters:

- No parameters required

Response: Returns an object with HTTP status code 201 (Created)

Example Request:

HEADERS

```
POST /api/batteries/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Example Response:

HEADERS

```
access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found

- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

GET /api/batteries/{id}/
Description: Retrieves a specifc object from the requested resource.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested resource object, with HTTP status code 200 (OK)

Example Request:

```
HEADERS

GET /api/batteries/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value2",
            ...
    "fieldN": "valueN",
}
```

Possible Errors:

- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PUT /api/batteries/{id}/

Description: Replaces the required database entry, with specific id, with the information provided by the body JSON object. Doesn't replace the optional ommited fields but instead replaces the values that sees that has changed.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the request resource changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PUT /api/batteries/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required

- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PATCH `/api/batteries/{id}/`

Description: Provides some fields to be changed for a specific database entry. The rest of the fields of this entry are not changed in any way. Request body contains only the fields that we want to change and no more.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PATCH /api/batteries/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "selected_field1": "value1",
    "selected_field2": "value1",
            ...
    "selected_fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

DELETE `/api/batteries/{id}/`
Description: Deletes the database entry with the id given by the parameter id in the url.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns HTTP status code 204 (No Content) if successful.

Example Request:

```
HEADERS

DELETE /api/batteries/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 0
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

Possible Errors:

- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

Module

API Endpoint

GET `/api/modules/`

Description: Retrieves all of the available information about the requested model.

Path Parameters:

- No parameters required

Response: Returns an array of objects for the required resource, with HTTP status code 200 (OK). If no objects are found it returns an empty array, also with HTTP status code 200 (OK).

Example Request:

```
HEADERS

GET /api/modules/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr
Authorization: Bearer {token}
```

Example Response:

```
[
    resource_object1,
    resource_object2,
            ...       ,
    resource_objectN
]
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

POST `/api/modules/`

Description: Creates a new database entry for the required resource according the values of the body of the request.

Path Parameters:

- No parameters required

Response: Returns an object with HTTP status code 201 (Created)

Example Request:

```
HEADERS

POST /api/modules/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
                {
                    "field1": "value1",
                    "field2": "value1",
                            ...
                    "fieldN": "value1",
                }
```

Example Response:

```
            {
                "field1": "value1",
                "field2": "value1",
                        ...
                "fieldN": "value1",
            }
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

GET `/api/modules/{id}/`
Description: Retrieves a specifc object from the requested resource.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested resource object, with HTTP status code 200 (OK)

Example Request:

```
HEADERS

GET /api/modules/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
```

```
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value2",
            ...
    "fieldN": "valueN",
}
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PUT `/api/modules/{id}/`

Description: Replaces the required database entry, with specific id, with the information provided by the body JSON object. Doesn't replace the optional ommited fields but instead replaces the values that sees that has changed.

Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the request resource changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PUT /api/modules/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
```

```
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PATCH `/api/modules/{id}/`
Description: Provides some fields to be changed for a specific database entry. The rest of the fields of this entry are not changed in any way. Request body contains only the fields that we want to change and no more.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PATCH /api/modules/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "selected_field1": "value1",
    "selected_field2": "value1",
            ...
    "selected_fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

DELETE `/api/modules/{id}/`
Description: Deletes the database entry with the id given by the parameter id in the url.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns HTTP status code 204 (No Content) if successful.

Example Request:

```
HEADERS

DELETE /api/modules/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

API Endpoint

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 0
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

Cell

API Endpoint

GET `/api/cells/`

Description: Retrieves all of the available information about the requested model.

Path Parameters:

- No parameters required

Response: Returns an array of objects for the required resource, with HTTP status code 200 (OK). If no objects are found it returns an empty array, also with HTTP status code 200 (OK).

Example Request:

```
HEADERS

GET /api/cells/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr
Authorization: Bearer {token}
```

Example Response:

```
[
    resource_object1,
    resource_object2,
            ...       ,
    resource_objectN
]
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

POST `/api/cells/`

Description: Creates a new database entry for the required resource according the values of the body of the request.

Path Parameters:

- No parameters required

Response: Returns an object with HTTP status code 201 (Created)

Example Request:

```
HEADERS

POST /api/cells/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
                {
                    "field1": "value1",
                    "field2": "value1",
                            ...
                    "fieldN": "value1",
                }
```

Example Response:

```
            {
                "field1": "value1",
                "field2": "value1",
                        ...
                "fieldN": "value1",
            }
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

GET `/api/cells/{id}/`
Description: Retrieves a specifc object from the requested resource.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested resource object, with HTTP status code 200 (OK)

Example Request:

```
HEADERS

GET /api/cells/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

```
HEADERS

allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
```

```
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value2",
            ...
    "fieldN": "valueN",
}
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PUT `/api/cells/{id}/`
Description: Replaces the required database entry, with specific id, with the information provided by the body JSON object. Doesn't replace the optional ommited fields but instead replaces the values that sees that has changed.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the request resource changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PUT /api/cells/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
```

```
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

PATCH `/api/cells/{id}/`
Description: Provides some fields to be changed for a specific database entry. The rest of the fields of this entry are not changed in any way. Request body contains only the fields that we want to change and no more.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns the requested changed object that is saved to the database, with HTTP status code 200 (OK) or 204 (No Content).

Example Request:

```
HEADERS

PATCH /api/cells/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

```
{
    "selected_field1": "value1",
    "selected_field2": "value1",
            ...
    "selected_fieldN": "value1",
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "field1": "value1",
    "field2": "value1",
            ...
    "fieldN": "value1",
}
```

Possible Errors:

- 400 — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- 401 — Unauthorized: Authentication required
- 403 — Forbidden: Insufficient permissions
- 404 — Not Found: Resource not found
- 500 — Internal Server Error: A generic server error occured
- 503 — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

DELETE `/api/cells/{id}/`
Description: Deletes the database entry with the id given by the parameter id in the url.
Path Parameters:

- id (required:Integer) – Unique identifier for the selected data registration

Response: Returns HTTP status code 204 (No Content) if successful.

Example Request:

```
HEADERS

DELETE /api/cells/{id}/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
Accept: application/json, text/html
Authorization: Bearer {token}
```

Example Response:

API Endpoint

```
HEADERS

access-control-allow-origin: *
allow: GET,PUT,PATCH,DELETE,HEAD,OPTIONS
connection: keep-alive
content-length: 0
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

Possible Errors:

- `401` — Unauthorized: Authentication required
- `403` — Forbidden: Insufficient permissions
- `404` — Not Found: Resource not found
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

2.2.3 Authentication

Token

API Endpoint

POST `/api/token/`

Description: Makes a request with authentication credentials and gets back a JWT(JavaScript Web Token). Essential for accessing all the other endpoints

Path Parameters:

- No parameters required

Response: Returns a user object with the JWT authentication token and refresh token. HTTP status code is 200 (OK) if successful.

Example Request:

```
HEADERS

POST /api/token/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
accept: application/json
Content-Type: application/json
X-CSRFTOKEN: CSRF-TOKEN
```

```
{
    "username": username:String,
    "password": password:String
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: POST,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "refresh": refreshToken:String,
    "access": accessToeken:String
}
```

Possible Errors:

- `400` — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- `401` — Unauthorized: Authentication required

API Endpoint

- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

API Endpoint

POST `/api/token/refresh/`

Description: Makes a request with the refresh token and gets back a valid JWT.

Path Parameters:

- No parameters required

Response: Returns a JSON object with key:access and value:newValidToken(string) and HTTP status code 200 (OK) when successful.

Example Request:

```
HEADERS

POST /api/token/refresh/ HTTP/1.1
Host: https://dev-battery2life.iccs.gr/
accept: application/json
Content-Type: application/json
X-CSRFTOKEN: CSRF-TOKEN
```

```
{
    "refresh": refreshToken:String
}
```

Example Response:

```
HEADERS

access-control-allow-origin: *
allow: POST,OPTIONS
connection: keep-alive
content-length: Integer
content-type: application/json
cross-origin-opener-policy: same-origin
date: Date
referrer-policy: same-origin
server: nginx/1.18.0 (Ubuntu)
vary: Accept,origin
x-content-type-options: nosniff
x-frame-options: DENY
```

```
{
    "access": accessToken:String
}
```

Possible Errors:

- `400` — Bad Request: Request structure is wrong (e.g. invalid JSON structure)
- `401` — Unauthorized: Authentication required
- `500` — Internal Server Error: A generic server error occured
- `503` — Service Unavailable: The server is temporarily unable to handle the request (e.g., due to maintenance or overload)

The endpoints chemical, material, manufacturers and safety_feature have the same functionality with the batteries endpoint and therefore are ommited for the sake of simplicity and concisesness

2.2.4 Endpoints Summary Table

Table 2.1: API Endpoints Summary

Category	**Endpoint**	**Methods**
EIS	`/api/eis/`	GET, POST
	`/api/eis/{id}/`	GET, PUT, PATCH, DELETE
Measurements	`/api/measurements/`	GET, POST
	`/api/measurements/{id}/`	GET, PUT, PATCH, DELETE
Authentication	`/api/token/`	POST
	`/api/token/refresh/`	POST
Battery Management	`/api/batteries/`	GET, POST
	`/api/batteries/{id}/`	GET, PUT, PATCH, DELETE
Cell Management	`/api/cells/`	GET, POST
	`/api/cells/{id}/`	GET, PUT, PATCH, DELETE
Chemical	`/api/chemical/`	GET, POST
	`/api/chemical/{id}/`	GET, PUT, PATCH, DELETE
Manufacturer	`/api/manufacturers/`	GET, POST
	`/api/manufacturers/{id}/`	GET, PUT, PATCH, DELETE
Material	`/api/material/`	GET, POST
	`/api/material/{id}/`	GET, PUT, PATCH, DELETE
Module	`/api/modules/`	GET, POST
	`/api/modules/{id}/`	GET, PUT, PATCH, DELETE
Safety Features	`/api/safety_feature/`	GET, POST
	`/api/safety_feature/{id}/`	GET, PUT, PATCH, DELETE

2.3 System Architecture

In this section, we aim to outline the main architectural concepts of our system, providing readers with a deeper insight into the project and the rationale behind our design choices. We will begin by detailing our platform's requirements. Then, we'll highlight the high-level components that will facilitate the achievement of our goals. Finally, we will dive deeper into the usage goals of our system.

2.3.1 System Requirements

Our platform is built on an array of requirements, as defined in the deliverable 2.3 of the Battery2Life project[9], that can be categorized into the following types: functional[3], non-functional[4] and data management requirements of our cloud platform.

1. We need to provide a working system that is as open as possible. In order to achieve this we integrate open standards. Then, we need to ensure the vendor neutrality for our API and communication protocols; therefore, using open-source tools will be paramount for this effort. Also, we need to make sure for our app's interoperable capabilities by implementing widely used and mature communication protocols. Our efforts for a holistic solution will be enhanced by integrating the Digital Battery Passport to our platform that will in turn provide all the necessary data and in the right format for the effective monitoring of the Battery System(BS) and traceability for its components.

2. Our platform needs to be developed in a way that will make the scalability aspect of our project an easy task. The concurrent devices connected and data streams are key parameters to consider in this effort. At the same time we need to cater for the high fault tolerance of the platform by implementing load balancing and error detection/correction strategies, that will in turn ensure high availability of the system.

3. Regarding the data, our platform must ensure their privacy by utilizing authentication and authorization mechanisms for the endpoints. Then we need to emphasize on establishing a secure environment that is impervious to outside attacks by implementing functionalities for the detection and mitigation of malicious actions. Last but not least, we need to take into account the integrity of the data by checking that all types and values are correct and the database is up-to-date with all the available data.

[3] functional: requirements related to the specific low level functions of our system
[4] non-functional: requirements related to the operation of the system as a whole

2.3.2 System Components

We will present the component diagram of our cloud platform as was designed per the specifications of the previous section and then we will delve into the details of every component:

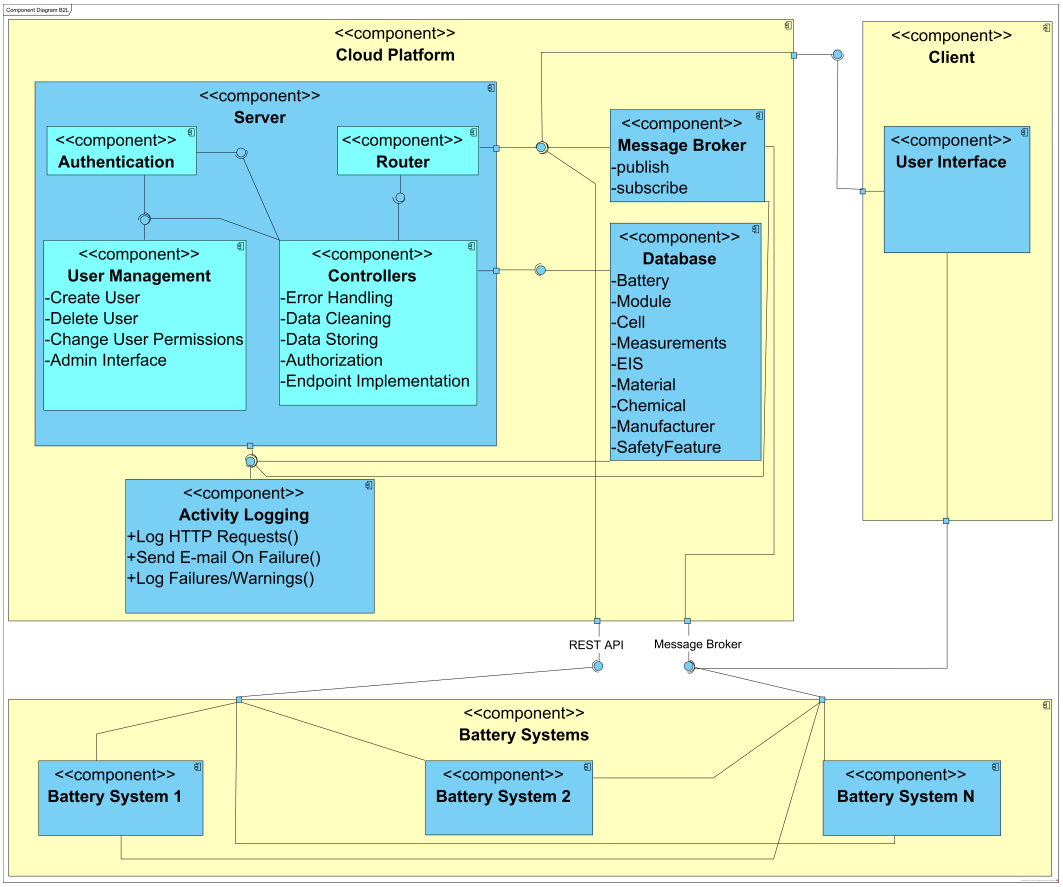


Figure 2.2: Battery2Life component diagram

Architecture

We chose a monolithic architecture for our application (client-server) due to the limited number of services, avoiding unnecessary complexity[10]. This architecture ensures efficient communication within the system's integrated components, minimizing latency and overhead compared to distributed systems. Additionally, it offers reduced operational costs, easier maintenance and faster development cycle. The ability to respond to big loads of requests isn't hindered at all, since scaling the system horizontally with more servers will be an easy task.

Communication

Two methods are going to be used for the communication of our app with 3rd party applications:

1. A publish-subscribe module will be implemented, where the server is going to be subscribed into a topic and get requests like this. This is very useful for maintaining data

consistency when the server is down, because the messaging broker stores the unprocessed requests and the application parses them when it resuscitates.

2. A RESTful[5] API with the specific endpoints described on Section 2.2 will be implemented. Therefore, any kind of application connected to the internet can pose HTTP(GET, POST, GET id, PUT id, PATCH id, DELETE id) requests by sending, receiving and editing the data that they have the right to take that specific action.

Authentication Module

The authentication module will include a way for the user to provide its credentials and get a perishable authentication token as a response, which can then by used in the headers of the HTTP request to gain access to our app resources. The frequent recycling of the authentication token will further enhance the security of our platform.

User Management Module

The User Management module handles the creation and management of users with different kind of permissions.

Database Module

The database module is the service that is responsible for the storage and management of the data. A database with a robust database management system (DBMS) is essential for our application. Especially since data type integrity/consistency, data access, efficient data processing and recovery mechanisms are all integrated into it.

Activity Logging

This module is used by the server, database and message broker modules. It is significant for the effective troubleshooting of system failures that occurred, or potential future failures, through the error/warning logs. Also, this module sends an email notifying the administrators that a service is down, contributing in the swift resolution of an error.

Server Module

This is the most basic module that encompasses the authentication module, the router module, the user management module and the controller module. The server is able to receive requests for getting, creating, deleting and modifying data from an outside client, message broker or REST API through an HTTP request. The router handles which controller needs to process the request. The user management and authentication modules are used when there is a request demanding the use of resources with restricted access. Additionally, the user management module is responsible for the creation/deletion of users and the assignment of new access rights to them.

[5] An API is RESTful when it adheres to the REST(Representational State Transfer) principles[11]. Such principles include client-server architecture, statelessness, layered system architecture etc.

Client Module

This is a user interface where the user is able to inspect, add and modify all the attributes regarding the specification of the available batteries. This module sends HTTP requests to the server via the REST API then displays the result for the user in a friendly and well organized manner.

2.3.3 Use Cases

The proper design of each individual component is crucial, but it's also important to consider how they interact with each other. This can be seen by outlining the three primary use cases for our system. The use cases include:

1. Every function regarding the **management of the battery systems**. A user can utilize the user interface (UI) of the client to log into the system, fill in and edit sections with data about the characteristics of the batteries. These data could include batteries, cells, manufacturer details and other important information about the bookkeeping aspect of the system as was described in section 2.1 . The UI, Server, Activity Logging and Database modules are utilized here.

2. Every function regarding the **data ingestion in the system** contributed by 3rd parties. The battery systems share real time data and data regarding experiments with the platform through the REST API and message broker channels. Then the battery system validates these data and stores them into the database. All the actions mentioned above require some sort of authorization in order to be able to use the associated resources. The Server, Message Broker, Activity Logging and Database modules were employed in this scenario.

3. Every function regarding **data sharing**. An authorized 3rd party can request certain parts of or all the dataset, either in JavaScript Object Notation(JSON) format or presented in the UI. These data will be mainly used for running predictive algorithms and monitoring by 3rd parties.

Chapter 3

Implementation

The goal for this chapter is to give the reader a basic understanding of how the components of chapter 2 were implemented and the rationale behind the implementation choices in terms of software tools. We begin by explaining how the components were implemented, then continue by presenting some of the key tools that aided the satisfaction of the non-functional requirements of the project and in the end by providing some basic insights about the practicalities of our system.

3.1 Component Communication

Now we are going to explain the two communication protocols that our platform uses.

3.1.1 REST Endpoints

For the communication of the components the main protocol used is HTTP requests. All of our endpoints implement the GET, POST, PUT, PATCH, DELETE methods as described in the specification section. For a GET, PUT, PATCH, DELETE request a parameter named "id" is used at the end of the URL section of the request in order to conduct the necessary activities for the specific request. For the POST, PUT and PATCH methods a body section is required, containing the information that we want to either insert into the database either modify the values of a specific row in a table or tables. For the body of the request, the data that needs to be inserted must be described in JavaScript Object Notation (JSON) format.

Although HTTP requests is the prevalent form of message transmission in our solution, many of the components are integrated as packages or tools so they implement their own communication protocols internally. For example, as we are going to see next, the authentication, user management, viewsets(controllers) and router components are all incorporated in Django and Django Rest Framework.

Some key aspects of the HTTP request to consider before developing endpoints that will implement this protocol is idempotency and safety. More specifically, by idempotency we mean the repeatability of requests like GET, PUT and DELETE without unwanted side effects and the consistency of the API. Conversely, methods like POST, PUT, PATCH, and DELETE alter the database and system state, so it's crucial to use appropriate tools to prevent both malicious and inadvertent misuse.

3.1.2 Message Broker - Mosquitto MQTT

We also implemented an alternative way of communication between clients and 3rd parties with our platform through a deployment and configuration of a Mosquitto Message Queueing Telemetry Transport (MQTT) broker.

We have created a Mosquitto broker as a docker container with the following parameters that we mount inside the container as a volume.

```
1        listener 1883
2        listener 9001
3        allow_anonymous false
4
5        persistence true
6        password_file /mosquitto/config/passwd
7        persistence_file mosquitto.db
8        persistence_location /mosquitto/data/
9        log_dest file /mosquitto/log/mosquitto.log
```

Listing 3.1: Example Mosquitto Configuration

Our application listens on ports 1883, 9001 and doesn't allow any messaging without authentication. Furthermore, this configuration allows us to persist messages so that when a service that had some downtime returns to operation, it receives all the messages that were sent to it while it was down. This functionality is paramount in order to ensure the

consistency of the dataset. Last but not least, the Mosquitto MQTT tool gives us options to encrypt the exchanged messages.

In order to validate that our configuration works, we created a Mosquitto client that runs also as a docker container. We made the configuration in order for it to subscribe in a certain topic and can receive messages successfully both from clients outside the docker network and inside it. Finally, we connected it to the Logging Activity module in order to keep a record of the status of the service and the messages received.

3.2 Server - Django/Django Rest Framework

At a high level when a user makes a request for a resource the server gets the request and then the router matches the URL with the corresponding view[1]. Then that view is responsible for utilizing the right modules to authenticate the user and check that they have the right permissions to perform the action that they wish to perform. After that the serializers take action and parse any JSON data that is included in the request. Finally, the controller performs the actions that the request mandates, returns a result and a code describing the status of the request. We will now delve into a more detailed explanation of how these processes occur.

The platform is developed with Django[12] and Django REST Framework(DRF)[13].

3.2.1 Authentication

In order to make it possible for a user to be authenticated, the user needs to make a POST request on the */api/token* endpoint with their credentials (username, password) into the body of the request. Then they will get a JSON response with two key-value pairs as shown below in figure 4.1 . Finally, the client retrieves the access token and includes it in the request headers to authenticate or users employing software to interact with our API can manually place it in the headers section of the request.


```
{
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
  eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTc0MDQ5MTk0NCwiaWF0IjoxNzQwNDA1NTQ0LCJqdG
  kiOiJhNjNmMDliYWFjZmQ0MWUwOWEyNWE5MWZiY2M3YjZjYyIsInVzZXJfaWQiOjF9.
  gwZ37E0hJRtGKbeHFT1MFaluVCqEHNoKw1dd0Pb24ZI",
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
  eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzQwNDA5MTQ0LCJpYXQiOjE3NDA0MDU1NDQsImp0aS
  I6ImZhNjQyYWQ0NWFkODQ2YzI4NjVhNWVjNzVjYTkzZWRmIiwidXNlcl9pZCI6MX0.
  q8VURkllHR6KDCgvNXMGZhpWJokKmbzFm57ISYL9flM"
}
```


Figure 3.1: JavaScript Web Token example

In order to be able to implement authentication, we used from *simplejwt*[14] module the *views.TokenObtainPairView* and *views.TokenRefrashView* and then assign the related URLs to them in our router. The access tokens are set to be valid for up to an hour and the refresh tokens for up to a day. Whenever the access tokens are invalid the user can make a POST request at */api/token/refresh* with the refresh token as payload and get a new access token as a result without providing their credentials anew.

3.2.2 Security

We enhance our app security in ways described in more detail below[15]:

- Several key configuration parameters are defined in environment variables and not hardcoded. Such as *secret_key* and database parameters.
- Host and origin restrictions are implemented. The *allowed_hosts* environment variables restricts domains which our application responds to. This functionality is further

[1]This is the piece of code that parses the request and produces a response. Stated as controller in the component diagram fig. 2.2

enhanced with the use of Cross-Site Request Forgery Protection (CSRF). Finally, Cross-Origin Resource Sharing Protection (CORS) is implemented on the production server with the *CORS_ALLOWED_ORIGINS* parameter listing the specific domains and the corsheaders module used.

- Django REST Framework authorization classes(*IsAuthenticated*, *JWTAuthentication*) enables certain users access rights to certain resources.

- The use of the SecurityMiddleware that adds specific security headers automatically.

- We have implemented password validators that check the length, commonality and inclusion of special characters of any password that a user defines and urges the user to make it stronger .

3.2.3 User Management

Django provides us with a very convenient admin interface[16] that lets us manage the user creation, management and permission assignment.

A user can log in with their credentials if they have elevated rights as shown below.

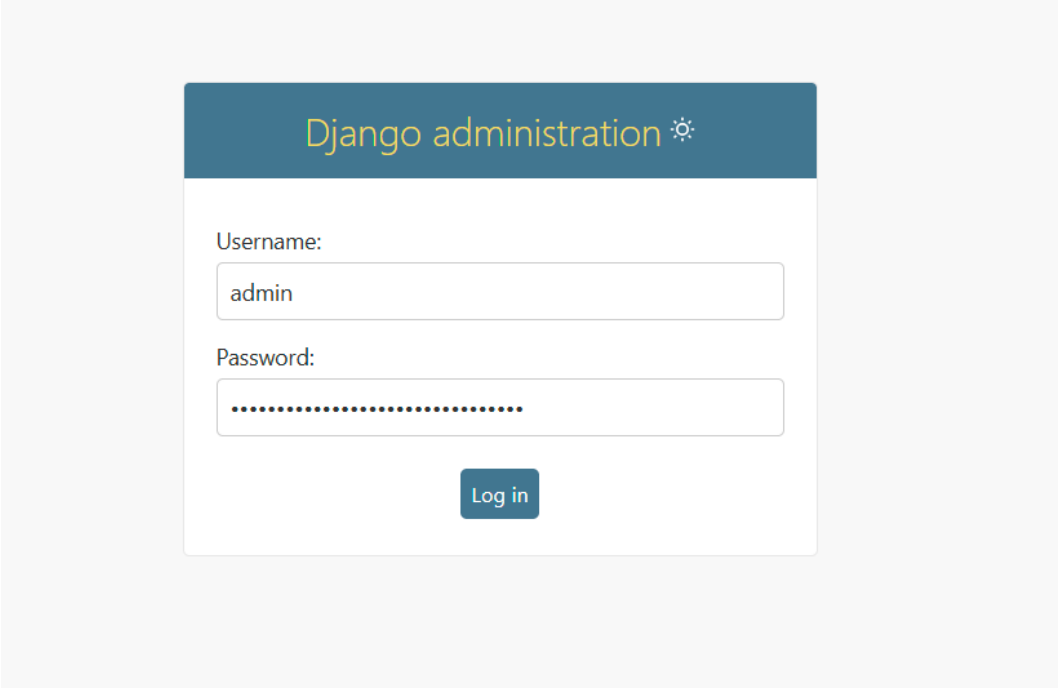


Figure 3.2: Django Admin Interface - Login

After logging in the superuser will be prompted to the main page.

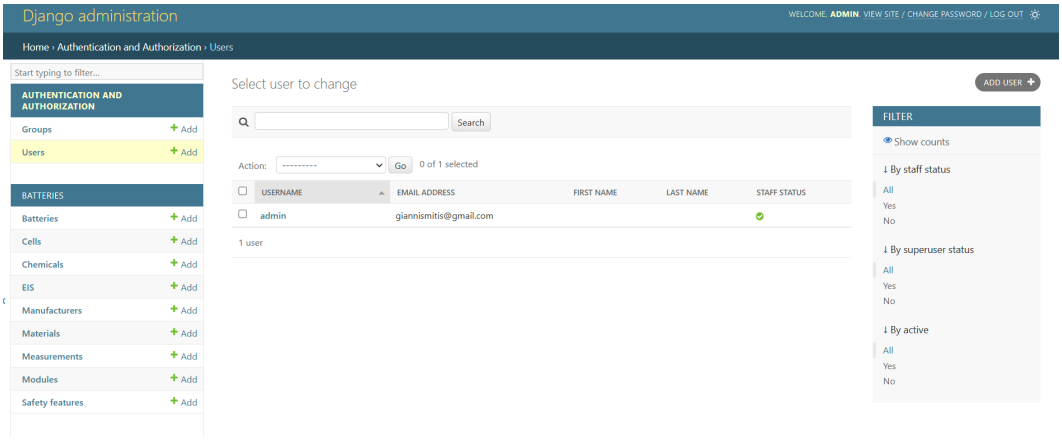


Figure 3.3: Django Admin Interface - Main Page

They can manage users or groups and assign specific permissions to them such as can add/change/view/delete model records, can add/change/view/delete user, can add/change/view/delete permission etc.

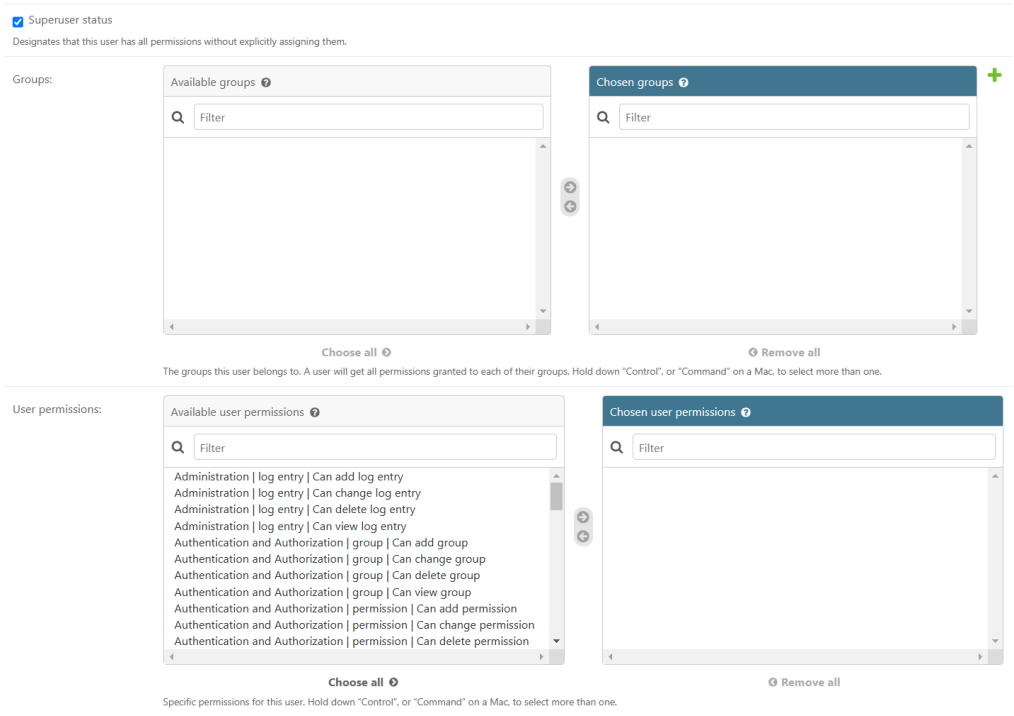


Figure 3.4: Django Admin Interface - Assign Permissions

They can also create records for the database models. We achieve this by registering these models on the *admin.py* file.

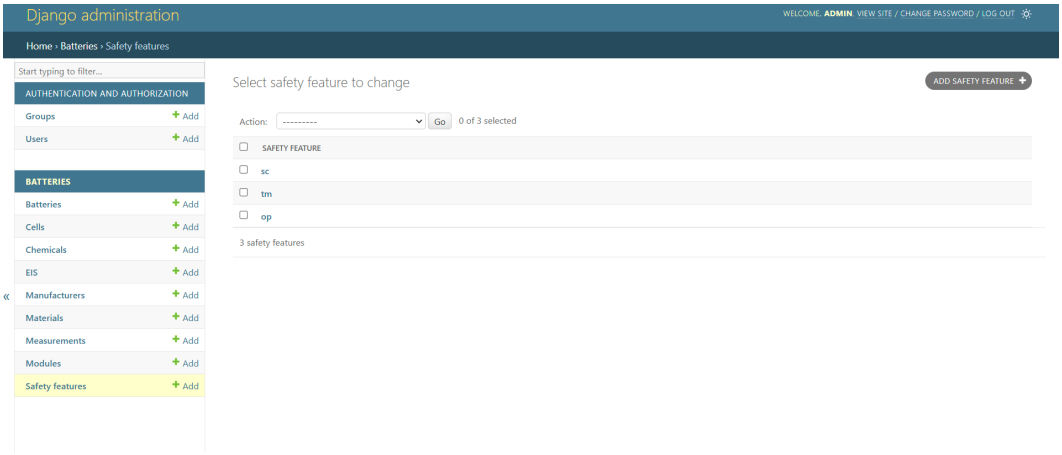


Figure 3.5: Django Admin Interface - Create Database Records

3.2.4 Router

Our platform consists of two applications, the main application which is called *battery2life* and the application that contains the API called *batteries*.

The routing functionality in the main app is seen on figure 3.6 . It includes the routes for the admin interface, authentication, SWAGGER UI documentation and the routes of the application batteries.

```
from django.contrib import admin
from django.urls import path, include
from drf_spectacular.views import SpectacularAPIView,SpectacularAPIView,
SpectacularRedocView, SpectacularSwaggerView
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/token/', TokenObtainPairView.as_view(), name="token-obtain-per-view"),
    path('api/token/refresh/', TokenRefreshView.as_view(), name="token-refresh"),
    path('api-auth', include('rest_framework.urls')),
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view
    (url_name='schema'), name='swagger-ui'),
    path('api/schema/redoc/', SpectacularRedocView.as_view(url_name='schema'),
    name='redoc'),
    path('api/', include('batteries.api.urls')),
]
```

Figure 3.6: Routing - Main App

For the main API the routing is a bit more complex, so we chose to implement it with the *DefaultRouter* from the *rest_framework module*[17] for the easier management of the endpoints. Each route is in the format *host/api/api_route*. In essence, each route defines which Django ViewSet is going to handle the specific request.

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from batteries.api.views import (ManufacturerViewSet,
                                 BatteriesViewSet, MeasurementViewSet,
                                 ModuleViewSet,
                                 CellViewSet,
                                 EISViewSet,
                                 ChemicalViewSet,
                                 SafetyFeatureViewSet,
                                 MaterialViewSet
                                 )      You, 3 months ago • Created EIS endpoint

router = DefaultRouter()
router.register(r"safety_feature", SafetyFeatureViewSet)
router.register(r"manufacturers", ManufacturerViewSet)
router.register(r"measurements", MeasurementViewSet)
router.register(r"batteries", BatteriesViewSet)
router.register(r"chemical", ChemicalViewSet)
router.register(r"material", MaterialViewSet)
router.register(r"modules", ModuleViewSet)
router.register(r"cells", CellViewSet)
router.register(r"eis", EISViewSet)
urlpatterns = [
    path("", include(router.urls))
]
```

Figure 3.7: Routing - API

3.2.5 Controllers - ViewSets

One of the main reasons we chose to develop our cloud platform with Django and Django REST Framework(DRF) is because it makes the implementation of Create, Replace, Update and Delete (CRUD) endpoints very simple and intuitive when the functionality is quite rudimentary. Faster and simpler development means much easier maintenance and less error frequency. This is ensured by utilizing a wide range of modules that are supported rigorously by the Django large community [18].

So the way that a ModelViewSet operates when a request is sent to our server is the following.

- the server parses the request
- the router directs the request to the appropriate view
- the view conducts the side effects requested[2]
- the view returns a response with a HTTP status code and a body

We have 8 views that have very basic functionality (*ChemicalViewSet, SafetyFeatureViewSet, MaterialViewSet, ManufacturerViewSet, BatteriesViewSet, ModuleViewSet, CellViewSet, EISViewSet*) and 1 view that needs some custom code because of the nature of the input that is delivered to us (*MeasurementViewSet*). Our views handle the communication with the database with ORM models which makes the correlation of the attributes of an SQL table with object in python and the querying of the database an automated task.

[2]For example getting, deleting or inserting some table records

For simplicity, we are going to explain the ChemicalViewSet and for reference we attach the code for it below in figure 4.8 . The ChemicalViewSet inherits from ModelViewSet which implements all the basic CRUD functionalities and also contains a custom LoggingMixin that records every incoming and outgoing request in a log file. Now, in order for this view to return a valid response it firstly checks if the user is authenticated with the *IsAuthenticated* class from the DRF module and then checks if the user has valid permissions to obtain the resource that they are asking with *JWTAuthentication* class from the *simplejwt* module. The next step is to validate the request input and reconstruct it in a python manageable form with a serializer (as shown in figure 4.9). Finally, the internal logic of the ModelViewSet handles all the further processing required to complete the request side effects and produce the necessary response.

```python
class ChemicalViewSet(LoggingMixin, ModelViewSet):
    queryset = Chemical.objects.all()
    serializer_class = ChemicalSerializer
    permission_classes = [IsAuthenticated]
    authentication_classes = [JWTAuthentication]
```

Figure 3.8: ModelViewSet - Chemical

```python
class ChemicalSerializer(serializers.ModelSerializer):

    You, 2 weeks ago | 1 author (You)
    class Meta:
        model = Chemical
        fields = "__all__"
```

Figure 3.9: Serializer - Chemical

The process for the *MeasurementViewSet* is similar, with the important distinction of needing custom logic to manage the POST request input. Unlike regular *ViewSets* and *Serializers*, *ModelViewSets* and *Serializers* require us to explicitly define each field and a custom validation function. The *Measurement* model fields are arrays containing four values, so to insert a single row into the database, we need to override the create function[19] of the ModelViewSet and implement the logic ourselves.

```python
class MeasurementViewSet(LoggingMixin, ModelViewSet):
    queryset = Measurement.objects.all()
    serializer_class = MeasurementsSerializer
    permission_classes = [IsAuthenticated]
    authentication_classes = [JWTAuthentication]

    # Save real time measurement data
    def create(self, request, *args, **kwargs):
        serializer = serializers.AddMeasurementSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        cell_id = serializer.validated_data.get("cell_ids")
        voltage = serializer.validated_data.get("voltage")
        temperature = serializer.validated_data.get("temperature")
        current = serializer.validated_data.get("current")
        sot = serializer.validated_data.get("sot")
        phase = serializer.validated_data.get("phase")
        soc = serializer.validated_data.get("soc")

        for id in range(4):
            cell = Cell.objects.get(id=cell_id[id])
            Measurement.objects.create(
                cell_id=cell,
                voltage=voltage[id],
                temperature=temperature[id],
                current=current[id],
                sot=sot[id],
                phase=phase[id],
                soc=soc[id]
            )

        return Response(serializer.data, status=status.HTTP_200_OK)
```

Figure 3.10: ModelViewSet - Measurements

```python
class AddMeasurementSerializer(serializers.Serializer):
    cell_ids = serializers.ListField(
        child=serializers.IntegerField(),
        min_length=4,
        max_length=4
    )

    voltage = serializers.ListField(
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    temperature = serializers.ListField(
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    current = serializers.ListField(      You, 3 months ago • Created Measurements endpoint
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    sot = serializers.ListField(
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    phase = serializers.ListField(
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    soc = serializers.ListField(
        child=serializers.FloatField(),
        min_length=4,
        max_length=4
    )

    def validate_cell_ids(self, value):
        queryset = Cell.objects.values_list('id')
        registered_ids = [item[0] for item in queryset]
        for candidate_id in value:
            if candidate_id < 0:
                raise serializers.ValidationError("cell_id field must be a positive integer number")
            if candidate_id not in registered_ids:
                raise serializers.ValidationError(f"Cell instance with cell_id: {candidate_id} must already exist, please create Cell instance")
        return value
```

Figure 3.11: Serializer - Measurements

3.3 REST API Documentation - Swagger UI

For the documentation of our REST API endpoints and for the easier testing of them we integrated the Swagger UI module[20] with our solution. When following the *https://dev-battery2life.iccs.gr/api/schema/swagger-ui/* URL we arrive at the landing page of our endpoint documentation (as seen in figure 3.12). Before we are able to use this UI to make requests to our app to test it we need to authorize. In order to do this we go to the token endpoints and make a POST request to the */api/token* endpoint with our username and password credentials. If successful, we will get an access and refresh token as a response. Then we need to click on the "Authorize" button on the top right, paste the access token into the tab shown in figure 3.13 and press "Authorize" then close the tab.

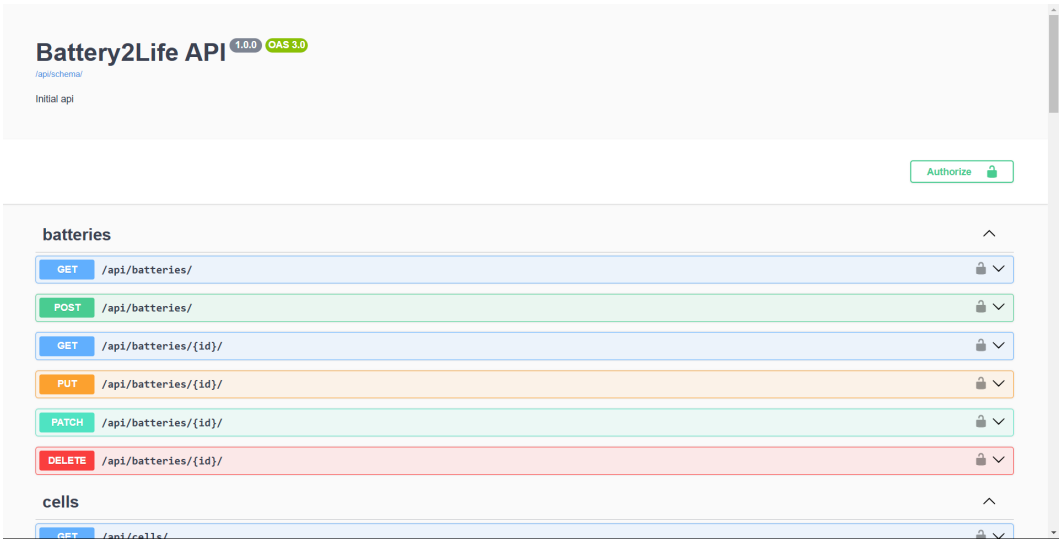


Figure 3.12: Swagger UI - Landing page

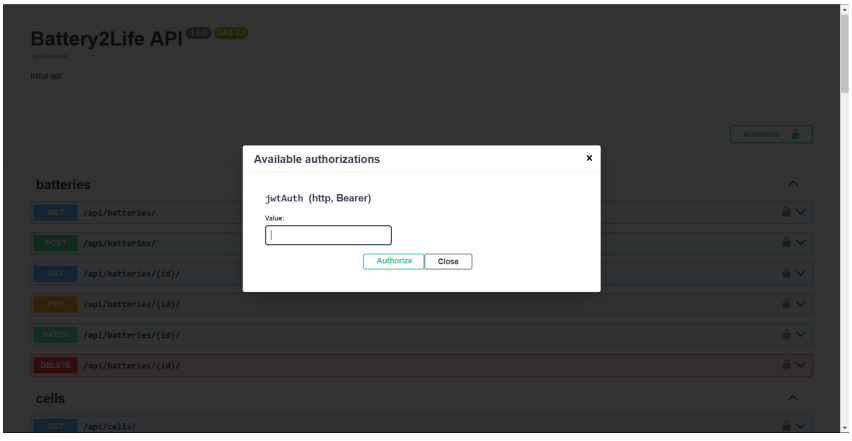


Figure 3.13: Swagger UI - Authorize

3.3.1 Performing HTTP requests

In order to make a request you need to click on a method of an endpoint (as show in picture 3.14) and then hit "try it out" and then "execute". If the request method is POST, you need to add the body of the request below before trying it out. If the request method is

a PUT, PATCH or DELETE you also need to provide the "id" for the record you want to change.

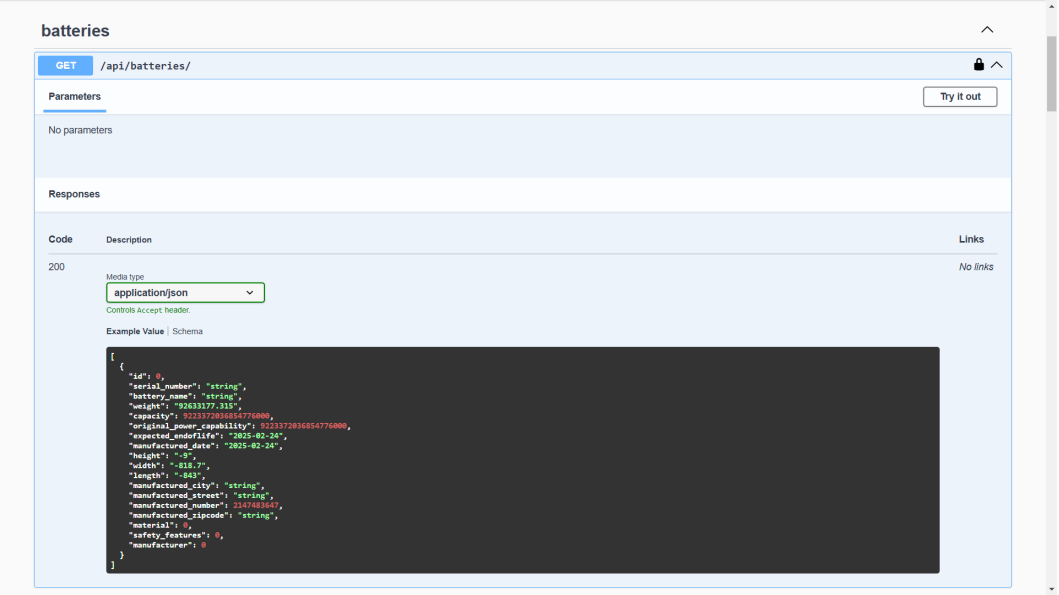


Figure 3.14: Swagger UI - Get request

The result of the previous experiments usually provides us with a response body, a status code and the response headers as seen in figure 3.15 .

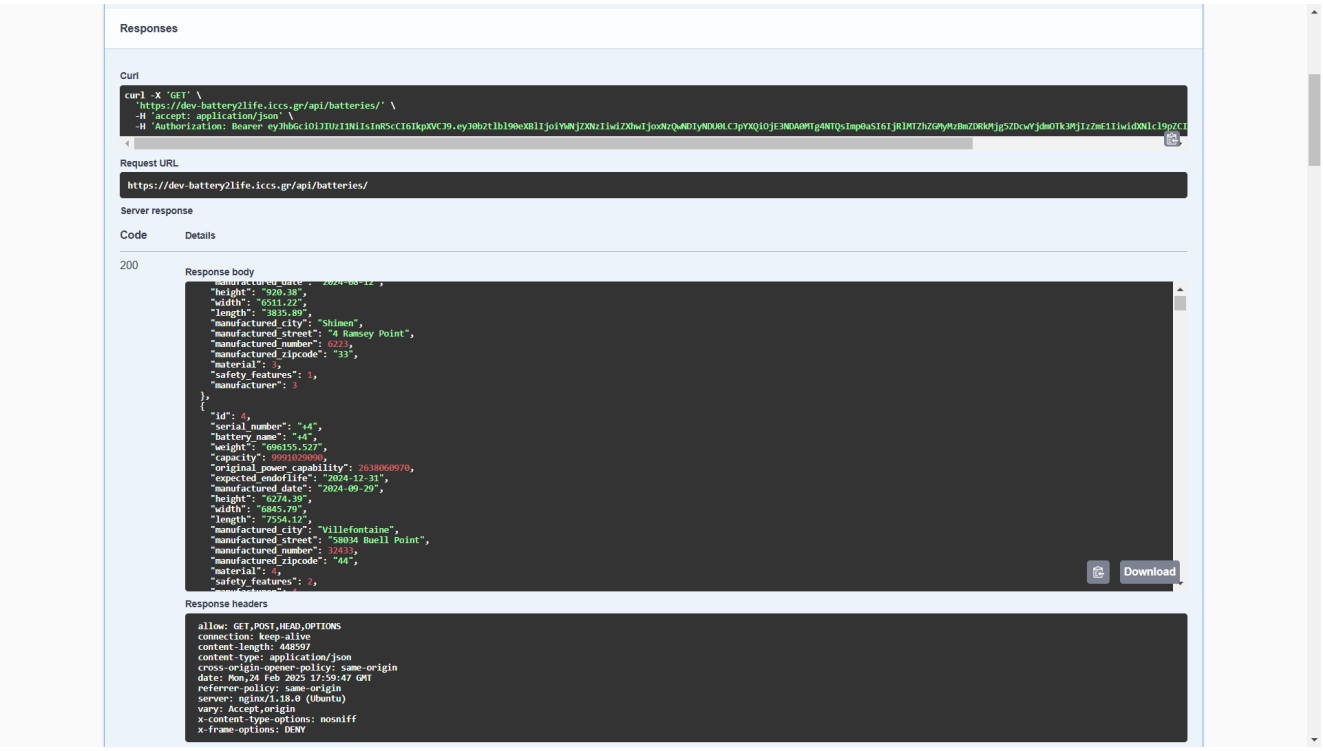


Figure 3.15: Swagger UI - Get response

3.4 Database - PostgreSQL

The database system of our choice is PostgreSQL. One of the many positives of PostgreSQL includes the Multi-Version Concurrency Control (MVCC) feature [21]. This functionality gives us high performance by providing concurrent execution of queries, which leads to low latency responses that is highly sought after in our system. Furthermore, its Atomicity, Consistency, Integrity and Durability (ACID)[22] design principles really aids to our goals to maintain a robust database without any data loss. Last but no least, PostgreSQL is open-source, which highly contributes to our goal of not having any vendor lock-in risks.

3.4.1 Database Django ORM

The Django ORM gives us a lot of flexibility. It implements SQL database table interpretations in corresponding python objects while still performing integrity checks with validators, checking for type matching, implementing foreign key constrains and deletion protocols and finally handling null/blank values. We chose the Cell model from our database to showcase the strength of this tool (shown in listing 3.2).

```
class Cell(models.Model):
    cell_name = models.CharField(max_length=255, unique=True)
    # Unit Ah
    nominal_capacity = models.IntegerField(blank=True, null=True, default=1)
    # Unit Wh
    nominal_energy = models.IntegerField(blank=True, null=True, default=1)
    nominal_cycles = models.IntegerField(blank=True, null=True, default=1)
    # Unit Wh/kg
    gravimetric_energy_density = models.IntegerField(blank=True, null=True, default=1)
    # Unit Wh/l
    volumetric_energy_density = models.IntegerField(blank=True, null=True, default=1)
    # (ex LFP71173207)
    industry_standard = models.CharField(blank=True, null=True, max_length=255)
    # Unit V
    nominal_voltage = models.DecimalField(max_digits=8, decimal_places=2, blank=True, null=True, default=1.0)
    # Unit V
    operating_voltage = models.DecimalField(
        max_digits=22,
        decimal_places=2,
        blank=True,
        null=True,
        validators=[MinValueValidator(0.0), MaxValueValidator(10.0)],
    )
    # Unit Megaohm
    ac_resistance = models.DecimalField(
        max_digits=22,
        decimal_places=2,
        blank=True,
        null=True,
        default=0.1
```

```
                    )
                    # Unit % / month
                    max_self_discharge_rate = models.DecimalField(
                        max_digits=22, decimal_places=2, blank=True, null=
                        True, default=1.0)
                    # Unit %
                    nominal_soc_at_delivery = models.DecimalField(
                        max_digits=22, decimal_places=2, blank=True, null=
                        True, default=1.0)
                    # Unit kg
                    cell_weight = models.DecimalField(max_digits=22,
                        decimal_places=2, blank=True, null=True, default
                        =1.0)
                    # Unit C
                    cell_charging_temperature = models.DecimalField(
                        max_digits=22,
                        decimal_places=2,
                        blank=True,
                        null=True,
                        validators=[MinValueValidator(0.0),
                            MaxValueValidator(60.0)],
                    )
                    # Unit C
                    cell_discharging_temperature = models.DecimalField(
                        max_digits=22,
                        decimal_places=2,
                        blank=True,
                        null=True,
                        validators=[MinValueValidator(-30.0),
                            MaxValueValidator(60.0)],
                    )

                    # cell dimension
                    # all metrics in mm
                    height = models.DecimalField(max_digits=6,
                        decimal_places=2,  blank=True, null=True)
                    width = models.DecimalField(max_digits=6,
                        decimal_places=2,  blank=True, null=True)
                    length = models.DecimalField(max_digits=6,
                        decimal_places=2,  blank=True, null=True)

                    cell_chemistry = models.ManyToManyField(Chemical,
                        blank=True, related_name='cells')
                    module = models.ForeignKey(Module, blank=True, null=
                        True, on_delete=models.CASCADE, related_name="
                        cells")
                    manufacturer = models.ForeignKey(
                        Manufacturer,
                        on_delete=models.CASCADE,
                        blank=True,
                        null=True,
                        related_name="cells",
                    )

                    class Meta:
                        verbose_name_plural = "Cells"

                    def __str__(self):
                        return self.cell_name
```

Listing 3.2: Cell Django ORM Model

We notice the following:

- A model interpretation of a database table inherits from a base class Model allowing high reusability and conciseness which enhances maintainability and simplicity of code
- Each model has certain fields types that helps us define different types of data
- Each field has certain attributes that define key characteristics of it. For example *cell_name* field defines an attribute of string type that has a max of 255 characters and each value is unique
- Django ORM defines its own *id*, when not stated otherwise, that auto-increments. The thing to keep in mind though is when populating the database manually (within the docker container and not from the API) this counter doesn't update on its own
- For foreign keys a one-to-many relationship is defined with the *on_delete* parameter stating what is going to happen on the connected values if the parent record is deleted. For example *on_delete=models.CASCADE* states that the record of the Cell model is retained if the record of the foreign model is deleted.
- Several Meta attributes can be defined for the easier management of the tables on the admin interface. For example the *__str__* attribute state the name of the model shown on the admin interface.

3.4.2 Database Configuration

Now we will talk about the **configuration and setup of the database**.

Our **database operates within a docker network as a docker container**. In order for our database to be created and configured the *init.sql* script (shown in figure 3.16) is mounted as a volume on the */docker-entrypoint-initdb.d* directory and is executed during the initialization of the container. Several environment variables including *user, password* and *db* are stated on the parameters of the *docker-compose.yaml* file which is described in section 3.7.1 .

The *init.sql* script configures PostgreSQL settings related to write-ahead logging (WAL), replication, and user management.

- The ALTER SYSTEM statements modify key parameters to enhance performance, particularly by reducing WAL-related overhead
- *synchronous_commit = 'off'*: disables synchronous commits, improving transaction speed by allowing transactions to be acknowledged before being written to disk, at the cost of durability
- *wal_level = 'minimal'*: minimizes the amount of WAL data generated, reducing logging overhead, which is ideal for non-replicated setups
- *wal_keep_size = 0* and *max_wal_senders = 0* :disable WAL retention and replication, ensuring WAL files are not retained for standby servers
- *archive_mode = off*: disables WAL archiving, preventing storage of logs for point-in-time recovery (PITR)

The script then creates the *b2l* database, followed by a procedural block (DO $$ ... $$) that ensures the root role exists. If not found, it creates the role with superuser, login, database creation, replication, and role creation privileges.

```
-- disable write ahead logging and checkpoints for batch population of the database
ALTER SYSTEM SET synchronous_commit = 'off';
ALTER SYSTEM SET wal_level = 'minimal';
ALTER SYSTEM SET wal_keep_size = 0;
ALTER SYSTEM SET max_wal_senders = 0;
ALTER SYSTEM SET archive_mode = off;


CREATE DATABASE b2l;

      You, 3 weeks ago • Added successfully Address Chemical Dimension S…

DO $$
BEGIN
    IF NOT EXISTS (SELECT FROM pg_roles WHERE rolname = 'root') THEN
        CREATE ROLE root WITH LOGIN PASSWORD 'root';
        ALTER ROLE root WITH SUPERUSER LOGIN CREATEDB REPLICATION CREATEROLE;
    END IF;
END $$;
```

Figure 3.16: Database - Initialization and Configuration script

3.4.3 Database Migrations

Another highly useful functionality, that Django offers us, is **database migrations**. Whenever we make a change to the model of the database, Django creates a new migration file containing information about how to apply them into the database with the *python manage.py makemigrations* command. Then, in order for these to take effect, we need to actually apply the migration with the *python manage.py migrate* command. This gives us a lot of information about previous database states and allows us to revert to previous versions very easily if required.

In our implementation such functions happen automatically during the initialization of the database docker container. Therefore, whenever we need to apply new migrations we have to stop, delete and restart the db and api containers (we need to maintain the volumes).

3.5 Activity Logging - Python/Django Loggers

Activity logging is a paramount component of our effort to provide a highly available platform with low fault tolerance by minimizing the downtime and debugging time. We implemented logging for two modules which includes the *views* of the Server module and for the Message Broker Client.

3.5.1 Python Logging Setup

A custom Django Mixin[23] was implemented in order to log the HTTP requests and responses of our cloud platform. Making this feature into a separate component allows us to reuse it therefore making our code simpler, all the while making our codebase easier to be maintained and extended as wished.

In order to implement a Django Logging Module we configured the *settings.py* of the main application as shown in figure 3.17 .

Figure 3.17: Logging - ModelViewSets

Two message logging *formatters* were created with the structure seen above. The *filters* were set to log when in Debug mode. Then we have some *handlers* that state how to parse potential messages received. Finally, we have the *loggers*, that is the highest level of abstraction, incorporating the whole logic that was defined with the specific filters, handlers, level of logging and hierarchy stated by the attribute *propagate*.

For the Mosquitto Client a more direct approach was taken. As seen below in figure 3.18 two loggers has been defined. One for printing logs into the command line and one for printing logs into a file, ensuring the persistence of the logs. The Mosquitto Client utilizes the logger

when connecting to specific clients, topics and when receiving a message.

```python
from paho.mqtt import client as mqtt
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s-%(levelname)s: %(message)s')

console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)
console_handler.setFormatter(formatter)

file_handler = logging.FileHandler('mosquitto.log')
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(formatter)

logger.addHandler(console_handler)
logger.addHandler(file_handler)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s: %(message)s' ,
    handlers=[
        logging.FileHandler('./log.txt'),
        logging.StreamHandler()
    ]
)

logger.debug('Starting Mosquitto Client...')

def on_connect(mqtt_client, userdata, flags, rc):
    if rc == 0:
        logging.debug('Client connected successfully to mosquitto...')
        mqtt_client.subscribe('hello/topic')
        logging.debug('Subscribed successfully')
    else:
        logging.debug(f'Bad Connection. Code: {rc}')

def on_message(mqtt_client, userdata, msg):
    logging.info(f"Got message with payload: {msg.payload} from topic: {msg.topic}")

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set(
    os.environ.get("MQTT_USER"),
    os.environ.get("MQTT_PASSWORD")
)
client.connect(
    host=os.environ.get("MQTT_BROKER"),
    port=int(os.environ.get("MQTT_PORT")),
    keepalive=int(os.environ.get("MQTT_KEEPALIVE"))
)

client.loop_forever()

```

Figure 3.18: Logging - Mosquitto Client

3.6 Development Tools

3.6.1 Execution Environment - Docker Containers

The Docker containers[24] enables us control the environment and configuration of the running services. Some key benefits include better horizontal scalability, separation of responsibility for different services and seamless deployment to any resource, cloud or local.

Now we are going to describe how we implemented each service as a container and their main behavior. As seen in listing 3.3 we have 4 services defined, which are:

- api: implements main server of our system
- db: implements the database section
- mosquitto: implements the Mosquitto Broker
- mosquitto-client: which implements the Mosquitto client

In order to be able to create a container first the exact image needs to be specified or the build section of the specific service. Applications like api and mosquitto-client specifies custom images with base Linux layers and installation of specific modules useful for their operation. On the other hand services like db and mosquitto get ready to use images from the docker hub[25] cloud, stating the specific version (in order to be able to know exactly if we have a bug which version caused it) of a PostgreSQL and Mosquitto broker. In turn, this entails that when we want to upgrade the database version we have to do it manually.

Furthermore, we see that containers are configured to restart every time they stop, run specific commands when initialized, depend on specific ontainers' initialization before they themselves begin and belong to a specific network stated as configuration. Port forwarding is essential for the ability to have these services available from the host machine.

One of the key points of the *docker-compose.yaml* configuration are the volumes. There files mounted during the initialization of the container inside the internal file structure of it mapped from the project directory. Lastly and most importantly, we can see some volumes like *pgdata* are managed from the docker daemon. These volumes constitute one of the most important parts of our applications because when the db service experiences downtime after it resuscitates the data are persisted and the database is revived entirely in its previous state.

Listing 3.3: Docker-Compose.yaml Configuration

```
services:
  api:
    build: ./battery2life
    container_name: api
    restart: unless-stopped
    command: >
      sh -c "␣python␣manage.py␣makemigrations␣--noinput␣&&
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣python␣manage.py␣migrate␣&&
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣python␣manage.py␣create_admin␣&&
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣python␣manage.py␣runserver␣0.0.0.0:8000
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
    depends_on:
      - db
    ports:
      - 8000:8000
    volumes:
      - ./battery2life:/usr/src/app
      - ./battery2life/app.log:/usr/src/app/app.log
    env_file:
      - ./.env.dev
```

```
    networks:
      - db


  db:
    image: postgres:17.0
    container_name: db
    restart: unless-stopped
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.
          sql
      - ./database/data:/data
      - ./database/dummy_data:/data
      - ./database/populate_dummy.sql:/populate_dummy.sql
    environment:
      - POSTGRES_USER=root/
      - POSTGRES_PASSWORD=root
      - POSTGRES_DB=b2l
    networks:
      - db


  mosquitto:
    image: eclipse-mosquitto
    container_name: mosquitto
    restart: unless-stopped
    depends_on:
      - db
    ports:
      - "1883:1883"
      - "9001:9001"
    volumes:
      - ./mosquitto/config:/mosquitto/config:rw
      - ./mosquitto/data:/mosquitto/data:rw
      - ./mosquitto/log:/mosquitto/log:rw
    networks:
      - mosquitto


  mosquitto-client:
    build: ./mosquitto-client
    container_name: mosquitto-client
    restart: unless-stopped
    depends_on:
      - mosquitto
    environment:
      - MQTT_BROKER=mosquitto
      - MQTT_PORT=1883
      - MQTT_KEEPALIVE=60
      - MQTT_USER=user1
      - MQTT_PASSWORD=user1
    volumes:
      - ./mosquitto-client/mosquitto-client.log:/usr/src/
          mosquitto-client/mosquitto-client.log
    networks:
      - mosquitto
    command: [ "python3",  "mosquitto_client.py" ]


volumes:
  battery2life:
  pgdata:
  mosquitto-client:
  mosquitto:

networks:
  mosquitto:
    driver: bridge
  db:
```

```
91                  driver: bridge
```

3.6.2 Version Control Management - Git

For the efficient management of our project versioning, the git tool[26] coupled with a GitLab repository was used. Through the use of branches like main, staging, development, feat/{feature} and bug/{bugFix} we were able to streamline the development, track the changes of specific features and isolate the specific fixes of a problem. A cloud repository was necessary when multiple people were working on the same project to ensure the project version consistency.

More specifically for each branch:

- **Main**: This is the branch were after rigorous testing on a development environment, a group of updates are published as releases.
- **Staging**: This is the branch where functional and unit testing are performed before publishing a release.
- **Development**: This is the branch that aggregates the development of entire features and bug fixes.
- **Feature**: This branch is created when we want to create a feature that we think multiple commits are going to be included
- **BugFix**: This is the branch that encapsulates the whole process of fixing a specific bug during development.

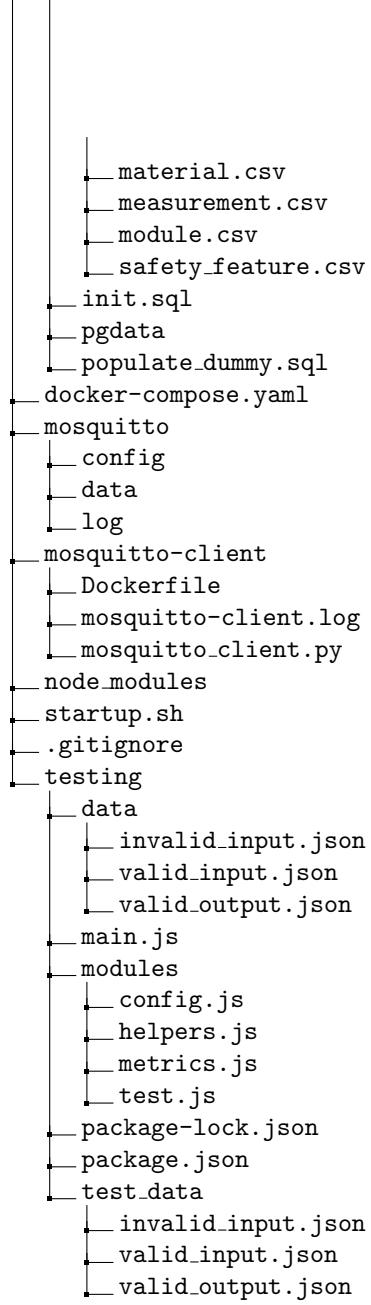
3.7 System Practical Instructions

The project is supported by a GitLab repository with scripts and HOWTOs. The rights for the access of source code are restricted, therefore for getting those right contact the administrators at ***vangelis.tsougiannis@iccs.gr*** and ***giannismitis@gmail.com***

3.7.1 Project file structure

The project file structure is as follows:

```
/
├── README.md
├── battery2life
│   ├── Dockerfile
│   ├── api_schema.yaml
│   ├── app.log
│   ├── batteries
│   │   ├── __pycache__
│   │   ├── admin.py
│   │   ├── api
│   │   │   ├── __pycache__
│   │   │   ├── serializers.py
│   │   │   ├── urls.py
│   │   │   └── views.py
│   │   ├── apps.py
│   │   ├── management
│   │   ├── migrations/
│   │   ├── mixins.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   ├── utils.py
│   │   └── views.py
│   ├── battery2life
│   │   ├── __init__.py
│   │   ├── __pycache__
│   │   ├── app.log
│   │   ├── asgi.py
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   ├── db.sqlite3
│   ├── manage.py
│   ├── requirements.txt
│   └── venv
├── create_necessary_files.sh
└── database
    ├── data
    ├── database_erd.mmd
    └── dummy_data
        ├── batteries_cell_cell_chemistry.csv
        ├── battery.csv
        ├── cell.csv
        ├── chemical.csv
        ├── dimension.csv
        ├── eis.csv
        └── manufacturer.csv
```



Explanation

For the entire system, the most vital files are:

- `docker-compose.yaml`[3]: This is the most important file and contains the definitions for all the available services in a format that makes them really easy to deploy.
- `.gitignore`: The file states which files are to be ignored from tracking their history by Git.
- `create_necessary_files.sh`: Script to create the files necessary for logging, since they contain sensitive information and are not tracked by Git.

[3]The docker-compose defines features like images (exact instructions to deploy each service), container parameters (host name, environment variables, port forwarding), volumes (maps directories used by the container) and networks (connecting container with private networks). The entire services stack is managed by a single command (docker-compose up/down/restart/stop)

Django App

The entire API resides in the `./battery2life` directory. The most important files are:

- `Dockerfile`: This is the custom Docker image we built to run the application, including a base system image and all the appropriate packages.
- `manage.py`: This is the basic script that allows us to perform database management actions and create new applications.
- `requirements.txt`: This file is used for the bookkeeping of the modules used in our application. The Dockerfile image uses it to install any module necessary inside the produced docker container.

Main Application:
Our main application is the one in the `./battery2life/battery2life` directory. It contains the following crucial files:

- `settings.py`: This is the main settings file containing settings for loggers, the database, authentication, etc.
- `urls.py`: This is the main file for URLs (Unified Resource Locators) of our resources.

API Application:
This is the application that implements the main functionalities. It resides in the directory `./battery2life/batteries`. It contains the following files and directories:

- `admin.py`: Includes the definitions for the database models to be used by the admin control panel.
- `api/`: This directory includes the application views (they parse the request and send responses), serializers (they translate the JSON input to a known Python format and vice versa), and URLs (they match views and endpoints/URLs).
- `migrations/`: This directory contains the database version history.
- `models.py`: Defines the Django Object Relational Mapping(ORM) models for the database.

PostgreSQL Database

The necessary files for the database are located in the `./database` directory. The most important ones are:

- `database_erd.mmd`: This is the entity-relationship schema of our database done in Mermaid markup language.
- `init.sql`: This is the script that is run when the database docker container starts. Its purpose is to initialize and configure the database.
- `pgdata`: This is the volume that is mapped by Docker and contains all the data of the database. This file ensures that all our data is preserved in the event of a database failure.

- `dummy_data/`: This is the directory that contains all the mock data useful for testing the API and the database.
- `populate_dummy.sql`: This is the script that populates the database with dummy data.

Mosquitto Broker

The necessary directories for the Mosquitto message broker are located in the `./mosquitto` directory. The most important ones are:

- `config/`: Contains the `mosquitto.conf` file, which is the configuration data for the broker, and the `passwd` file used for authentication of clients.
- `data/`: Contains the data that persists when we receive a message.
- `log/`: Contains the log data from the broker.

Mosquitto Client

The necessary files for the Mosquitto Client are located in the `./mosquitto-client` directory. The most important ones are:

- `Dockerfile`: The custom image created to enable the client to use a Python logger, connect to the broker and handle the incoming/outgoing messages.
- `mosquitto_client.py`: The implementation of the Mosquitto client together with the client logger.

3.7.2 Basic Usage Instructions

Next follows some basic instructions to download and run the app locally. For more detailed instructions refer to the administrators for access.

Clone the repository

```
# clone repository
cd ~/
git clone {repository_url}

# navigate to the necessary repository
git branch -a
git checkout -b [BranchName] origin/[BranchName]

```

Run the app

```
            # run automated creation script
            chmod +x create_necessary_files.sh
            ./create_necessary_files.sh

            # start all of the services
            cd ~/backend
            docker-compose up -d --build --force-recreate

            # stop all of the services and remove all containers and
    networks
            cd ~/backend
            docker-compose down

```

Test the API

Step 1: Navigate to the API Documentation

- Start the application as described above and go to `http://localhost:8000/api/schema/swagger-ui/`.

Step 2: Authenticate

- Click on the `/api/token/` endpoint.
- Make a POST request with your username and password.
- Copy the access token from the response of the request.
- On the top right corner of the page, click **Authorize** (green button with a lock).
- Paste your access token into the value placeholder.
- Click the **Authorize** button and then press the **Close** button.

Step 3: Test Endpoints

- Click on an endpoint.
- Click on **Try it out**.
- Fill in the required HTTP method information (e.g., ID, body).
- Click on the blue **Execute** button.
- Inspect the response headers, URL, status code, and body below in the **Response** section.

Step 4 (Optional): Request Example Payloads

- You can find example payloads for the */api/measurements/* and */api/eis* endpoint in section 2.2.1: **REST API Endpoint Structure**.
- You can also see example responses to compare if your response was valid.

Chapter 4

Results

In this chapter, we will begin by explaining the reasoning for the testing, then we are going to give a high level description of the testing methods applied to the cloud platform and then we are going to highlight the metrics employed for the validity of the system. Finally, we are going to close with the presentation and commenting of the results of these experiments.

4.1 Testing Methodology

4.1.1 Testing philosophy

As we have mentioned in chapter 2, our system is going to be used by BMSs to publish data and by administrators for the management/bookkeeping of the system[1].

Therefore, some key considerations for measuring our system performance and usability includes:

- Our system needs to be able to process requests at a minimum rate of one request per second [8]
- Our request needs to adhere to low response times to ensure a basic Quality of Service (QoS)
- The API endpoints needs to have the expected responses and side effects
- Whenever the system doesn't behave in the way that it should, there should be mechanisms to bring the system back to consistency.
- The Get all HTTP requests are performed once per week by State-of-Warranty algorithms therefore they need to present satisfying Quality of Service (QoS)

4.1.2 Testing Solution

In order to satisfy, to the best of our ability, the aforementioned considerations we devised a method of testing. This method includes performing a number of HTTP requests for all the available methods of the most basic endpoints.

A number of concurrent Virtual Users(VUs) are going to pose these requests in parallel for a limited amount of time. The endpoints under test include the analytics endpoints (EIS, Measurements) and the most basic battery management endpoints (Batteries, Modules, Cells).

Two experiments are going to be conducted, one testing the system under normal load and one under big load. The key metric that defines how load heavy is the experiments lies on the number of concurrent Virtual Users. These experiments are going to be conducted two times each, once with valid inputs and once with invalid inputs. The invalid inputs will include missing values, wrong value type, out of range values, wrong parameters, duplicate values for unique fields.

4.1.3 Testing Validity/Performance

For the functionality aspect of the system for each request there is a *check* routine for validating if the request was parsed as expected. This routine checks the HTTP status code, the type of the response body and the contents of the response body to define if the request has been parsed successfully or not.

For the performance aspect of the API we used a confusion matrix including the parameters: *accuracy*, *misclassification*, *precision*, *sensitivity* and *specificity*. These metrics really helps

[1]Key physical system information about batteries/modules/cells such as models, components, specifications etc.

our testing scheme identify and quantify the positive and negative cases, giving us a good overview about the areas of improvements and the restrictions of our API[27].

Some key metrics that gives us a good overview about the systems scalability and user experience are *throughput* and *response time*. They show us how good is a system at processing high volumes of data in the unit of time and showcase the delay factor of the response of the requests[28].

4.1.4 System Under Test

Our system is deployed in a cloud Virtual Machine(VM) as a network of docker containers. Below is a table with the specifications of this machine:

Table 4.1: System Specifications

Component	**Specifications**
Memory	**Main:** - Model: QEMU - Space: 16 GB - Capabilities: Not specified **Secondary:** - Space: 100 GB - Type: SSD - Model: QEMU HARDDISK - Capabilities: 5400 RPM, GPT-1.00 partitioned
CPU	- Number: 4 - Cores per CPU: 2 - Processing Speed: 2.49 GHz - Model: Not specified
OS	- Name: Ubuntu - Version: 22.04.4 LTS
Network	- Download Speed: 925 Mbps - Upload Speed: 940 Mbps

4.2 Testing Results Presentation

4.2.1 Testing Implementation

For our experiment, we used k6[29] for performing the tests and Grafana[30] for the visualization of metrics. Prior to starting, we generated 1000 records of dummy data for each of the main endpoints and populated the database.

One full experiment includes several stages. Firstly, we conduct a request to gain the necessary JWT in order to be able to call the endpoints. After that we iterate over every endpoint and conduct the same test with 5 valid inputs or 5 invalid inputs. Then we get the result of the request and conduct checks about the status of the response[2], the type of the returned item and the composition of the returned item (all fields are valid). Finally, we update the metrics necessary to calculate the confusion table at the end[3].

An example of valid and invalid input respectively are shown below.

VALID_INPUT:

```
{
    "/api/batteries/": {
        "GET": null,
        "POST": [
            {
                "serial_number": "zdaz6d51",
                "battery_name": "ufwo0sf6",
                "weight": 6671.295,
                "capacity": 776000,
                "original_power_capability":
                    776000,
                "expected_endoflife": "2027-02-15",
                "manufactured_date": "2023-02-15",
                "height": 3025.14,
                "width": 3,
                "length": 8888,
                "manufactured_city": "string",
                "manufactured_street": "string",
                "manufactured_number": 2147483647,
                "manufactured_zipcode": "string"
            }
        ]
    },
    "/api/batteries/{id}/": {
        "GET": null,
        "PUT": [
            {
                "serial_number": "cmqp0bcj",
                "battery_name": "wfi1gw01",
                "weight": 6400.85,
                "capacity": 7850000,
                "original_power_capability":
                    7850500,
                "expected_endoflife": "2029-03-15",
                "manufactured_date": "2022-12-01",
```

[2]Successful status codes include: 200, 201, 204

[3]Includes: True Positive (we have a valid input and successful status), True Negative (we have an invalid input and unsuccessful status), False Positive (we have an invalid input and successful status), False Negative (we have a valid input and unsuccessful status)

```
                                "height": 3100.25,
                                "width": 5,
                                "length": 8700,
                                "manufactured_city": "Chicago",
                                "manufactured_street": "Lake Shore
                                    Drive",
                                "manufactured_number": 500,
                                "manufactured_zipcode": "60611"
                            }
                        ],
                        "PATCH": [
                            {
                                "serial_number": "fsj67mqx",
                                "battery_name": "m9owrafc",
                                "weight": 6420.12
                            }
                        ],
                        "DELETE": null
                    }
                }
```

INVALID_INPUT:

```
                {
                    "/api/batteries/": {
                        "POST": [
                            {
                                "serial_number": "zl4xkln9",
                                "battery_name": "
                                    uwgvizkongdfogdfsngoisdfghdpijj
                                ghdfsogudfshgpsidfugjsdfhgpudfgd
                                fhsgipusdfjhipudfsjghdsfipugjhdf
                                siupgdfsjghdsfuijgkhdsfipugjdkfh
                                gpifsdujghsdfiughdfiugfdspuidsiy
                                euwgvizkongdfogdfsngoisdfghdpij
                                jghdfsogudfshgpsidfugjsdfhgpudfg
                                dfhsgipusdfjhipudfsjghdsfipugjhd
                                fsiupgdfsjghdsfuijgkhdsfipugjdkf
                                hgpifsdujghsdfiughdfiugfdspuidsi
                                yeuwgvizkongdfogdfsngoisdfghdpi
                                jjghdfsogudfshgpsidfugjsdfhgpudf
                                gdfhsgipusdfjhipudfsjghdsfipugjh
                                dfsiupgdfsjghdsfuijgkhdsfipugjd
                                kfhgpifsdujghsdfiughdfiu",
                                "weight": 5000.754353,
                                "capacity": 650000,
                                "original_power_capability":
                                    650000,
                                "expected_endoflife": "2028-06-20",
                                "manufactured_date": "2022-08-10",
                                "height": 2800.5,
                                "width": 5,
                                "length": 7500,
                                "manufactured_city": "New York",
                                "manufactured_street": "5th Avenue"
                                    ,
                                "manufactured_number": 120,
                                "manufactured_zipcode": "10001"
                            }
                        ]
                    },
                    "/api/batteries/{id}/": {
```

```
                        "GET": null,
                        "PUT": [
                            {
                                "serial_number": "zl4xkln9",
                                "battery_name": "
                                vaiuvaipuefgaipuvbaipuuwgv
                                izkongdfogdfsngoisdfghdpijj
                                ghdfsogudfshgpsidfugjsdfhgpudfgd
                                fhsgipusdfjhipudfsjghdsfipugjhdf
                                siupgdfsjghdsfuijgkhdsfipugjdkfh
                                gpifsdujghsdfiughdfiugfdspuidsiy
                                euwgvizkongdfogdfsngoisdfghdpij
                                jghdfsogudfshgpsidfugjsdfhgpudfg
                                dfhsgipusdfjhipudfsjghdsfipugjhd
                                fsiupgdfsjghdsfuijgkhdsfipugjdkf
                                hgpifsdujghsdfiughdfiugfdspuidsi
                                yeuwgvizkongdfogdfsngoisdfghdpi
                                jjghdfsogudfshgpsidfugjsdfhgpudf
                                gdfhsgipusdfjhipudfsjghdsfipugjh
                                dfsiupgdfsjghdsfuijgkhdsfipugjd
                                kfhgpifsdujghsdfiughdfiu",
                                "weight": 50.754353,
                                "capacity":
                                    650543253425235342545240000,
                                "original_power_capability":
                                    650000,
                                "expected_endoflife": "2028-06-20",
                                "manufactured_date": "2022-08-10",
                                "height": 2800.5,
                                "width": 5,
                                "length": 7500,
                                "manufactured_city": "New York",
                                "manufactured_street": "5th Avenue"
                                    ,
                                "manufactured_number": 120,
                                "manufactured_zipcode": "10001"
                            }
                        ],
                        "PATCH": [
                            {
                                "serial_number": "c5axt2qj",
                                "width": 5,
                                "manufacturer": 200,
                                "battery_name": "n9inbrba",
                                "manufactured_city": "London",
                                "original_power_capability":
                                    7750500
                            }
                        ],
                        "DELETE": null
                    }
                }
```

4.2.2 Experiment 1: Normal Load Testing

Methodology

For the first experiment, we tested the system under normal load. More specifically, we implemented a testing scenario where we had from the get-go 5 concurrently running VUs and they run 4 stages in total.

During the first stage, for the duration of 1 minute one iteration (full experiment) was going to be executed every 1 second for every VU. The second stage featured the retention of the number of VUs but it run 2 iterations for the duration of 2 minutes. The third stage scaled to 5 iterations per VU (again for the same number of VUs) for a duration of 1 minute. Finally, for 1 minute we have the stage of ramping-down with no additional requests being made but ongoing requests are allowed to finish.

An overview of the experiment can be seen below. The orange time series represents the rate of failed requests (measured in request/s), the blue represents the response time value (measured in milliseconds), the yellow represents the rate of requests made (requests/s) and the green represents the number of the available Virtual Users.

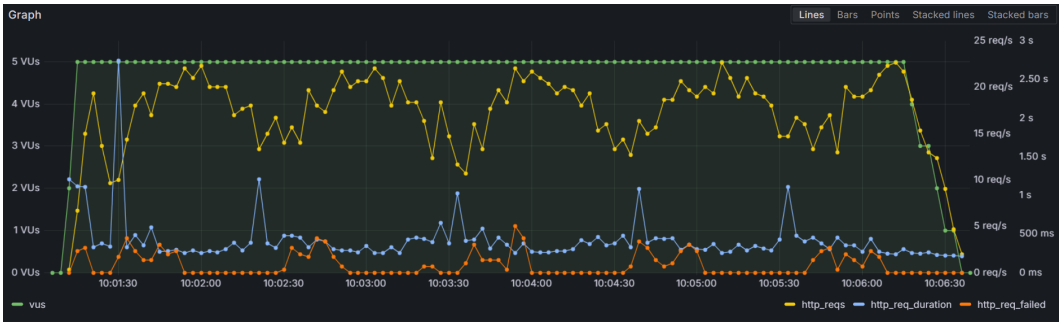


Figure 4.1: Experiment 1 - Overview

Results

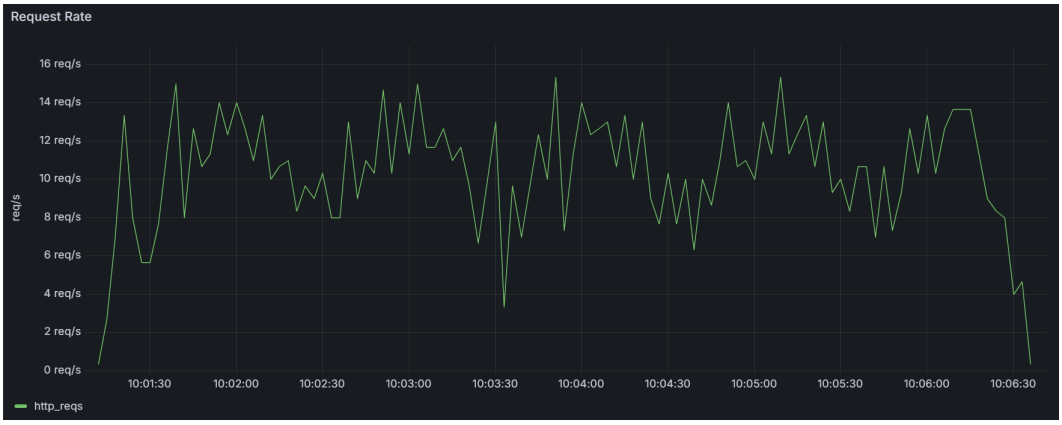


Figure 4.2: API - request rate

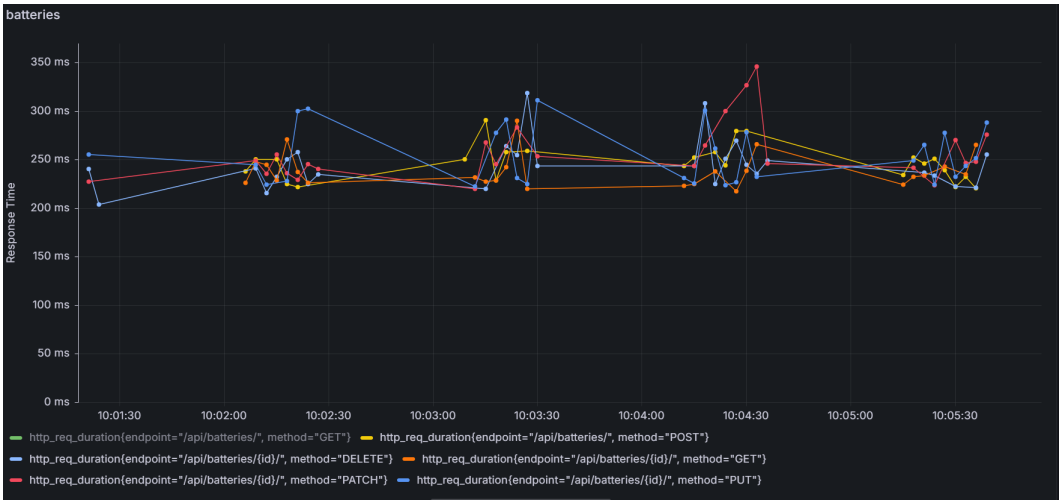


Figure 4.3: Batteries endpoint - response time

Figure 4.4: Modules endpoint - response time

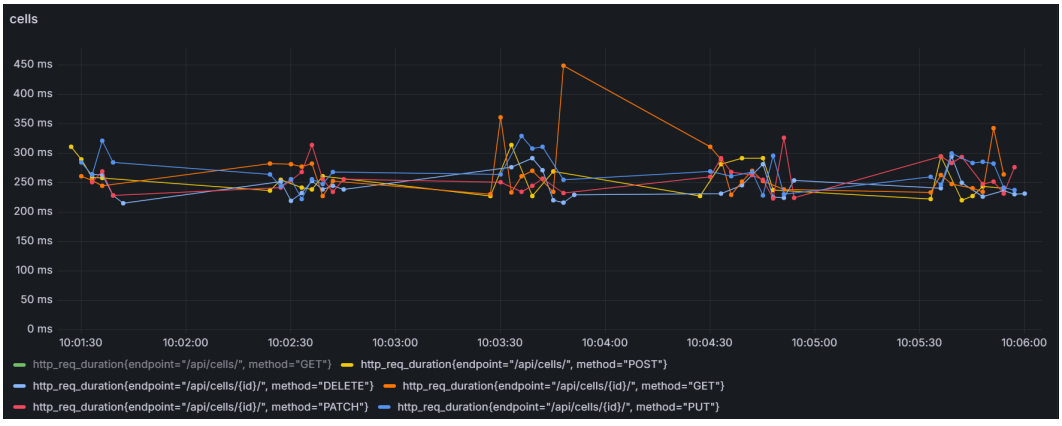


Figure 4.5: Cells endpoint - response time

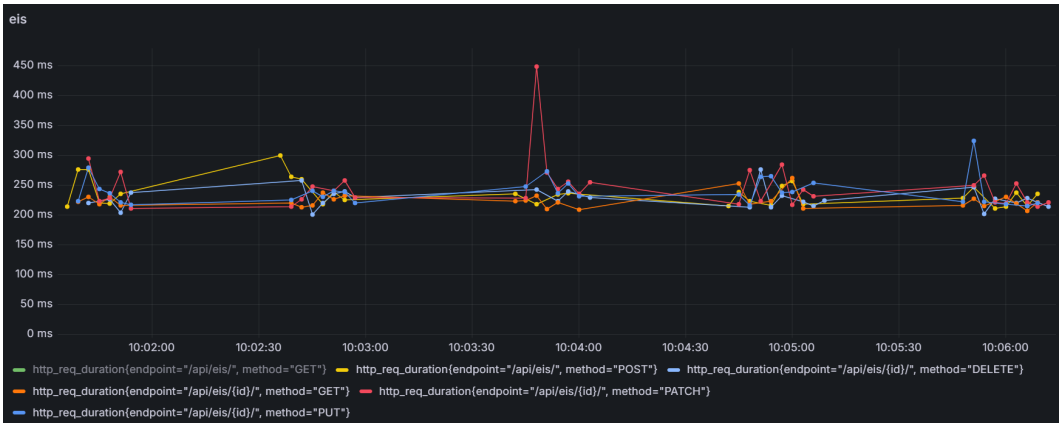


Figure 4.6: EIS endpoint - response time

Figure 4.7: Measurements endpoint - response time

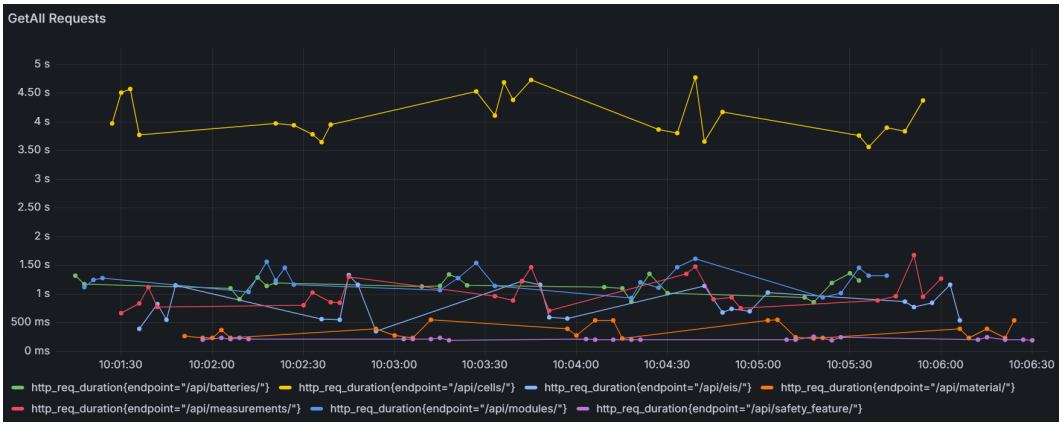


Figure 4.8: All endpoints get all method - response time

Experiment 1	**Iterations**	**Concurrent Virtual Users**
Input 1 (Valid Input)	5721	5
Input 2 (Invalid Input)	6072	5

Table 4.4: Experiment Parameters

API validation confusion matrix	Valid Input	Invalid Input
Valid output	5354 (True positive – TP)	3 (False positive – FP)
Invalid output	367 (False negative – FN)	6069 (True negative – TN)

Table 4.2: API Validation Confusion Matrix

Metric	Value
Accuracy	96.86%
Misclassification	3.14%
Precision	99.94%
Sensitivity	93.58%
Specificity	99.95%

Table 4.3: Performance Metrics

4.2.3 Experiment 2: Heavy Load Testing

Methodology

In this experiment, the system is going to be inspected under much heavier than normal use. Namely, we have two stages where in the first stage the system gradually scales to 25 VUs over a period of 1 minute. During this period an for every period there is no set number of iterations, but the VUs conduct as many of them as they can. Then for 2.5 minutes the system retains the same number of Virtual Users. Finally, the system is given 30 seconds grace-period to finish all the ongoing requests before it terminates.

An overview of the whole second experiment can be seen below.

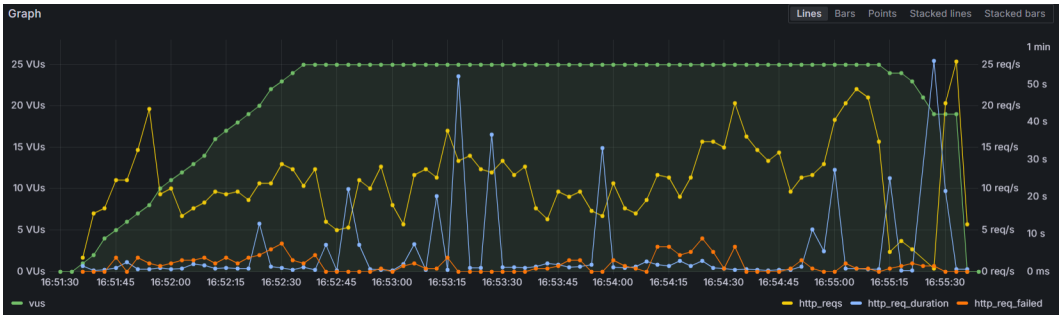


Figure 4.9: Experiment 2 - Overview

Results

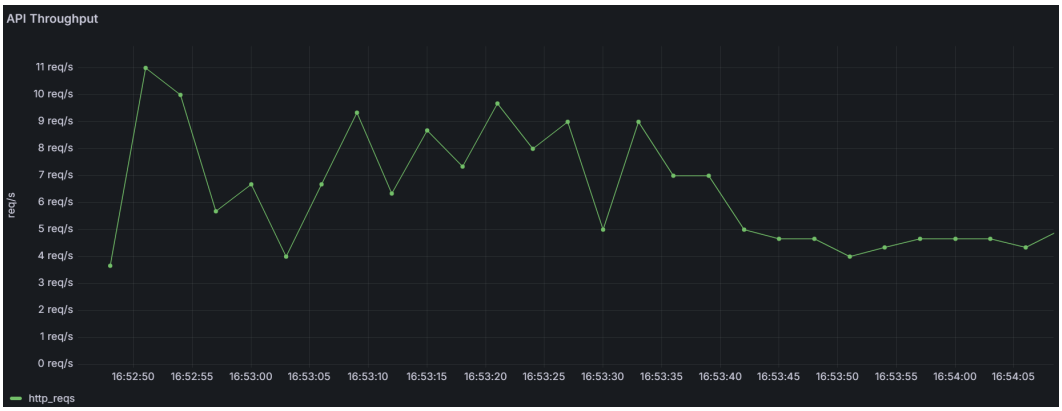


Figure 4.10: API endpoint - request rate

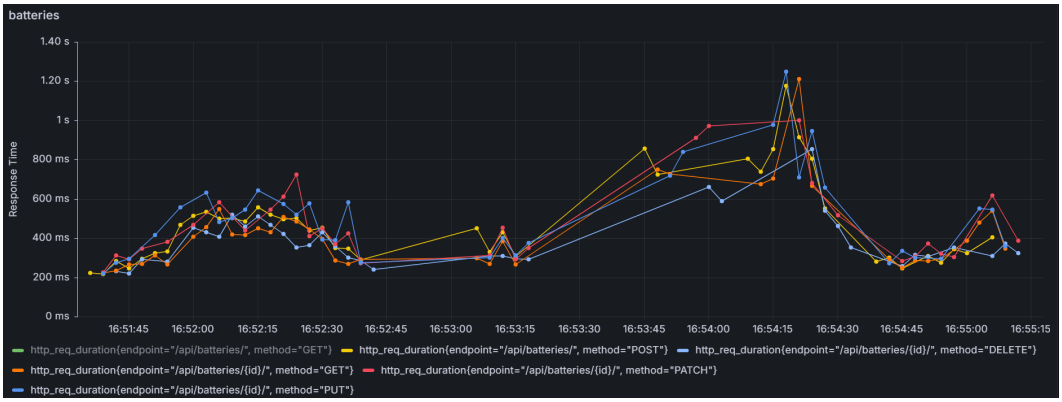


Figure 4.11: Batteries endpoint - response time

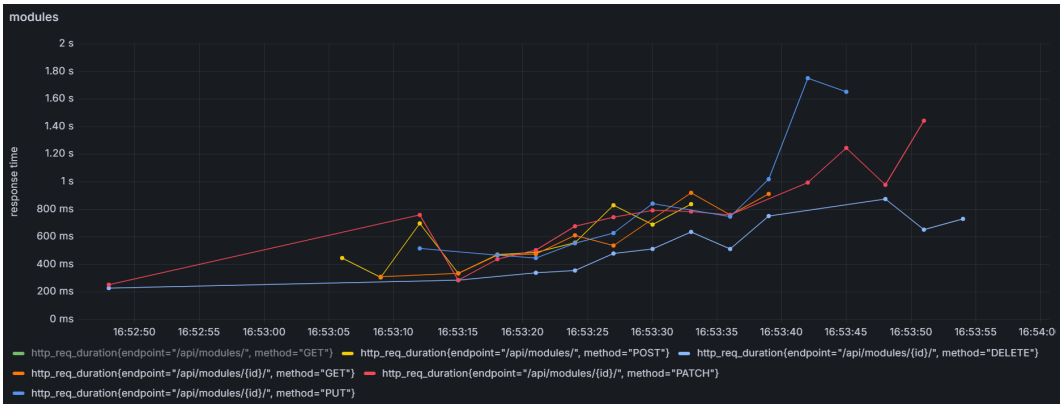


Figure 4.12: Modules endpoint - response time

Figure 4.13: Cells endpoint - response time

Figure 4.14: EIS endpoint - response time

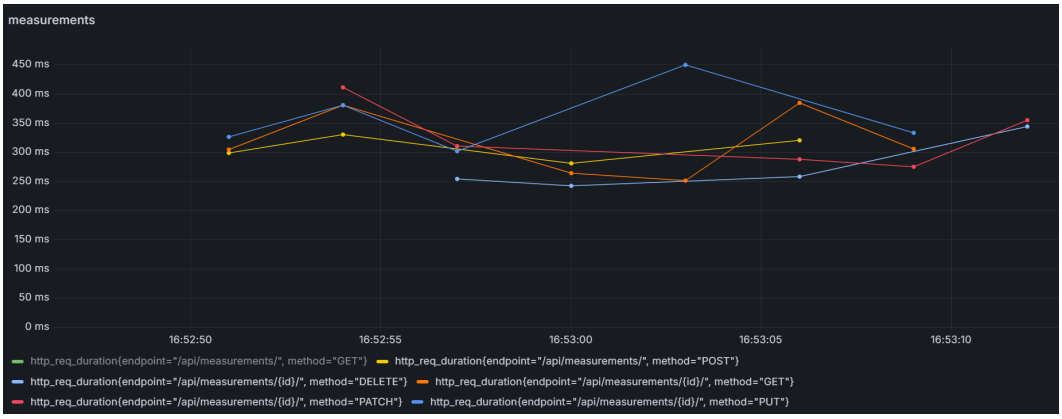


Figure 4.15: Measurements endpoint - response time

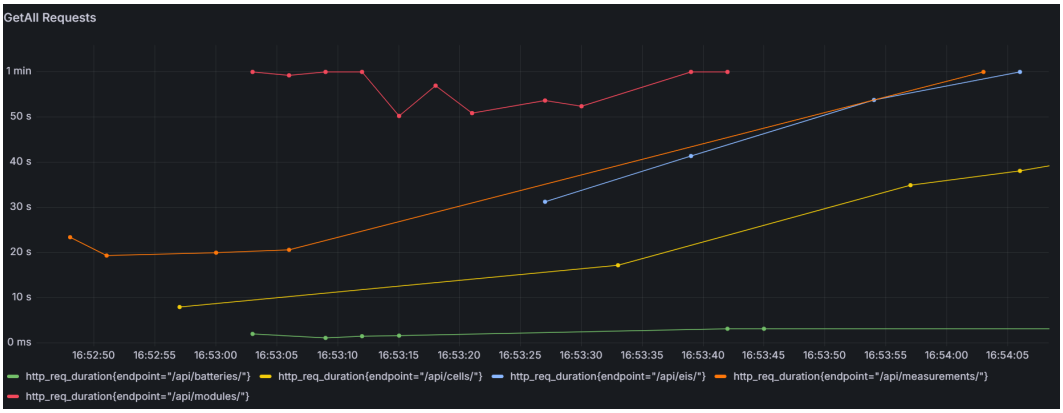


Figure 4.16: All endpoints get all method - response time

Experiment 2	**Iterations**	**Concurrent Virtual Users**
Input 1 (Valid Input)	5082	25
Input 2 (Invalid Input)	11789	25

Table 4.5: Experiment Parameters

API validation confusion matrix	**Valid Input**	**Invalid Input**
Valid output	4465 (True positive – TP)	3 (False positive – FP)
Invalid output	617 (False negative – FN)	11786 (True negative – TN)

Table 4.6: API Validation Confusion Matrix

Metric	**Value**
Accuracy	99.98%
Misclassification	3.81%
Precision	99.93%
Sensitivity	87.80%
Specificity	99.98%

Table 4.7: Performance Metrics

4.2.4 Findings

Confusion Matrix

While looking at the confusion matrix of the two experiments, there is some key takeaways and things to consider for the future use and development of our platform.

Firstly, we notice that even though accuracy is very high in both cases its is not perfect. This means that in some cases we expected certain results, but we got wrong ones. By taking a look at the failed requests, we notice that most failures that stem from POST requests provide subsequent failures to PATCH, PUT and DELETE methods as our tests are conducted in a way that any record created by the test must be edited and deleted and no other ones. And this is not the case for the endpoint measurements, which yields further errors. Nonetheless, the spike in false negatives is not going to have a big effect on the utility of our platform, since some data are not going to be available rather than wrong. Furthermore, we utilized a second way of communication with the Mosquitto Broker and logging failures with Python loggers so this will make it very easy for us to recover the loss data and maintain the consistency of the database. We also notice that the True Positives in the heavy load experiment is comparable to the True Positives of the normal load experiment. This is because our platform cannot process much more requests at the same time, therefore it hits a ceiling. But the virtual difference in the accuracy, which is quite significant (3.12%), is mostly attributed in the failure of the system to live up to the high demand and since true negative cases are much faster to process and therefore more in number, it is easier for us to think our platform is more accurate when we get more requests when that is not the case.

We also notice that the rate of misidentifying an input for an invalid or valid falsely is quite low, around the 3% mark. One key area to notice is that the mistakenly misinterpretations of valid inputs as invalid is doubled from experiment 1 to experiment 2. Meanwhile, valid input interpreted as valid input cases drops on the second experiment. This means that because of the higher utilization of our resources, the system is unable to process the already undertaken tasks and makes a lot of error that it wouldn't make. This points to a load balancing mechanism being quite beneficial.

As far as precision is concerned in both experiments, it is kept quite high. Meaning, our system very rarely mistakes an invalid input to a valid one. This is all attributed to the type checking and validator implementation of the Django ORM which flushes out any inconsistencies with incorrect data.

We notice that our system is more sensitive by a factor of 5% as the system scales in concurrent users. This is also attributed to the inability of the system to manage higher loads of requests and therefore making mistakes with the existing requests because of timeouts as well.

Finally, our system boasts high specificity marks around the 99,9 percentage. This also attributes to the fact that the system can flush out easily any wrong information.

Performance Metrics

From an initial perspective of the API request rates of the two experiments (figure 4.2 and figure 4.10) we notice something strange. Even though the high load test experiment has more VUs than the normal load test, the median requests per second sent appears to be higher in experiment 1 than in 2. That makes sense if we think that the high volume of demand for the resources of the system by the 2nd experiment clutters the cloud platform, therefore being unable to process current request or receive request to its maximum potential.

As far as the processing power of our system, we get a better view by comparing the response time diagrams for each endpoint on both experiments. The first thing that we notice is that for experiment 1 the average response time function is fairly linear with no visible trend or frequent big deviations from the average. This is in line with our experiment structure since we have fairly lower number of VUs therefore a manageable amount of requests leads it way to the API. More importantly, for the analytics endpoints, we notice that the average response time is around 250 milliseconds(ms). Meanwhile, for experiment 2 we notice an evident upwards trend for the average response time for most endpoints with the sole difference of the batteries endpoints were at first the average response time is fairly uniform with an upwards trend on the second part and a return to uniformity for the third part. The upwards trend is very well in line with our experiment with is increasingly putting more weight on the system with the gradual increase of VUs and limitless amount of iterations per second possible. The response time for the measurements endpoints still remains in the ballpark of 300 ms and the EIS endpoints is around 1 second.

In comparison, all response times for management endpoints in experiment 2 seem to be 2 to 4 times higher than their counterparts. The measurements endpoint response times remain pretty much the same while the EIS response times scales up to 5 times higher.

Last but not least, we notice that the most intensive endpoints include all the endpoints that implement the GET all HTTP method. For up to 5 concurrent users most endpoints respond within a second while the slowest endpoint (/api/cells) responds in 5 seconds. The view is totally different when the system scales to 25 concurrent users where responses vary from 1 second up to 1 minute with the most frequent being around 30 seconds. This method get all the records of the requested table, therefore it is normal to be that intensive in processing power and in turn latency.

Key Takeaways

Considering that the main use of our platform is going to include mostly POST requests to analytics endpoints, with a frequency of 1 request per second, from the Battery Systems (that amount to a total of 4) and GET requests from 3rd party software systems that run real time analytic algorithms our system is going to live up to the specifications that are set.

Our system implements a messaging solution retaining information from the request of the BS's that have failed, therefore there is no question that even in the case of our system recognizing valid inputs as invalid the integrity of the main function of the system will remain and total consistency of the database can be very easily implemented in the future.

Special attention must be given when it is required by the system to handle larger loads, since the key points of interest will not be met, leading to an inconsistent state of the system. This will be remedied easily by the horizontal scaling of the system resources and inclusion of a load balancer.

Chapter 5

Conclusions

5.1 Work Summary

During the course of this thesis, we undertook the task of creating a cloud platform capable of storing data according to specifications given regarding the consistency and structure of the data, the response times, security, interoperability and fault tolerance.

We began by stating the problem, analyzing its importance and putting our application into the perspective of the whole Battery2Life project. Then we laid out the specifications for our platform regarding the data management, the REST API structure and the architecture of the whole solution. More specifically, our solution required the interconnection and correct implementation of the components' server, database, publish-subscribe broker and activity logger.

Following that, we proceeded with the implementation of the solution by organizing the data into data models with the help of Django ORM, organize our server request handlers into ModelViewSets with the help of Django Rest Framework, documented our API with Swagger UI, utilized docker container to execute the application in a secluded and stable environment, implemented system monitoring with Python loggers and set up all the necessary security protocols including CSRF, CORS, authentication and authorization with the help of various python modules.

Additionally, we laid out the schemes for the methodical and meaningful testing of our platform where under normal circumstances we establish a 96.86 percentage of accuracy, 99.94 percentage of precision and response times well below the crucial limit of 1 second. Ultimately, in order to ensure the Quality of Service(QoS) of our platform we also conducted a heavy load test with 25 concurrent users where we determined that the accuracy and precision metrics maintained their values with 99.98% and 99.93% respectively and response times were kept within QoS acceptable values.

Although our platform works very well under normal circumstances, it isn't bulletproof to all scenarios. From the last experiment we deduced that when the number of user scales, the response times and incorrect responses also have an upwards trend.

5.2 Future Work

Software systems are a dynamic entity and always have a tendency to evolve along with the system requirements over time.

To name a few possible evolutions, if we consider that our platform will receive at least one POST request per second and the request will contain data in the ballpark of 500 bytes, then we can easily deduce that per month the system is going to be required to handle and store approximately 1.3 terabytes of data. For this reason, further consideration should be given to the architecture of the application regarding the storage of all of those data as well as the management of the communication of all of these data with 3rd parties.

When handling such amounts of data, even if the tiniest of probabilities that something's is going to go wrong exists, this means that things are going to go wrong quite often. For this reason, special care needs to be given to implement the handling of HTTP requests (especially POST) by the Mosquitto broker. Furthermore, several protocols, such as handling failed status of responses with care in order to ensure the consistency of the database, should be put in place.

For the availability and scaling of the system, a configuration that deploys the services in a Kubernetes cluster should be considered.

Furthermore, with the mature and implementation of the Digital Battery Passport administrators should update and extend the schema of the database with useful information derived from the real deployment of the system virtually at will.

Last but not least, with the integration of the system into the general solution of Battery2Life future maintainers should consider real world feedback and enhance the system with features in an agile way.

Bibliography

[1] European Commission and Project Consortium. Battery2life — horizon-cl5-2023-d2-01: Description of action (doa). Grant Agreement No. 101137615, Unpublished document, December 2023.

[2] The Battery Passport Initiative. Battery passport technical guidance, 2024. Accessed: 2025-02-18.

[3] The Battery Pass Consortium. Battery passport content guidance, December 2023. Version 1.1.

[4] Battery Pass Consortium. 2023 battery passport content guidance. Technical report, Battery Pass Consortium, 2023.

[5] Battery Passport. 2024 battery passport value assessment, 2024. Accessed: 2025-02-20.

[6] Mattia Gianvincenzi, Marco Marconi, Enrico Maria Mosconi, and Francesco Tola. A standardized data model for the battery passport: Paving the way for sustainable battery management. *Procedia CIRP*, 122:103–108, 2024.

[7] DIN and DKE. Din dke spec 99100: Technical specification for the eu battery passport, 2024. Accessed: 2025-02-20.

[8] CSEM and EPFL. Wifi module documentation. Battery2Life — HORIZON-CL5-2023-D2-01, Grant Agreement No. 101137615, Unpublished document, November 2024.

[9] ICCS. D2.3: Bms cloud platform (version 1.0). Battery2Life — HORIZON-CL5-2023-D2-01, Grant Agreement No. 101137615, Unpublished document, December 2024.

[10] GeeksforGeeks. Monolithic vs microservices architecture, 2023. Accessed: March 6, 2025.

[11] IBM. What is a rest api?, 2023. Accessed: 2023-02-26.

[12] Django Software Foundation. Django 5.1 documentation, 2023. Accessed: 2023-02-26.

[13] Django REST Framework. Django rest framework documentation, 2023. Accessed: 2023-02-26.

[14] Simple JWT. Simple jwt documentation, 2023. Accessed: 2023-02-26.

[15] ITservices Expert. Django security best practices: Fortifying your web application, 2022. Accessed: 2023-02-26.

[16] Django Software Foundation. The django admin site, 2023. Accessed: 2023-02-26.

[17] Django REST Framework. Routers, 2023. Accessed: 2023-02-26.

[18] Simeon Emanuilov. Is django still relevant in 2024?, 2024. Accessed: February 24, 2025.

[19] Django REST Framework. Viewsets, 2023. Accessed: 2023-02-26.

[20] Swagger. Swagger ui, 2025. Accessed: 2025-02-26.

[21] PostgreSQL Global Development Group. *PostgreSQL Documentation*. 2022. Accessed: 2024-02-24.

[22] Databricks. Acid transactions, 2025. Accessed: 2025-02-26.

[23] Django Software Foundation. Django class-based view mixins, 2025. Accessed: 26-Feb-2025.

[24] Docker Documentation. *Docker Documentation*, 2025. Accessed: 2025-02-26.

[25] Docker Inc. Docker hub, 2025. Accessed: 3 March 2025.

[26] Git Documentation. *Git Documentation*, 2025. Accessed: 2025-02-26.

[27] Scikit learn Developers. *Scikit-learn: Machine Learning in Python*, 2023. Documentation on Classification Metrics.

[28] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[29] k6 Team. *k6 Documentation*, 2023. Official documentation for the k6 load testing tool.

[30] Grafana Labs. *Grafana Documentation*, 2023. Official documentation for Grafana monitoring and visualization.