POLITECNICO
MILANO 1863

Design Document

eMall – e-Mobility for All

Software Engineering 2, 2022/23

Giulia Huang, Zheng Maria Yu, Linda Zhu

V1.1 2023/01/17

Contents

1 Introduction

1.1 Purpose

In the last years, the high carbon footprint has been a global challenge for its negative environmental impact, since it plays a relevant role in climate change and air pollution. In particular, the transportation sector is one of the primary sources of greenhouse gas emissions due to the combustion of fossil fuels.

Electric mobility represents a solution to alleviate the problem: electric cars require less energy and produce less polluting gas as well. However, people should take into account the charging-related issues when using electric vehicles: they need to find available charging stations, and to consider the charging time of the cars too.

To facilitate the use of electric vehicles, the eMall (e-Mobility for All) system proposes to keep together and to coordinate all the activities that the charging process would require. In fact, The eMall system connects the various providers involved in the activity: e-Mobility Service Providers (eMSPs) help the drivers in planning and completing the charging process of their electric vehicles introducing minimal interference and constraints with respect to their daily schedule, while Charging Point Operators (CPOs) manage the changing stations, offer functionalities through their own Charge Point Management System (CPMS), and eventually acquire energy from Distribution System Operators (DSOs).

This document aims to introduce the overall design of the system, focusing on both architectural aspects with the description of components, interfaces and interactions, and the user interfaces as well. It is considered the basis for the development of the system, and it also proposes a plan for implementation, integration and testing activities.

1.2 Scope

The present document will focus on the analysis of the eMall system design, which adopts a 3-tier architecture consisting of a presentation tier, an application logic tier and a data tier. Moreover, a particular attention is given to the two component subsystems eMSP and CPMS and to their interactions.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Definition	Description
Electric vehicle	Vehicle with battery and electric motors for propulsion
Driver	User of electric vehicles
Energy	Power used by electric vehicles
Charging station	Location where the electric vehicles can be charged, it could have batteries to store energy
Charging column	Structure positioned in charging stations to charge electric vehicles. It has a display screen and a charging socket
Charging socket	Connector to plug in electric vehicles. There are different types of charging sockets according to the power output, such as slow/fast/rapid
Operator	Charging station administrator working for a CPO
User	Person using the system, can be a driver or an operator
Reservation	Booking made by a driver for charging at a specified time frame

1.3.2 Acronyms

Definition	Description
RASD	Requirements Analysis and Specification Document
DD	Design Document
eMall	e-Mobility for All
eMSP	e-Mobility Service Provider
CPO	Charging Point Operator
CPMS	Charge Point Management System
DSO	Distribution System Operator
DBMS	Database Management System
API	Application Programming Interface

1.3.3 Abbreviations

Abbreviation	Description
RX	Requirement number X

1.4 Revision history

- V1.0: Initial version
- V1.1: Minor modification of overview and integration diagrams

1.5 Reference Documents

- Project specification: "Assignment R&DD A.Y. 2022-2023 v3"
- Software Engineering 2 Course slides A.Y. 2022-2023
- RASD of the eMall system V1.0

1.6 Document Structure

This document is composed of seven sections described as follows:

Section one consists in a general introduction to our project with the required background information.

Section two focuses on the architecture of the system. An overview on architectural choices and system components is given, followed by the component diagram, the description of each component and the details of the interfaces. The system infrastructure is defined by the deployment diagram, and the dynamics of the interactions are shown through sequence diagrams.

Section three presents the user interface of the system through the mockups.

The requirements traceability matrix is reported in Section four. It shows the mapping between the system requirements and the designed components, making sure that all the requirements are satisfied by the chosen architectural design.

Section five describes and justifies the plan for implementation, integration and testing activities of the system.

The presentation of the effort spent by each team member can be found in Section six. The number of hours spent on each activity is reported.

Section seven contains the references used in writing this document.

2 Architectural Design

2.1 Overview

A three-tier architecture consisting of a presentation tier, an application logic tier and a data tier is chosen for the e-Mall system, mainly due to the following reasons:

- Module independence: the three tiers can be developed independently at the same time.
- Security: it guarantees major security, since clients cannot have direct access to the database.
- Re-use of data: the database can be re-used by other applications easily.

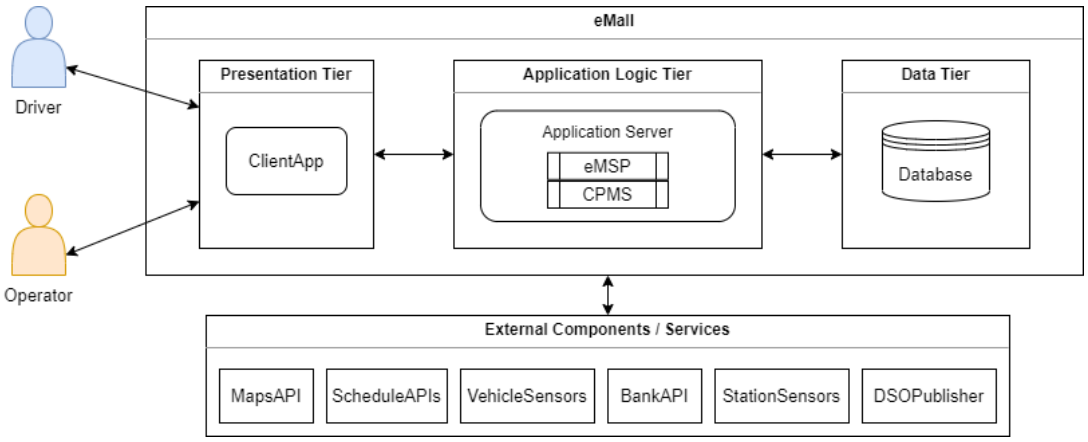

Figure 1: High-level overview of the eMall system architecture

A description of the three tiers is reported below.

- **Presentation tier**: it is the top-level tier of the designed mobile application with a graphical user interface. It interacts with the users by displaying information and collecting inputs from them, and it communicates with the application logic tier.
- **Application logic tier**: this middle tier consists in the application server, which supports the system's functionalities by processing the data collected by the presentation layer and handling the operations concerning the database. In our case, it supports the activities of both eMSP and CPMS subsystems, and it also deals with various external components and services.

- **Data tier**: it is responsible for persistent data storage, and it can be only accessed by the application server.

2.2 Component view

Here the component diagram offers a general view of the system's components and their interactions. In addition, all the components are presented and described.

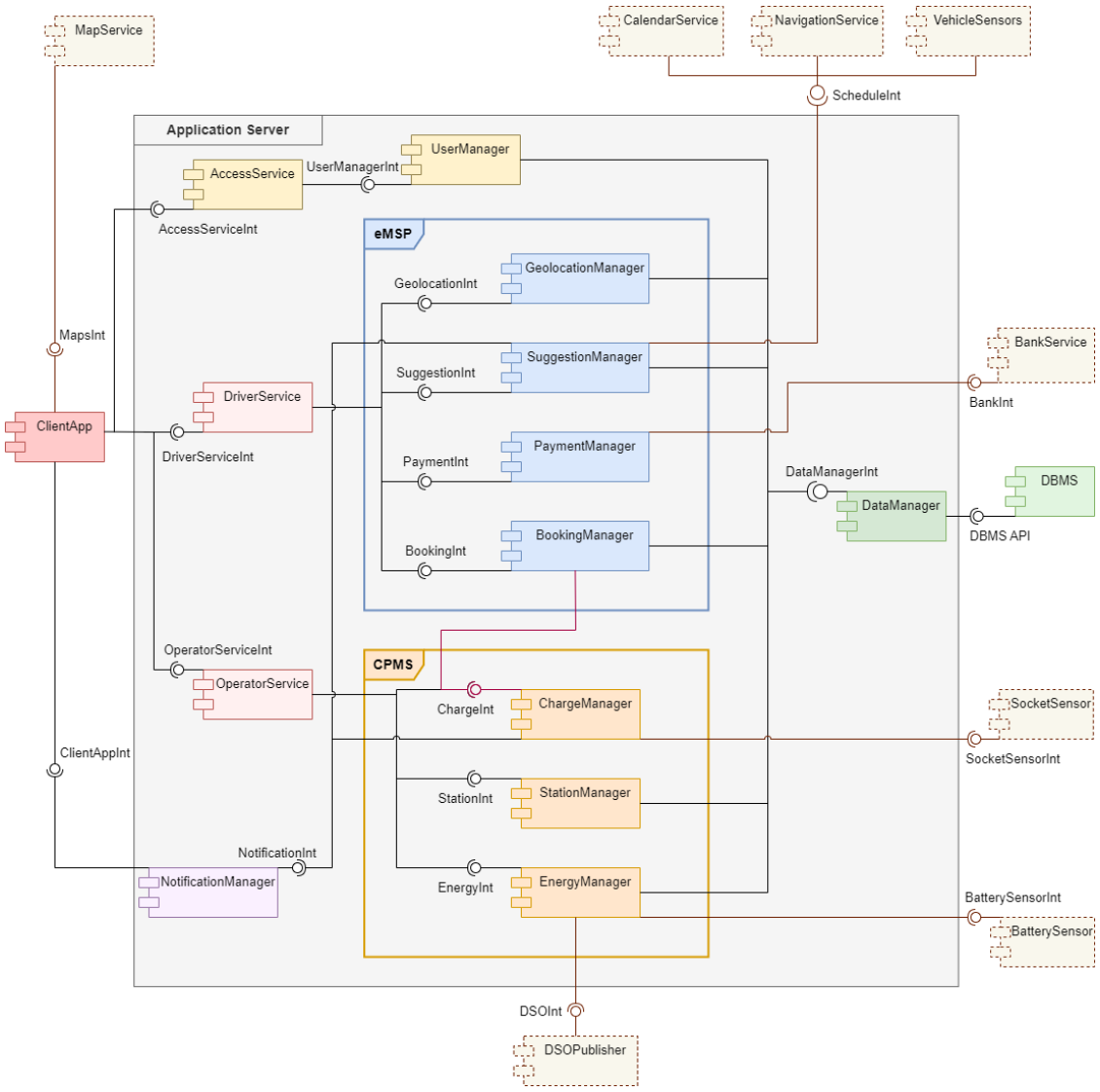

Figure 2: Component Diagram of the eMall system

2.2.1 System Components

The system's architectural components are:

- **ClientApp** represents the mobile application client, and it generates views for the user based on the responses from the server. It communicates with the external MapService for map functionalities.
- **AccessService** handles the registration process, the login process, and the modification of the account's personal data. It also gives the correct authorization role to the user after the successful completion of the authentication.

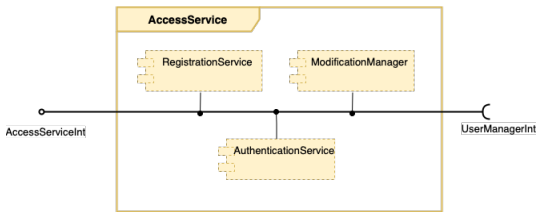

Figure 3: Sub-components of AccessService component

  It consists of the following sub-components:

  - **RegistrationService** is responsible for the registration process of the user (only for Driver).
  - **AuthenticationService** is responsible for the authentication process for both Driver and Operator. If the operation is successful, the user's login token is stored client-side in a encrypted way.
  - **ModificationManager** manages the modification of data regarding the user's account.

- **UserManager** interacts with DataManager, retrieving and storing user-related data for the creation of an account, the verification of credentials and the modification of account's personal information.
- **DriverService** handles the available operations for a driver: visualize the nearby charging stations and their external status, book a charge, start and end a charging process, pay for the received service, and receive charging suggestions. The user authorization is checked each time before accessing to the methods.

- **GeolocationManager** is responsible for the visualization of the charging stations, along with their respective charging costs, special offers, and charging availabilities. It gets the necessary information about charging stations from DataManager.

- **BookingManager** deals with the procedures related to the operation of booking a charge. It allows the driver to book a charge, to start the charge according to the reservation, and to end the charge. The information about the driver account's linked vehicles is obtained from UserManager, and it interacts with DataManager to retrieve the charging station's availability, to register a new reservation and to update the reservation's status. Moreover, it communicates with ChargeManager to start and end the charging process at the chosen charging station.

- **PaymentManager** activates the payment process of the driver after the completion of a charge. It communicates with an external BankService to process the payment.

- **SuggestionManager** is responsible of preparing the charging suggestions for the driver according to his schedule. The suggestions are made with respect to the stations' availability got from DataManager, and the user schedule, thanks to the information pulled periodically from the external APIs.

- **OperatorService** handles the available operations for a charging station's operator: check the status of the charging station, promote a special offer, modify the charging cost, modify the source of energy used for the charges, and acquire energy from DSOs. The user authorization is checked each time before accessing to the methods.

- **StationManager** is responsible of visualizing the external and internal status of the charging station, and it allows the operator to modify the charging costs and to promote special offers. It interacts with DataManager to update the available costs and the offers.

- **ChargeManager** allows to start and end a charging process at the station according to the reservation. It gets information about the vehicle connected to the charging column through SocketSensor.

- **EnergyManager** is responsible of modifying the energy source for charges, checking the level of batteries of the charging station, and handling the energy acquisition process. It communicates with the external DSOPublisher to get

information about energy costs and to acquire energy from DSOs. Moreover, it updates the energy source status by interacting with DataManager. If batteries are present, it also deals with the BatterySensor of the station.

- **NotificationManager** sends the notifications generated by the server to the user's device. In particular, SuggestionManager uses it to send charging suggestions, and ChargeManager uses it to notify the user when the vehicle's battery is fully charged.
- **DataManager** handles the data requests of the other system components. According to the operation, it retrieves or stores proper information by interacting with the DBMS, making queries or updating it.
- **DBMS** contains the system's persistent database with different schemas.

2.2.2 External Components

On the other hand, the external components and services are:

- **MapService** offers the possibility to visualize maps with the charging stations' location, and it also allows to search for specific addresses or station names. Its functionality of device localization requires a working GPS sensor of the mobile phone.
- **CalendarService** allows the system to get access to the driver's calendar information.
- **NavigationService** allows the system to get access to the driver's navigation system, in order to know about his paths.
- **VehicleSensors** offers the access to the electric vehicle's sensors, and the system can acquire the battery level information of the vehicle in this way.
- **BankService** handles the operation's payment with respect to the chosen method and the credit information inserted by the driver. The supported payment options are credit/debit cards and mobile payment systems such as PayPal, Apple Pay and Google Pay.
- **SocketSensor** monitors the status of the electric vehicle connected to the charging column's socket. In particular, it is able to check the vehicle's number plate and its battery level in time.
- **BatterySensor** monitors the status of the batteries of the charging station, if present.
- **DSOPublisher** handles the interactions with DSOs: it allows the system to know the energy costs published by the DSOs, and to acquire energy from them with respect to the chosen option.

2.2.3 Entity Relationship Diagram

The high-level entity relationship diagram of the database is shown below.

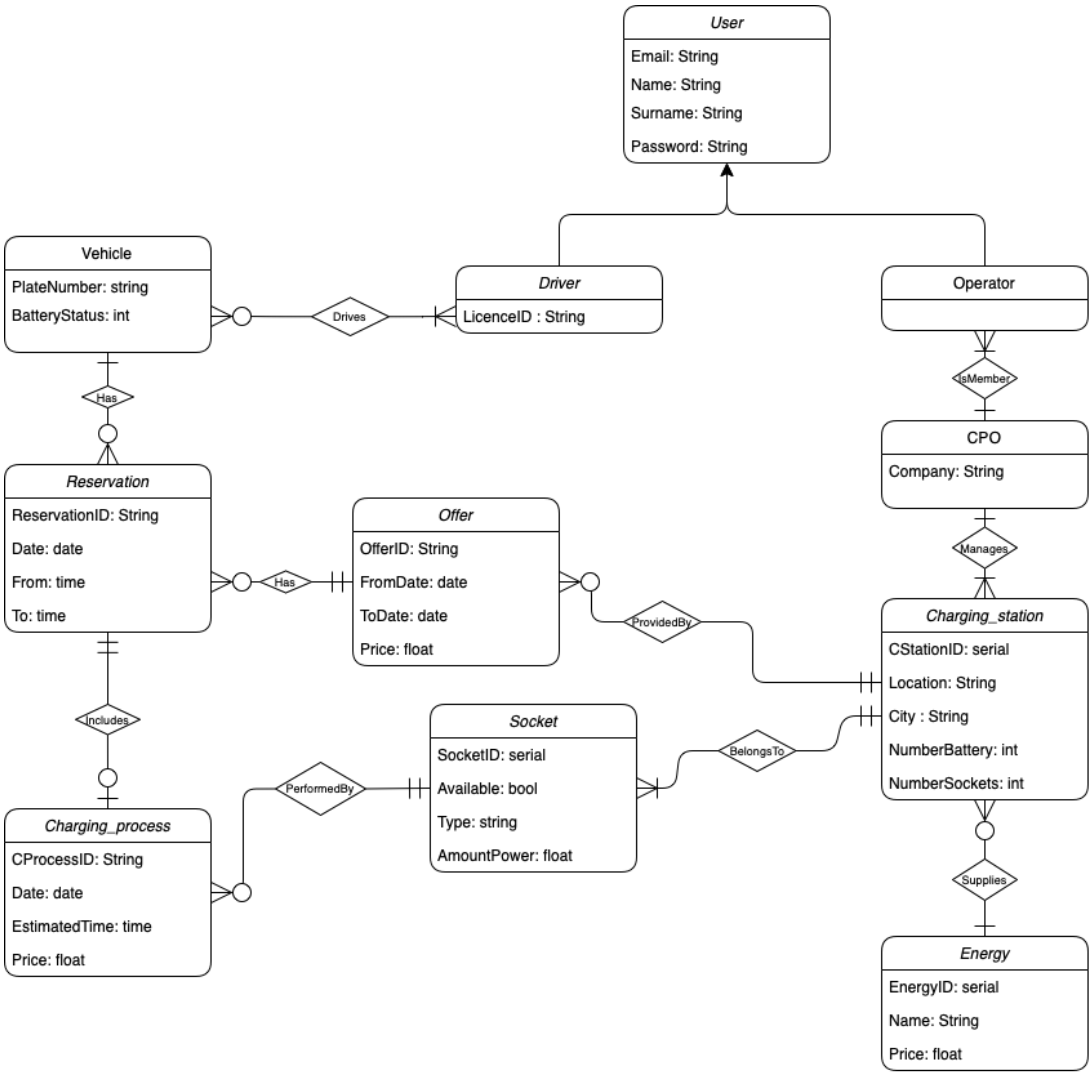

Figure 4: Entity Relationship Diagram of the e-Mall database

From the previous ER schema, it is possible to derive the following logical model:

Driver (LicenceID, Email, Name, Surname, Password)

Operator (Email, Name, Surname, Password)

CPO (Company)

ChargingStation (CStationID, Location, City, NumberSockets)

Energy (EnergyID, Name, Price)

Socket (SocketID, Available, Type, AmountPower)

Offer (OfferID, FromDate, ToDate, Price)

Vehicle (PlateNumber, BatteryStatus)

Reservation (ReservationID, Date, From, To)

ChargingProcess (CProcessID, Date, EstimatedTime, Price)

2.3 Deployment view

In this section a deployment diagram of the system is presented, it describes the environments and tools used to build the eMall system. The communication protocols between the various parts are also specified.

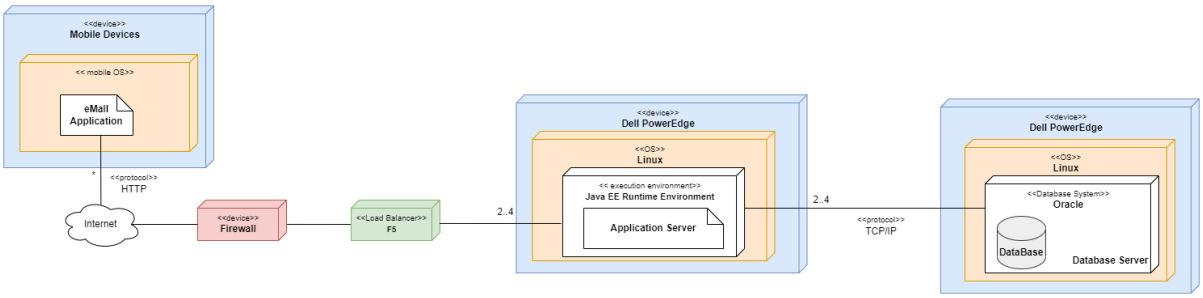

Figure 5: Deployment Diagram

- **Mobile Devices** are smartphones and tablets where the Users (drivers and operators) install and run eMall Application. These devices need to connect to the system via Internet. The Client End will be separately developed for the different kind of operating systems, such as IOS, Android and HarmonyOS.
- **Firewall** manages the dataflow from clients to servers by filtering the packets from the Internet for security issues.
- **Load Balancer** distributes the workload among available resources to achieve availability and reliability. The idea is a load balancer with Least Connections method to balance the workload between the available application servers.
- **Application Server** contains all the application logic of the system and interacts with the Database Server. It should be replicated to increase reliability and availability. The Server device use Linux OS and the application server runs in a Java EE Runtime Environment, because it allows to develop distributed applications through standardized and modular components, with the aid of various integrated services. For example, the Persistence API can be used for connecting and manage the DBMS thanks to the object/relational mapping, and a REST API can be easily implemented for web services through JAX-RS.
- **Database Server** hosts the system's Database, which is accessible through MySQL.

2.4 Runtime view

This section contains the runtime sequence diagrams about the eMall system's main functionalities, considering the use cases analyzed in the RASD. The diagrams describe the dynamics of the interactions among system's components.

- Register a new account

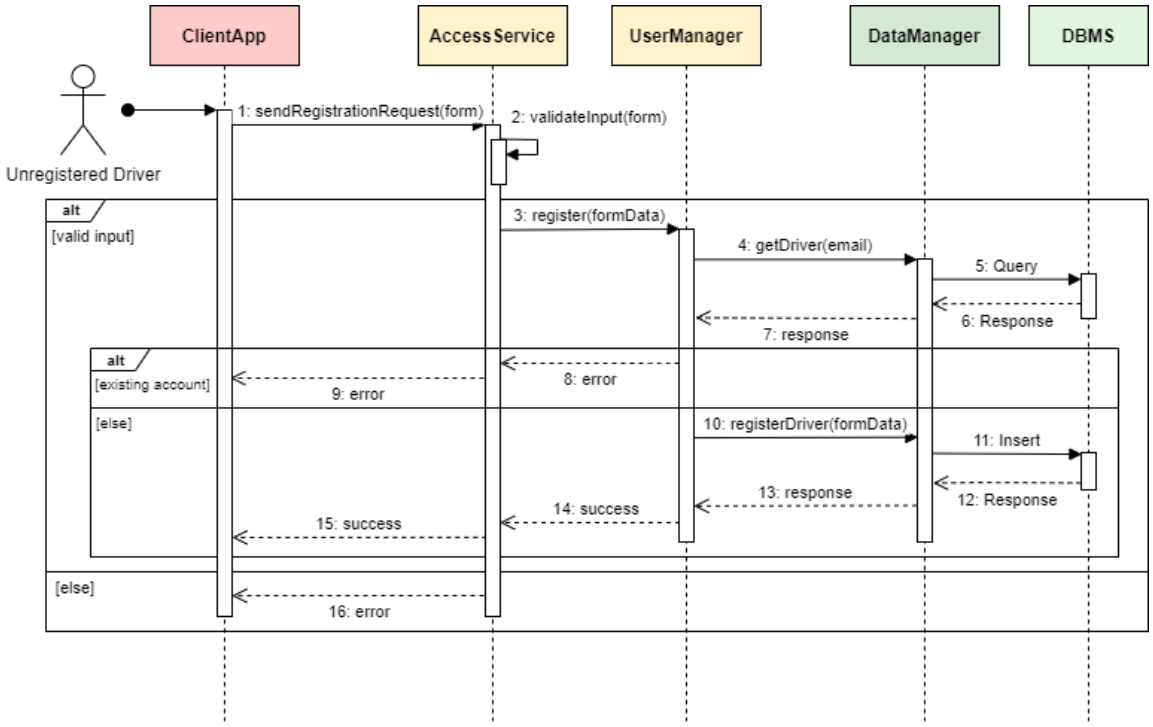

Figure 6: Runtime View of Driver Registration

The sequence diagram above shows the driver registration process. The unregistered driver is on the Login page of the eMall application and he chooses to register an account. The registration form is displayed by clicking on the "Register" button. After filling the form, the driver clicks on "Confirm" and the registration request is sent to AccessService, which begins to handle the operation. An error message will be shown if the procedure fails, e.g. the input is invalid, or the email address is already associated to an account. Otherwise, the driver will see a success message.

- Login

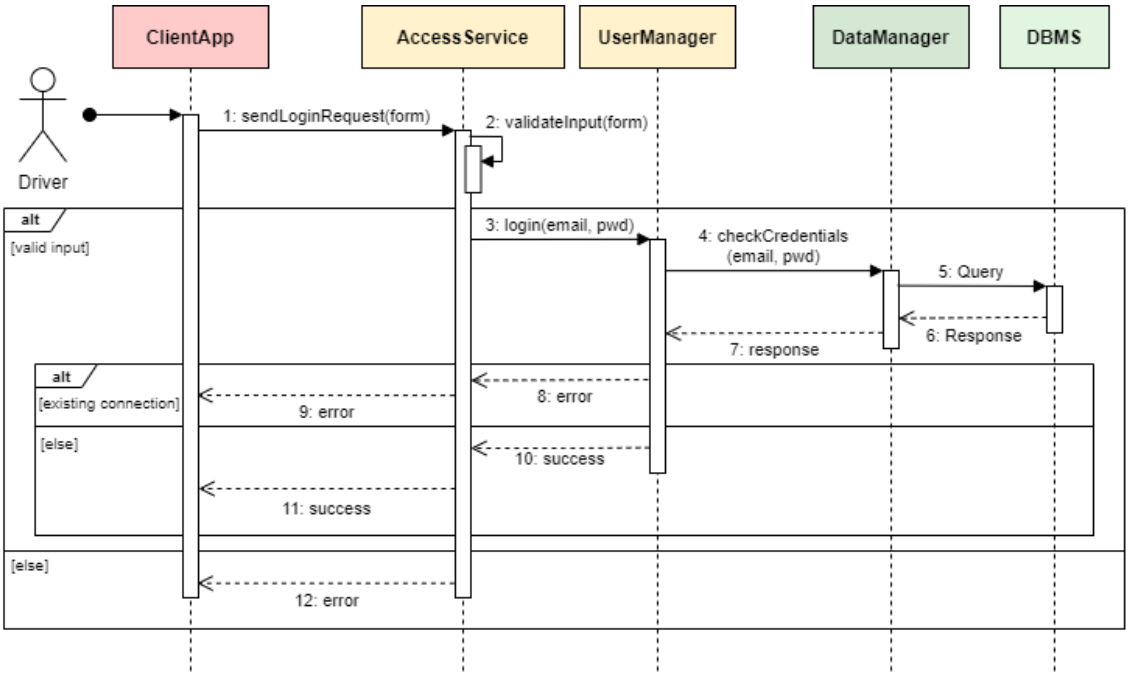

Figure 7: Runtime View of User Login

The sequence diagram above shows the login process. The user (driver or operator) is on the Login page of the eMall application, and he clicks on the "Login" button. The request is sent to AccessService after filling and submitting the login form. If a success response is received, now the user is correctly authenticated and he is redirected to the corresponding Main Page according to his role.

- Add a vehicle to the Driver profile

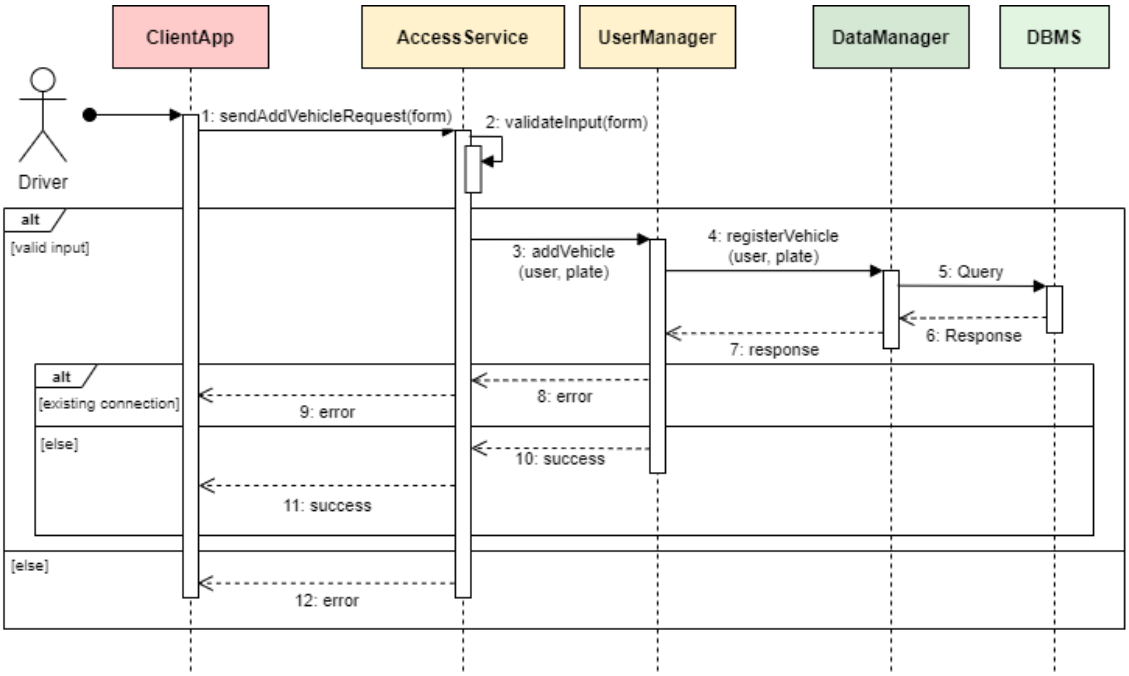

Figure 8: Runtime View of adding a vehicle to the profile

The sequence diagram above shows an example of updating the account's personal information, more specifically we deals about adding a vehicle to the driver's profile. The driver is correctly logged in, and he finds the option to add a vehicle in his personal profile page. The request is processed by AccessService, which calls UserManager to complete the procedure. Then the vehicle is linked to the driver's account if the input data is valid and the vehicle has not been connected to this account yet.

- Visualize charging stations nearby

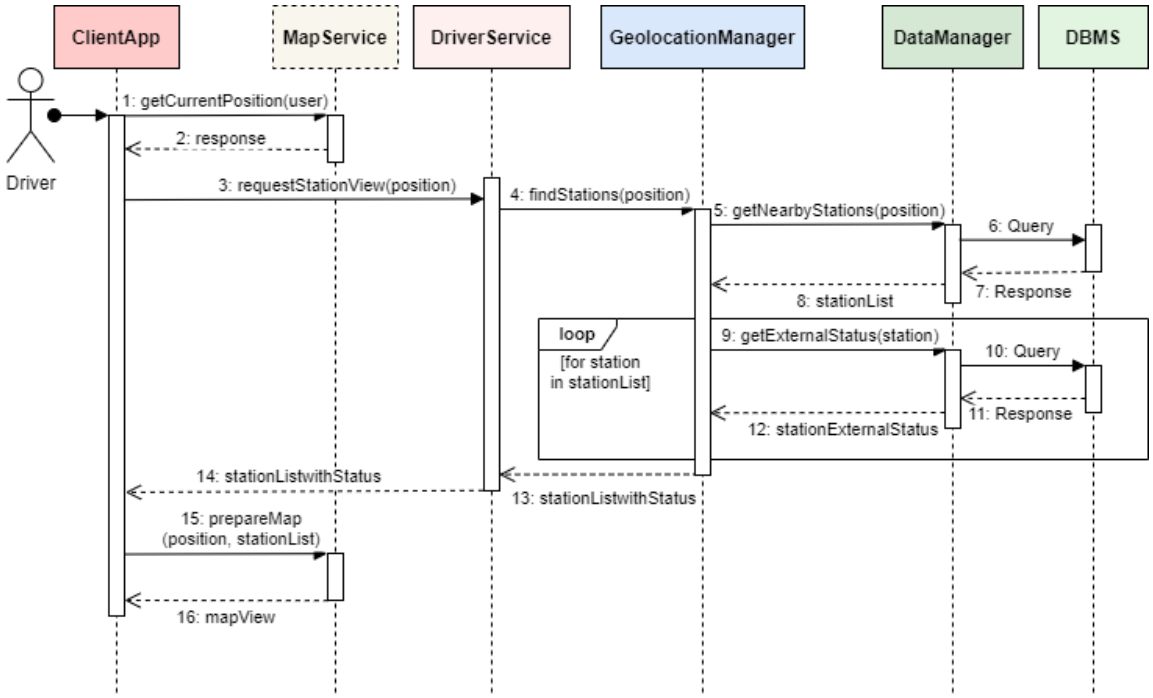

Figure 9: Runtime View of visualizing charging stations nearby

The sequence diagram above describes the process of visualizing nearby charging stations on the map. The application gets the device's current position through the external MapService, and DataManager makes query to the DBMS in order to find the nearby stations and their status. The retrieved information is processed by GeolocationManager and sent to the client, which asks MapService to prepare the view of the map.

- Book a charge

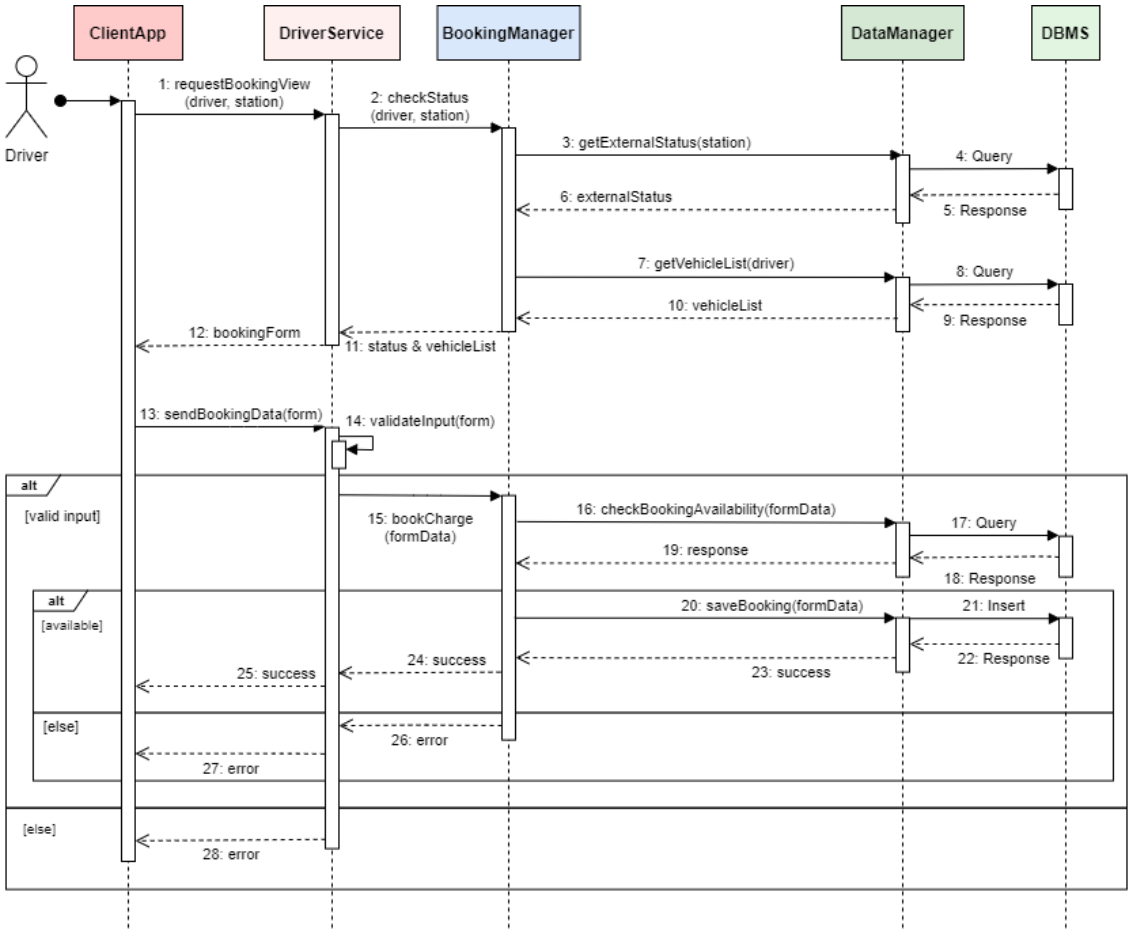

Figure 10: Runtime View of booking a charge

The sequence diagram above describes the process of booking a charge. The logged driver chooses a charging station to initiate the booking process. BookingManager checks the external status of the selected station and the list of vehicles of the driver, and then a booking form is displayed. After submitting the form, BookingManager handles the registration of the reservation if the data is valid and the option is available.

- Start a charge

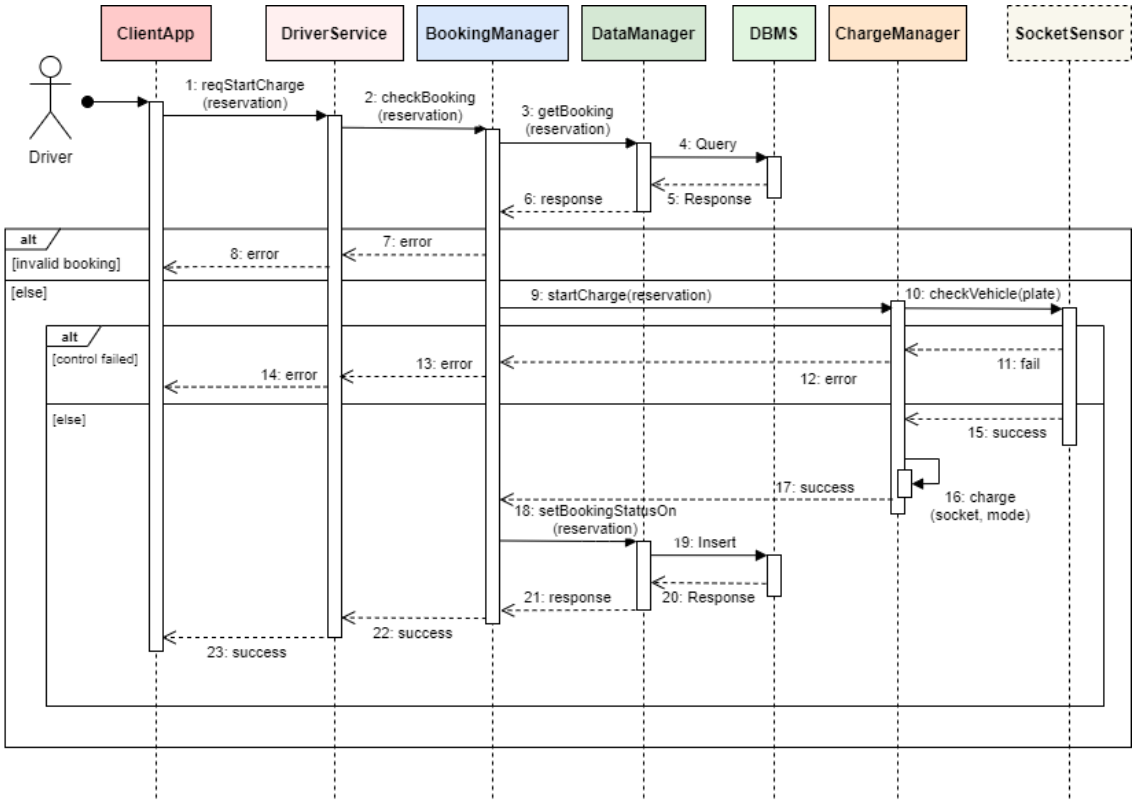

Figure 11: Runtime View of starting a charge

The sequence diagram above shows the procedure to start a charging process. The driver arrives at the charging station, plugs the vehicle in the assigned charging socket, and sends a start charge request from the eMall application on his device. BookingManager checks the validity of the reservation and asks ChargeManager to handle the charging process at socket level. If the vehicle has begun charging correctly, it becomes locked and the booking status is updated to "ON" in the database.

- Finish a charge

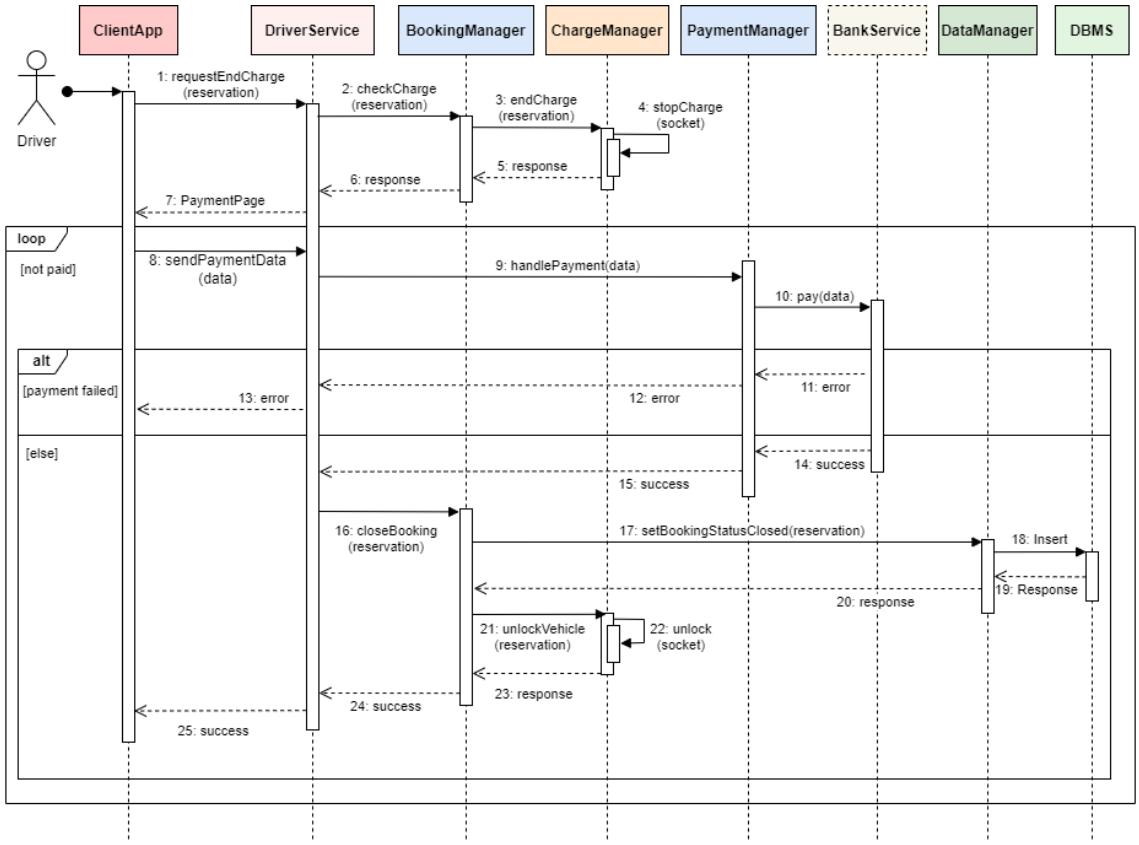

Figure 12: Runtime View of finishing a charge

The sequence diagram above shows the procedure to end a charging process. The driver receives the notification that the vehicle's battery is fully charged, and he arrives at the charging station. He confirms to end the charging process in the application. BookingManager checks the charging status and ChargeManager stops the power supply of the socket.
Then the driver is asked to pay for the obtained service, and the payment is processed with the help of the external BankService. If the payment is successful, the booking status is updated to "CLOSED" and the vehicle is unlocked at this point.

- Suggest the driver to charge the vehicle

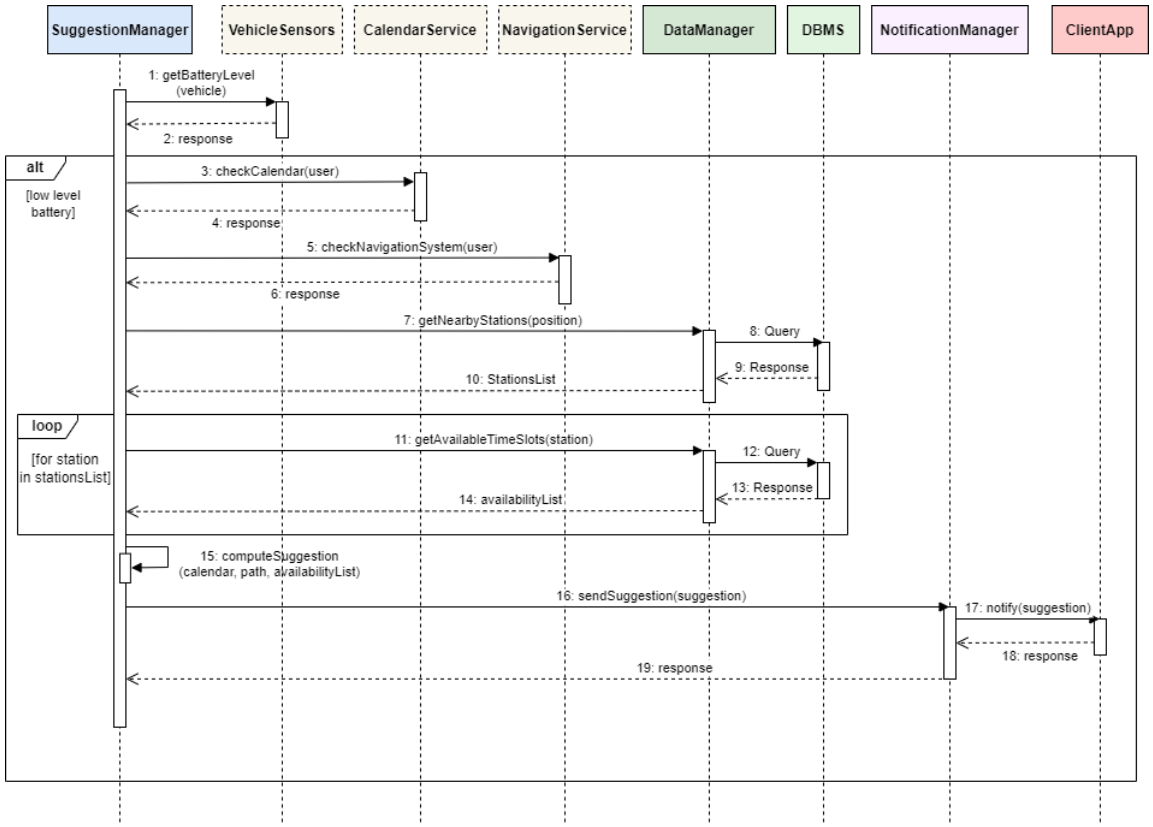

Figure 13: Runtime View of charging suggestions

The sequence diagram above illustrates the process of making suggestions to the driver for the vehicle's charging.
SuggestionManager periodically checks the vehicle's battery status through VehicleSensors during the day. It retrieves information about the driver's schedule from CalendarService and NavigationService if the low level battery status is detected. In addition, it interacts with DataManager to find the nearest charging stations with the respective available time slots. Then it computes the best plans and sends the suggestions to NotificationManager, which notifies the Client Application.

- Check the overall status of charging station

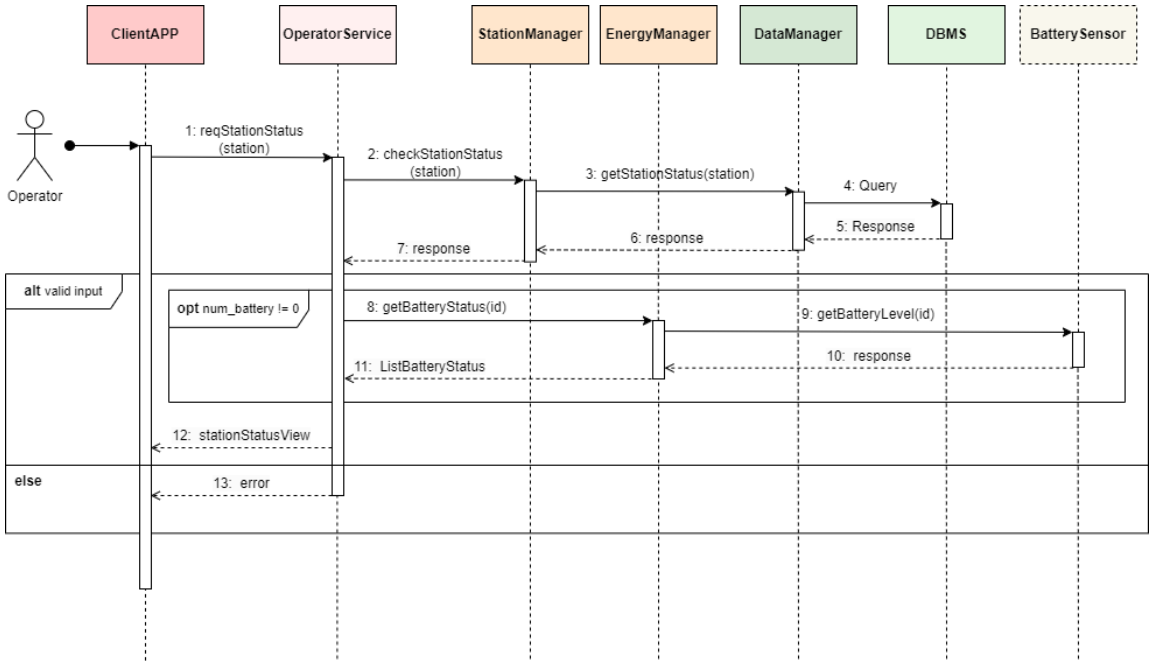

Figure 14: Runtime View of checking the charging station status

The sequence diagram above shows the process that a logged operator checks the status of a chosen charging station. StationManager retrieves the information about the station stored in the database, while EnergyManager communicates with BatterySensor to get the level of the batteries in the station if they are present. Then the overall view is displayed to the operator.

- Define a special offer

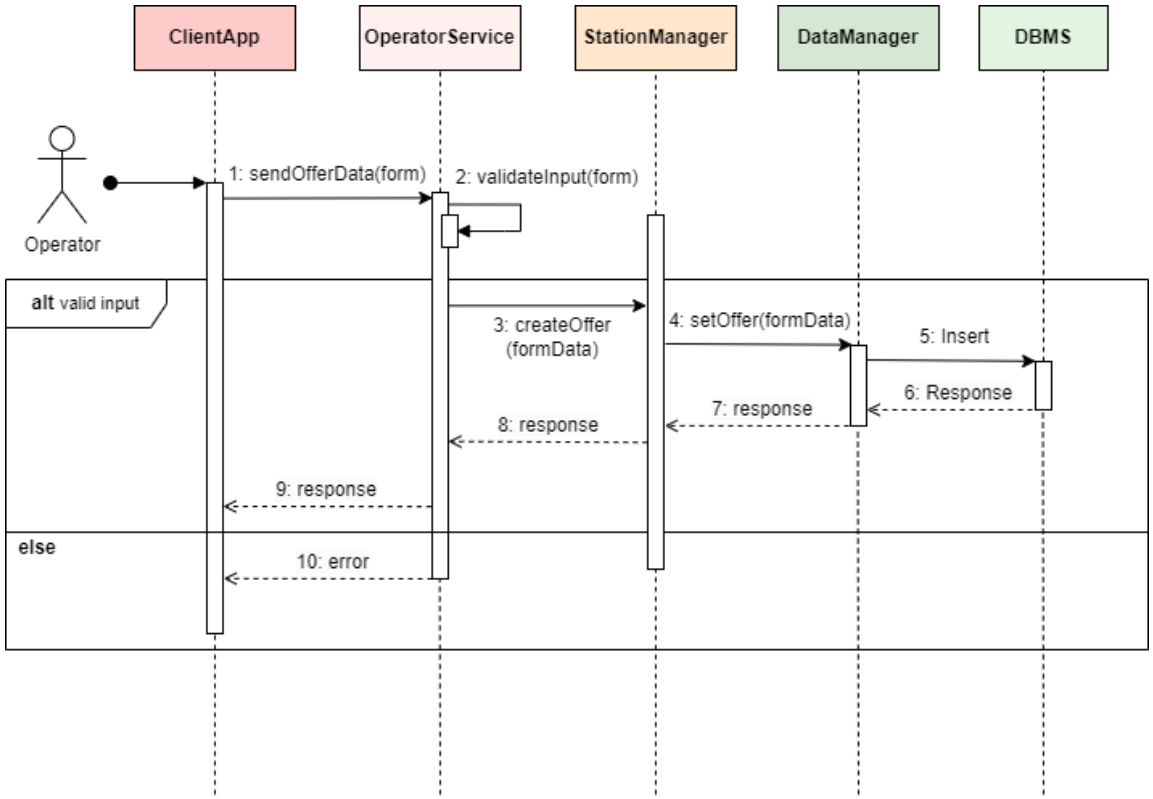

Figure 15: Runtime View of defining a special offer

The sequence diagram above shows the process of defining a special offer for a chosen charging station. The operator submits the form with the required information (socket type, cost, start and end dates), and StationManager process it. If the input is valid, the offer is updated in the database.

- Modify the charge cost

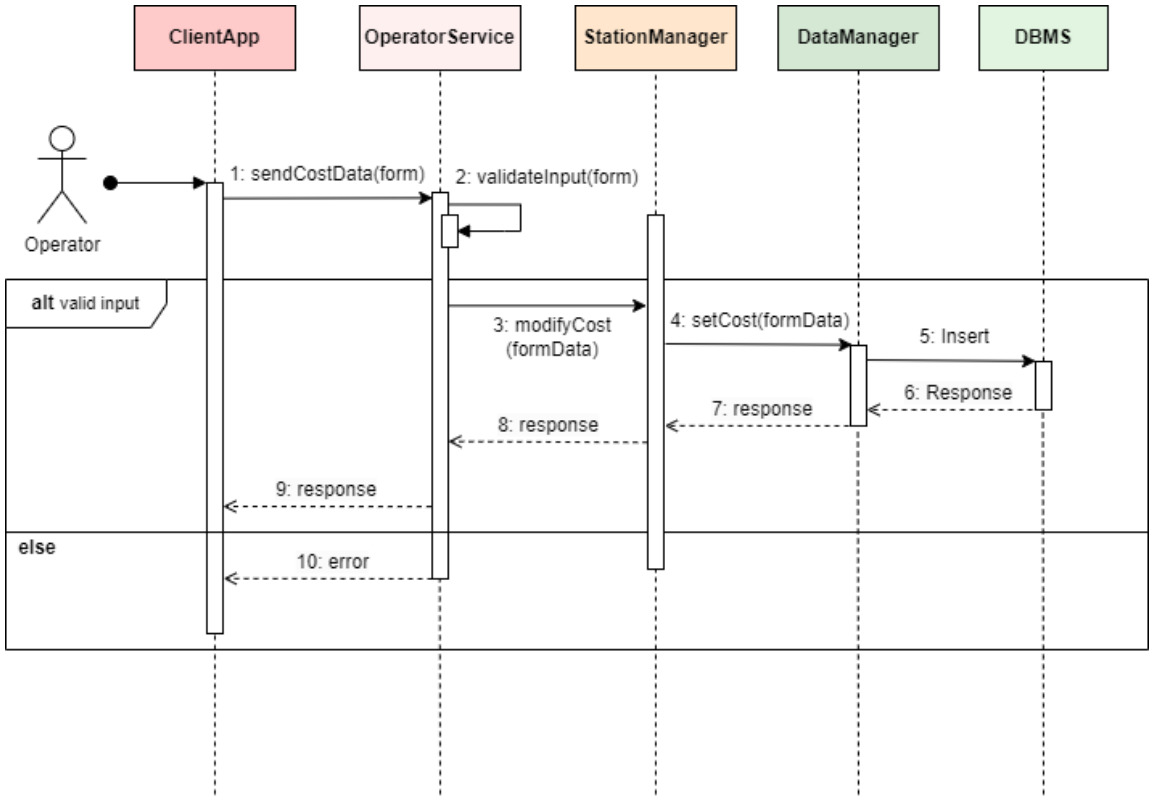

Figure 16: Runtime View of modifying the charge cost

The sequence diagram above shows the process of modifying the charge cost of a chosen charging station. The operator submits the form with the required information (socket type and cost), which is processed by StationManager.

- Acquire energy from DSO

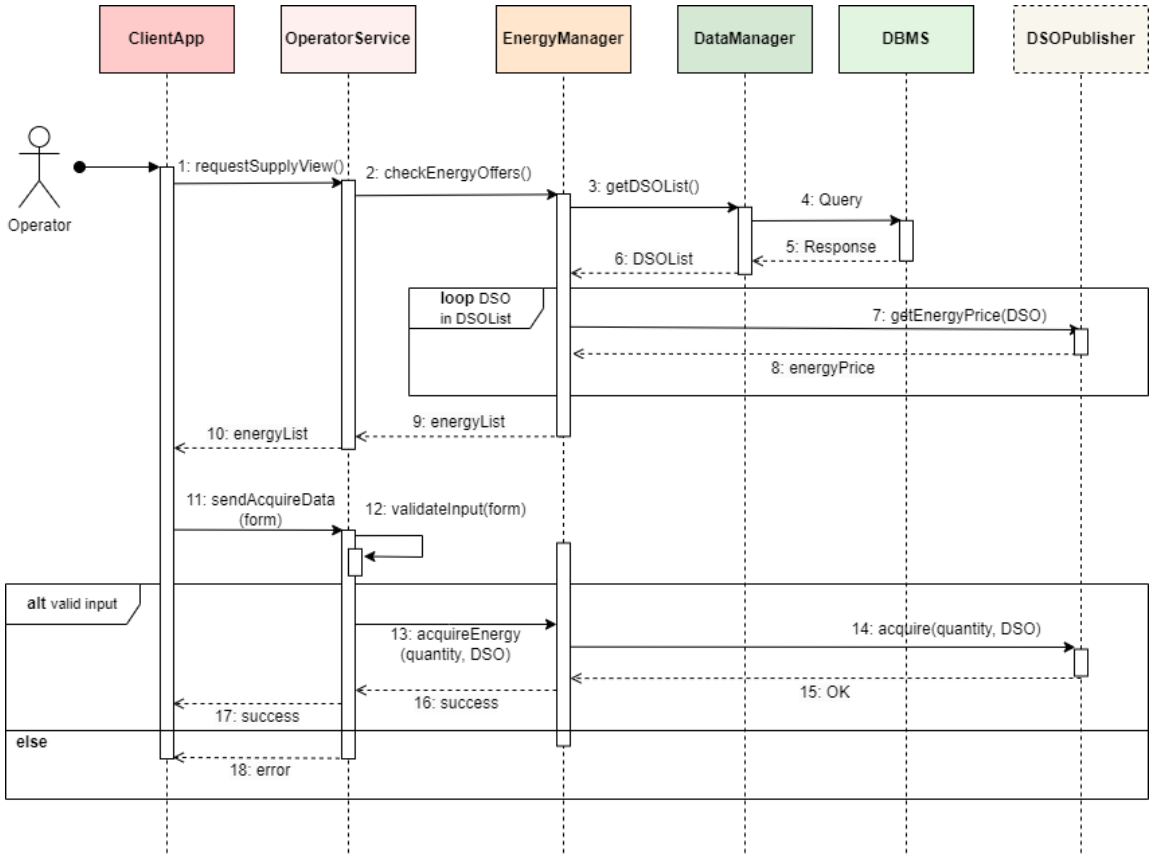

Figure 17: Runtime View of the energy acquisition process

The sequence diagram above describes the process of acquiring energy from DSOs. EnergyManager communicates with DataManager to retrieve the list of DSOs, and it interacts with the external DSOPublisher to get the current energy offers of each DSO. If the operator wants to buy energy for the charging station, he submits the form for the energy acquisition which will be handled by EnergyManager and DSOPublisher.

- Modify the energy source for the charges

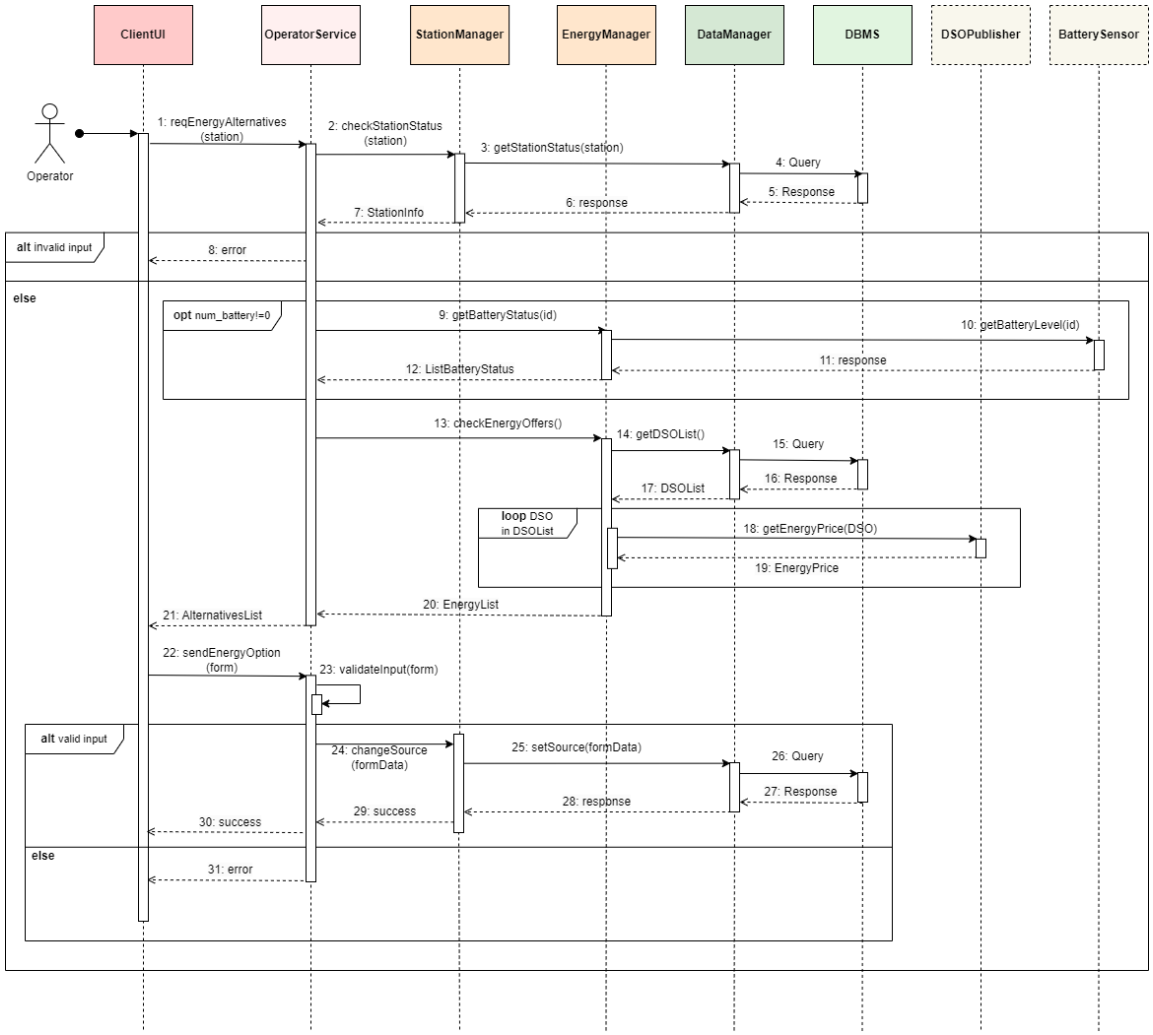

Figure 18: Runtime View of modifying the energy source for the charges

The sequence diagram above shows the operation of modifying the energy source for charging the vehicles at the station. The logged operator requests the alternatives for energy supply, and the station status is checked. Then EnergyManager controls the level of batteries of the station and gets the current energy offers from DSOPublisher. The list of alternatives is then displayed to the operator. After choosing an option, the process of changing the energy source is handled by StationManager.

2.5 Component interfaces

The component interfaces are presented in detail in the following part.

- **AccessServiceInterface** is accessible from ClientApp.
  - **sendRegistrationRequest(form)**: it takes as input the data filled in the account registration form, which includes the driver's name and surname, the email address, the driving licence number and the password. It returns a success message if the registration is correctly completed, otherwise an error message is shown.
  - **validateInput(form)**: the data format of the form fields is checked. It returns False if the data is formatted wrongly, otherwise True.
  - **sendLoginRequest(form)**: it takes as input the user login form containing the email address and the password. It returns a success or an error message.
  - **sendLogOutRequest(email)**: it takes as input the user's email address and the password. It returns a success or an error message.
  - **sendAddVehicleRequest(form)**: the number plate of the vehicle that the driver wants to add to the own profile is recorded and wrapped in a form along with the account identifier as the function's input. If the operation is successful, it returns a success message, otherwise the driver is informed of the error.
  - **sendDelVehicleRequest(form)**: the number plate of the vehicle that the driver wants to delete from his own profile is recorded and wrapped in a form along with the account identifier as the function's input. If the operation is successful, it returns a success message, otherwise the driver is informed of the error.
- **UserManagerInterface** is accessible from AccessService, and it offers functionalities for the retrieval and the storing of user-related data in the database.
  - **register(formData)**: it takes as input the registration request processed by AccessService, and it returns an error or a success message with respect to the registration's outcome.
  - **login(email, pwd)**: it takes as input the email address and the password inserted by the user from AccessService, and it returns the authorization token for the user's current session if the login is correctly done, otherwise it returns an error.

  - **logout(email)**: it takes as input the email address of the user and it returns an error or a success message with respect to the operation's outcome.
  - **addVehicle(user, plate)**: it takes as input the driver identification and the number plate of the vehicle to be added. It returns an error or a success message with respect to the operation's outcome.
  - **delVehicle(user, plate)**:it takes as input the driver identification and the number plate of the vehicle to be deleted. It returns an error or a success message with respect to the operation's outcome.

- **DriverServiceInterface** is accessible from ClientApp after obtaining the authorization to access to the system's functionalities, and it exposes the operations available for a driver.
  - **requestStationView(position)**: it takes as input the current position of the driver, and it returns the list of the nearby charging stations with their external status.
  - **requestBookingView(driver, station)**: it takes as input the identification of the driver and the chosen charging station, and it returns the list of available booking options wrapped in a form.
  - **sendBookingData(form)**: it takes as input the information filled in the charge booking form, and it returns a success or error message.
  - **requestStartCharge(reservation)**: it takes as input the identification of the reservation made by the driver, and it returns a success or error message.
  - **requestEndCharge(reservation)**: it takes as input the identification of the reservation made by the driver, and it returns the payment page if the charge successfully stops.
  - **sendPaymentData(data)**: it takes as input the payment related data inserted by the driver, and it returns a success or error message.
  - **validateInput(form)**: it takes as input the form data and validates its format. It returns False if the data has wrong format, and True if it is correctly formatted.

- **GeolocationInterface** is accessible from DriverService, and it handles the functionality of visualizing the nearby charging stations along with their external status.

  - **findStations(position)**: it takes as input the current position of the driver, and it returns a list of charging stations with their external status.

- **BookingInterface** is accessible from DriverService, and it offers the functionalities related to the charge booking.
  - **checkStatus(driver, station)**: it is involved in the preparation of the booking form for the driver. The input consists of the driver and the chosen station, while the return object includes the external status of the chosen charging station and the list of vehicles associated to the driver's account.
  - **bookCharge(formData)**: its input is the data contained in the booking form, and it returns a success or an error message.
  - **checkBooking(reservation)**: its input is the ongoing reservation's identifier, and it returns a success or an error message.
  - **checkCharge(reservation)**: its input is the ongoing reservation's identifier, and it returns the operation summary with the value of its cost.
  - **closeBooking(reservation)**: its input is the ongoing reservation's identifier, and it returns a success or an error message.

- **SuggestionInterface** offers the possibility to create charging suggestions for drivers. It is accessible from DriverService.
  - **computeSuggestion(calendar, path, availabilityList)**: it computes the best suggestions according to the parameters, which are the driver's calendar schedule, the navigation path, and the list of availabilities of the charging stations. It returns a list containing suggestions.

- **PaymentInterface** is accessible from DriverService.
  - **handlePayment(data)**: it takes as input the payment data inserted by the driver, and it returns a success or error message.

- **OperatorServiceInterface** is accessible from ClientApp after obtaining the authorization to access to the system's functionalities, and it exposes the operations that an operator is allowed to perform.
  - **requestStationStatus(station)**: it takes as the input the station to consider, and it returns an object encapsulating the external and internal status of the selected station.

- **sendOfferData(form)**: it takes as input the data contained in the offer creation form. It returns a success message if the operation is completed, otherwise an error message is displayed.
- **validateInput(form)**: it checks the validity of the data contained in the input form. The return value is True if it does not find errors, otherwise it is False.
- **requestSupplyView()**: it returns the list of energy costs published by DSOs.
- **sendAcquireData(form)**: it takes as input the data contained in the energy acquisition form, and it returns a success or an error message.
- **sendCostData(form)**: its input is the data contained in the form for the modification of the charge cost, and it returns a success or an error message.
- **requestEnergyAlternatives(station)**: it takes as input the station for which the operator wants to modify the energy source used for the charging operations, and it returns the list of available energy sources for that charging station.
- **sendEnergyOption(form)**: its input is the chosen option to modify the energy source used for the charges, wrapped in a formData format along with the station's identifier. At the end of the operation, a success or an error message is returned.

- **ChargeInterface** handles the charging process at the station according to the driver's reservation, and it offers the possibility to the eMSP to access to its functionalities from BookingManager.
  - **startCharge(reservation)**: it takes as input the reservation made by the driver, and it returns a success or an error message.
  - **charge(socket, mode)**: it takes as input the socket assigned for the charge and the charging mode, and it returns True if the operation correctly begins, otherwise False.
  - **stopCharge(socket)**: it takes as input the socket assigned for the charge. The return value is True if the power supply correctly stops, False otherwise.
  - **unlockVehicle(reservation)**: it takes as input the reservation made by the driver, and it returns a success or an error message.

  - **unlock(socket)**: it takes as input the socket where the vehicle is plugged in. It returns True if the vehicle is successfully unlocked, False otherwise.

- **StationInterface** is accessible from OperatorService.
  - **checkStationStatus(station)**: the input is the charging station the operator wants to check the status, and it outputs the status information of the station.
  - **createOffer(formData)**: its input consists in the data necessary for the creation of a new offer. It returns a success message or an error message.
  - **changeSource(formData)**: its input is the data necessary for the modification of the energy source used for the charges. At the end of the operation, it returns a success or an error message.
  - **modifyCost(formData)**:

- **EnergyInterface** is accessible from OperatorService.
  - **getBatteryStatus(id)**: its input is the battery's identification number, and it returns the value of the battery level.
  - **checkEnergyOffers()**: it returns a list of energy costs published by DSOs.
  - **acquireEnergy(quantity, DSO)**: it takes as input the quantity of energy and the DSO provider from which to acquire energy, and it returns a success message or an error message.

- **DataManagerInterface** is accessible from the components which requires the retrieval and the storage of information in the database, and its methods are meant to send queries and insertions to the DBMS.
  - **getDriver(email)**: it finds the driver associated to the input email address.
  - **registerDriver(formData)**: it registers a new account for the driver, with the information given by the formData. It returns a success or an error message.
  - **checkCredentials(email, pwd)**: it checks the validity of the email address - password combination in the database. The return value is True if the combination is correct, otherwise False.

- **registerVehicle(user, plate)**: it associates the vehicle identified by the number plate to the driver's account. It returns a success or an error message.
- **getNearbyStations(position)**: it returns the list of stations within a certain range of the given position.
- **getExternalStatus(station)**: it returns the external status information of the chosen station.
- **getAvailableTimeSlots(station)**: it returns a list containing the available time slots of the chosen station.
- **checkBookingAvailability(formData)**: it checks the possibility of doing the reservation with respect to the input information.
- **saveBooking(formData)**: it stores the reservation in the database with respect to the input information.
- **getVehicleList(driver)**: it returns the list of vehicles associated to the given driver.
- **getBooking(reservation)**: it retrieves the reservation according to the given information.
- **setBookingStatusOn(reservation)**: it sets the input reservation's status to ON (vehicle charging), and it returns True if the operation is successful.
- **setBookingStatusClosed(reservation)**: it sets the input reservation's status to CLOSED (reservation ended and paid). It returns True if the operation is successful.
- **getStationStatus(station)**: it returns the overall status information of the chosen station.
- **setOffer(formData)**: it registers a new offer with respect to the input information. It returns a success or error message.
- **getDSOList()**: it returns the list of the available DSOs.
- **setSource(formData)**: it modifies the energy source option for the selected charging station. It returns a success or error message.
- **setCost(formData)**: it modifies the energy cost for the selected charging station. It returns a success or error message.

- **NotificationInterface**: it is accessible from SuggestionManager and ChargeManager, and it aims to deliver notification to the user's mobile device.

  - **sendSuggestion(suggestion)**: it sends the input suggestion to the destination user interface.
  - **sendNotification(notification)**: it sends the input notification to the destination user interface.

- **ClientAppInterface**: it is accessible from NotificationManager with the objective to show the notification to the user.
  - **notify(message)**: it takes as input the message that the system wants to send to the user's device.

From the other hand, the interfaces of the external components and sensors are:

- **MapsInterface**: it provides localization and map functionalities to ClientApp.
  - **getCurrentPosition(user)**: it returns the current position of the user's device.
  - **prepareMap(position, stationList)**: it returns the map view, on which the given position and the stations' location are highlighted.
- **SocketSensorInterface**: it is accessed by ChargeManager and it offers the possibility to check the status of the vehicle connected to the socket.
  - **checkVehicle(plate)**: it verifies the identity of the connected vehicle by checking its plate number.
  - **checkVehicleBattery()**: it returns the battery level of the connected vehicle.
- **BankInterface**: it is accessed by PaymentManager.
  - **pay(data)**: it takes as input the payment data of the driver. After executing the operation, it returns an error message or a success message.
- **BatterySensorInterface**: it is accessible from EnergyManager.
  - **getBatteryLevel(id)**: it returns the power level of the battery indicated by the input identification number.
- **DSOPublisher**: it is accessible from EnergyManager.
  - **getEnergyPrice(DSO)**: it returns the energy cost published by the chosen DSO.

  - **acquire(quantity, DSO)**: it takes as input the quantity of energy to acquire, and the DSO from which the operator wants to acquire energy. It returns a success message or an error message.

- **ScheduleInterface**: it is accessible from SuggestionManager, and it offers the possibility to check the driver's schedule.
  - (VehicleSensors) **getBatteryLevel(vehicle)**: it returns the battery level of the input vehicle.
  - (CalendarService) **checkCalendar(user)**: it returns the list of today's user activities registered in the calendar.
  - (NavigationService) **checkNavigationSystem(user)**: it returns the user's path indicated in the navigation system.

2.6 Selected architectural styles and patterns

2.6.1 Three-tier architecture

As described in Section 2.1, we adopt the three-tier architecture for the eMall system for keeping apart the application logic from the data layer. In particular, the presentation tier represents a thin client model because it does not contain heavy operations which requires complex logic, it only handles the tasks for the visualization of different views.
On the other hand, in order to realize the separation of concerns, the middle-tier Application Server contains all the application logic, and it handles the operations by interacting with the presentation tier and the data tier.

2.7 Other design decisions

2.7.1 Relational Database

In the e-Mall system, all the persistent data is stored in the database. For data management, we make use of a Relational DBMS. It guarantees the ACID properties of transactions, which are important for the system. In fact, multiple users can access the database to retrieve information at the same time and even if data is being updated. It ensures data integrity by verifying that the data necessary for creating the relationships are actually present, and it also assures data accuracy, avoiding duplicates so that data are defined and well organized. Lastly, data is secure as Relational Database Management System allows only authorized users to directly access the data. No unauthorized user can access the information.

2.7.2 Stateless web service

The choice of Java EE for the implementation of the backend allows to use JAX-RX for creating REST web services, which is stateless: the server's response do not rely on the previous requests sent by the client. A stateless application is more scalable, and it requires less server-side storage.

3 User Interface Design

3.1 Mockups

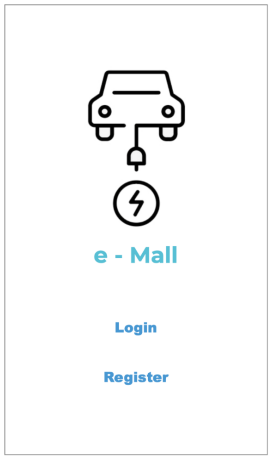

Figure 19: Login page

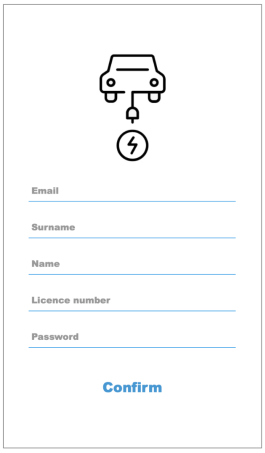

Figure 20: Registration Form

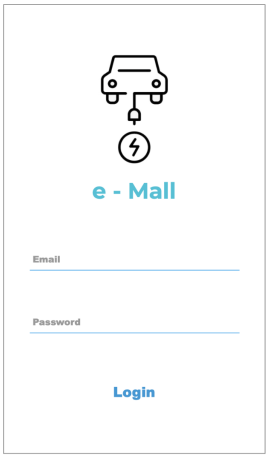

Figure 21: Login Form

The Login Page (Figure.19) and the Login Form (Figure.21) are shared for both type of users (Driver and Operator), while the Registration Form (Figure.20) allows a driver to register an account without privileged functionalities.

3.1.1 Driver's UI

Figure 22: Map Page

This represents the Main Page for the driver, as he is redirected here after the login operation. There is a map showing the position of the Driver and the nearby charging stations, which are represented with their icon (leaves in this case), where the Driver can select one of those to see further information. The search icon above is used for searching nearby charging stations according to the inserted address.

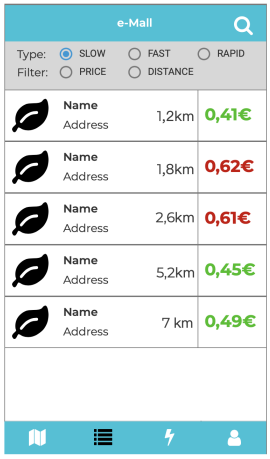

Figure 23: List Page

There is a list of charging stations where the user can filter selecting the buttons on top of the application. Driver can select one of those to see further information.

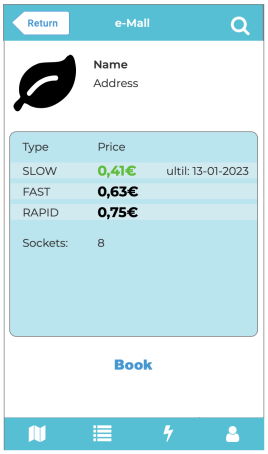

Figure 24: Charging Station Info

The page is showing information on the charging station. In this page Driver can book a charge.

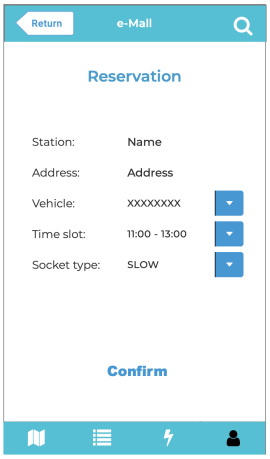

Figure 25: Reservation Page

In this page, Driver can compile the forum and click on confirm button to complete the booking process.

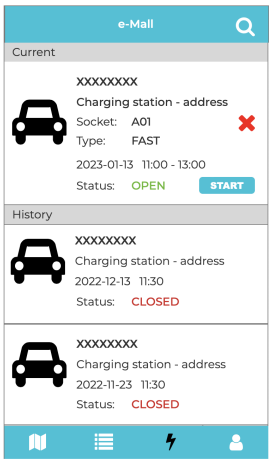

Figure 26: List of reservations

There is the list containing the current and past reservations. Under "current" it appears the last reservation that hasn't completed yet, where the user is able to cancel it or start charging his vehicle when he arrived at the charging station.

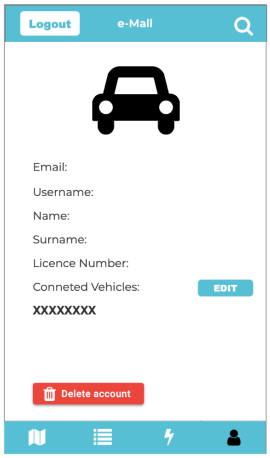

Figure 27: Account page

Here, all the Driver's profile information is reported. He can edit the list of vehicles he is connected to, and he can choose to delete his own account.

3.1.2 Operator's UI

Figure 28: List of Charging Stations

This is the Main Page for the logged Operator. It shows the list of Charging stations that the current Operator can manage.

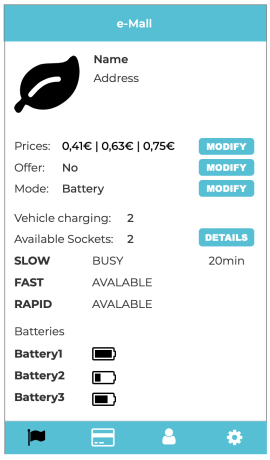

Figure 29: Charging Station Info

Information on the selected charging station. The Operator can edit some fields and see more details about them.

Figure 30: Sockets Info

In this page, there is more information on the charging station: details on each socket such as its status, price, charging vehicle, power absorbed by vehicle and the time left to complete the charging process.

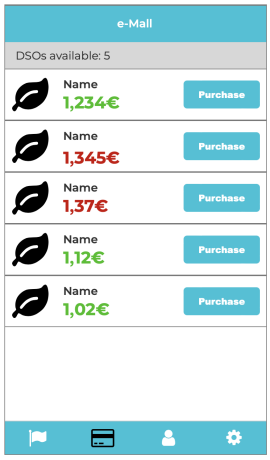

Figure 31: List of DSOs

List of purchasable DSOs' energy, where Operator can acquire energy from the available DSOs.

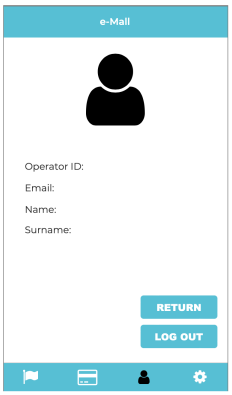

Figure 32: Account page

This is the profile of the Operator. The "Return" button lets him to return to the page where he can select the available charging stations (Figure.29).

4 Requirements Traceability

In this section, the mapping between the requirements specified in the RASD and the components of the eMall system is shown. To all the components abbreviations are given:

- **CA** - ClientApp
- **AS** - AccessService
- **UM** - UserManager
- **DS** - DriverService
- **GM** - GeolocationManager
- **BM** - BookingManager
- **PM** - PaymentManager
- **SuM** - SuggestionManager
- **OS** - OperatorService
- **SM** - StationManager
- **CM** - ChargeManager
- **EM** - EnergyManager
- **NM** - NotificationManager
- **DM** - DataManager
- **DB** - DBMS
- **MAP** - MapService
- **CalS** - CalendarService
- **NavS** - NavigationService
- **VeS** - VehicleSensors
- **BankS** - BankService

- **SocS** - SocketSensor
- **BattS** - BatterySensor
- **DSO** - DSOPublisher

Id	Description	Components
R1	The system shall allow an unregistered Driver to register and create an account	CA, AS, UM, DM, DB
R2	The system shall allow a registered Driver to log into his account	CA, AS, UM, DM, DB
R3	The system shall allow a registered Driver to delete his account	CA, AS, UM, DM, DB
R4	The system shall allow a registered Driver to add one or more vehicles to his account	CA, AS, UM, DM, DB
R5	The system shall allow a registered Driver to delete one or more vehicles from his account	CA, AS, UM, DM, DB
R6	The system shall allow a registered Driver to book a charging slot of a station for a certain time frame	CA, AS, UM, DS, BM, SM, DM, DB
R7	The system shall allow a registered Driver to acquire information of a selected charging station	CA, AS, UM, DS, GM, DM, DB
R8	The system shall allow a registered Driver to visualise the details of his charging process	CA, AS, UM, DS, BM, CM, DM, DB
R9	The system shall allow a registered Driver to visualize the current offers of charging stations	CA, AS, UM, DS, GM, DM, DB
R10	The system shall allow a registered Driver to pay for the obtained service through an external API	CA, AS, UM, DS, BM, CM, PM, DM, DB, BankS
R11	The system shall allow a registered Driver to cancel a reservation	CA, AS, UM, DS, BM, DM, DB
R12	The system shall not allow a registered Driver to book for a charge when he has already a reservation	CA, AS, UM, DS, BM, DM, DB
R13	The system shall allow a registered Driver to know about his actual position and nearby charging stations through an external API	CA, AS, UM, DS, GM, DM, DB, MAP
R14	The system shall allow an Operator working for the CPO to log into his account	CA, AS, UM, DM, DB

R15	The system shall allow a registered Operator to create an offer to the charging station managed by him	CA, AS, UM, OS, SM, DM, DB
R16	The system shall allow a registered Operator to modify the charging source of the charging station managed by him	CA, AS, UM, OS, SM, EM, DM, DB, DSO, BattS
R17	The system shall allow a registered Operator to modify the price of the energy of a charging station managed by him	CA, AS, UM, OS, SM, DM, DB
R18	The system shall allow a registered Operator to check the status of the charging station managed by him	CA, AS, UM, OS, SM, EM, DM, DB, BattS
R19	The system shall allow a registered Operator to visualise data provided by DSOs	CA, AS, UM, OS, SM, EM, DM, DB, DSO
R20	The system shall allow a registered Operator to choose from which DSO to acquire energy	CA, AS, UM, OS, SM, EM, DM, DB, DSO, BattS
R21	The system must be able to decide automatically where to get energy for charging	SM, EM, DM, DB, DSO, BattS
R22	The system must be able to notify the Driver when the charging process starts	CA, AS, UM, DS, BM, CM, DM, DB, SocS
R23	The system must be able to notify the Driver when the charging process ends	CA, AS, UM, CM, NM
R24	The system must be able to suggest the Driver to charge the vehicle, the suggestion is based on information retrieved through external APIs	CA, AS, UM, DS, SuM, NM, DM, DB, CalS, NavS, VeS
R25	The system must be able to notify user in case of exceptions	CA, AS, UM, DS, GM, BM, PM, OS, SM, CM, EM, DM, DB
R26	The system must be able to notify user on successful actions	CA, AS, UM, DS, GM, BM, PM, OS, SM, CM, EM, DM, DB
R27	The system must be able to retrieve a map through an external API	CA, AS, UM, MAP
R28	The system shall allow a registered Driver to start his charge	CA, AS, UM, DS, BM, CM, DM, DB, SocS
R29	The system shall allow a registered Driver to finish his charge	CA, AS, UM, DS, BM, CM, PM, DM, DB, BankS

	CA	AS	UM	DS	GM	BM	PM	SuM	OS	SM	CM	EM	NM	DM	DB	MAP	CalS	NavS	VeS	BankS	SocS	BattS	DSO
R1	X	X	X											X	X								
R2	X	X	X											X	X								
R3	X	X	X											X	X								
R4	X	X	X											X	X								
R5	X	X	X											X	X								
R6	X	X	X	X		X				X				X	X								
R7	X	X	X	X	X									X	X								
R8	X	X	X	X		X					X			X	X								
R9	X	X	X	X	X									X	X								
R10	X	X	X	X		X	X				X			X	X					X			
R11	X	X	X	X		X								X	X								
R12	X	X	X	X		X								X	X								
R13	X	X	X	X	X									X	X	X							
R14	X	X	X											X	X								
R15	X	X	X						X	X				X	X								
R16	X	X	X						X	X		X		X	X							X	X
R17	X	X	X						X	X				X	X								
R18	X	X	X						X	X		X		X	X							X	
R19	X	X	X						X	X		X		X	X								X
R20	X	X	X						X	X		X		X	X							X	X
R21										X		X		X	X							X	X
R22	X	X	X	X		X					X			X	X						X		
R23	X	X	X								X		X										
R24	X	X	X	X				X					X	X	X		X	X	X				
R25	X	X	X	X	X	X	X		X	X	X	X		X	X								
R26	X	X	X	X	X	X	X		X	X	X	X		X	X								
R27	X	X	X													X							
R28	X	X	X	X		X					X			X	X						X		
R29	X	X	X	X		X	X				X			X	X					X			

Figure 33: Table of Requirements Traceability

5 Implementation, Integration and Test Plan

This section will be describe how the various components of the system will be implemented, integrated together and tested. The implementation, integration and test plan will follow a bottom-up approach, applied to the entire system, including both Server and Client side.

Assuming that the external services are reliable, it is not required to do unit testing on them.

5.1 Development Process

Integration and testing will be done incrementally since it facilitates bug tracking. Each component should be checked under unit testing, and this is done before, during and after integration of a new module into the main software package. This involves testing of each individual code module.

The whole system will be implemented, tested and integrated mainly using a bottom-up approach. The latest emphasizes coding and early testing, which can begin as soon as the first module has been specified. In this way, the system could be completed incrementally and could be tested in parallel to its implementation. The available elements are used in the construction of new and more powerful elements at every stage.

The dependencies between components are described in the diagram below:

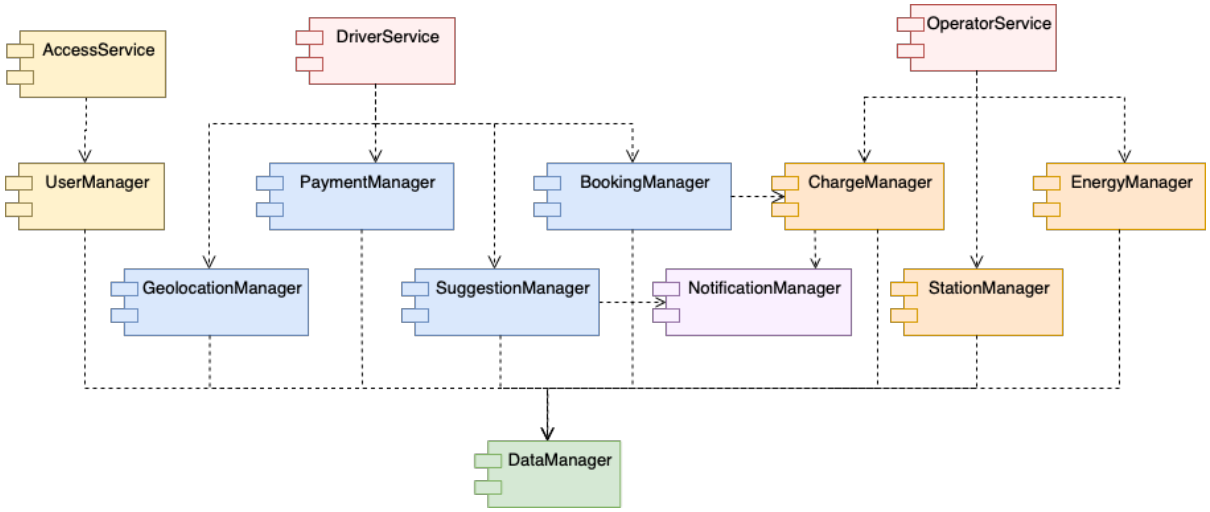

Figure 34: Dependency Diagram

5.2 Implementation Plan

The implementation of the components of the Application Server is the follow:

1. **DataManager,NotificationManager**
   The first component that need to be implemented will be DataManager. We can clearly see on the component diagram that a lot of components of the server side relies on it for the communication with Database. Since NotificationManager does not depend on any component, it can be implemented in parallel with DataManager.

2. **GeolocationManager, SuggestionManager, PaymentManager, BookingManager, ChargeManager, StationManager, EnergyManager, UserManager**
   These are the components of eMSP and others of the CPMS. Most of them can be implemented in parallel since they are independent to each other. They are dependent just on DataManager, which has been implemented before, and/or External APIs, which are ready to be used.
   The only problem we have is that BookingManager which is dependent on ChargeManager. This can be solved by implementing the main functionalities in ChargeManager which are required by BookingManager, so that the implementation of the latest can be started as soon as possible.

3. **AccessService, DriverService, OperatorService**
   These components' functionalities heavily depend on the previous mentioned components, so that they must wait for those to be completed. However, these components are independent to each other so that they can be implemented in parallel.

5.3 Integration Sequence

In this section, the integration plan of the e-Mall system will be explained. Each component after its development will go under Unit Testing. After that integration testing process will start soon. The model will be integrated into the models of low level in order to test the behaviour of developed subsystem, in order to solve as many inconsistencies as possible.

In the diagram below, it's shown then integration process at various levels of dependency. Note that Bottom-up strategy is used for those components which do not interact with external systems, while top-down strategy is used for the components which need to acquire information from external systems.

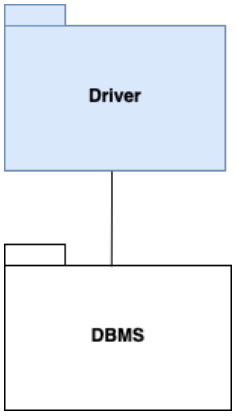

Figure 35: Initial state before integration

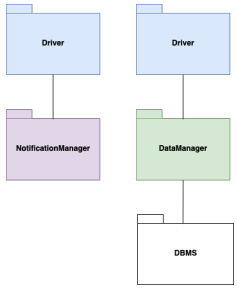

Figure 36: Integration of DataManager and NotificationManager

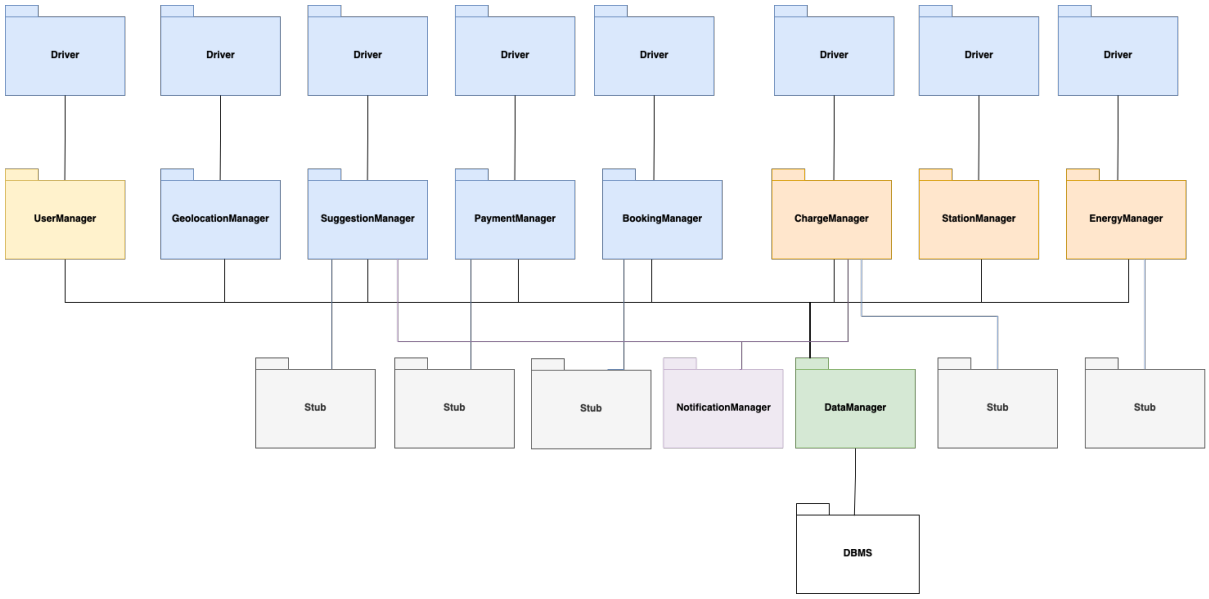

Figure 37: Integration of GeolocationManager / SuggestionManager / PaymentManager / BookingManager / StationManager / ChargeManager / EnergyManager / UserManager

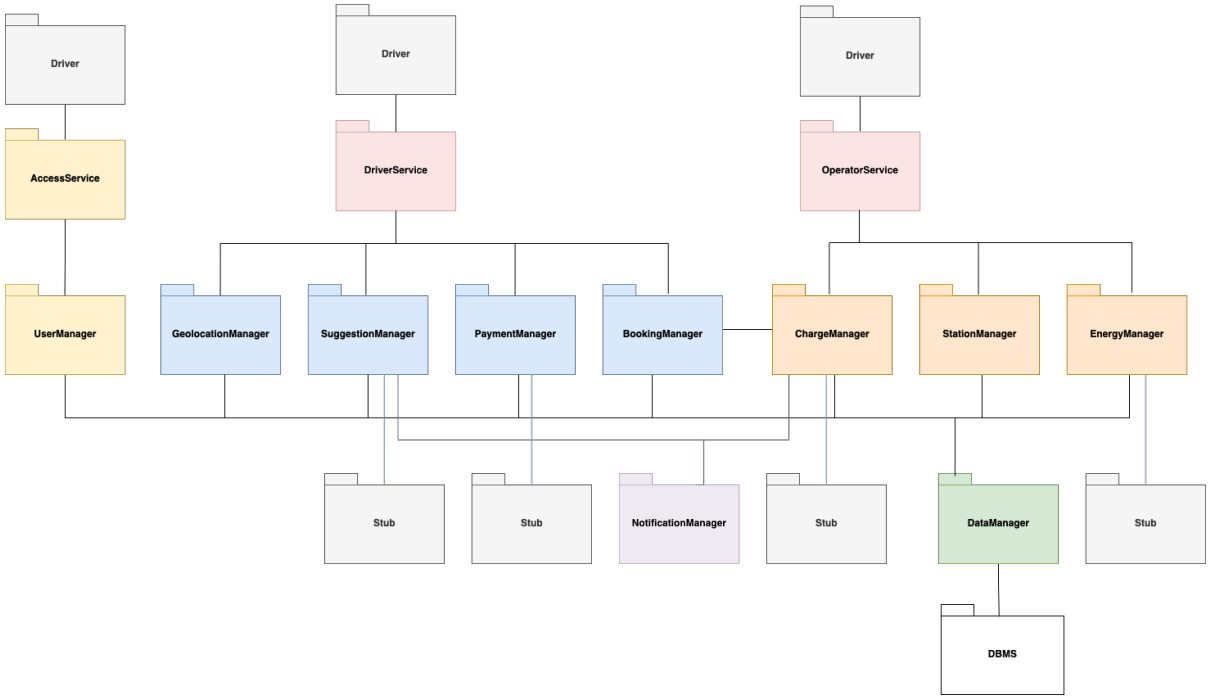

Figure 38: Integration of AccessService/ DriverService/ OperatorService

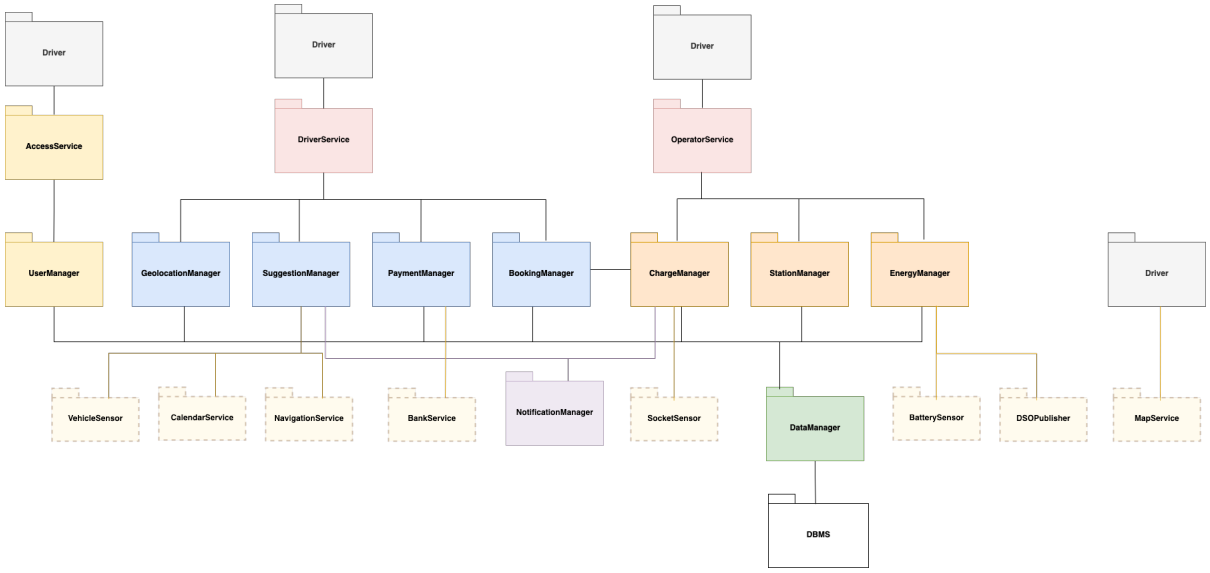

Figure 39: Integration of external services: CalendarService/ NavigationService/ VehicleSensor/ BankService/ SocketSensor/ BatterySensor/ DSOPublisher/ MapService

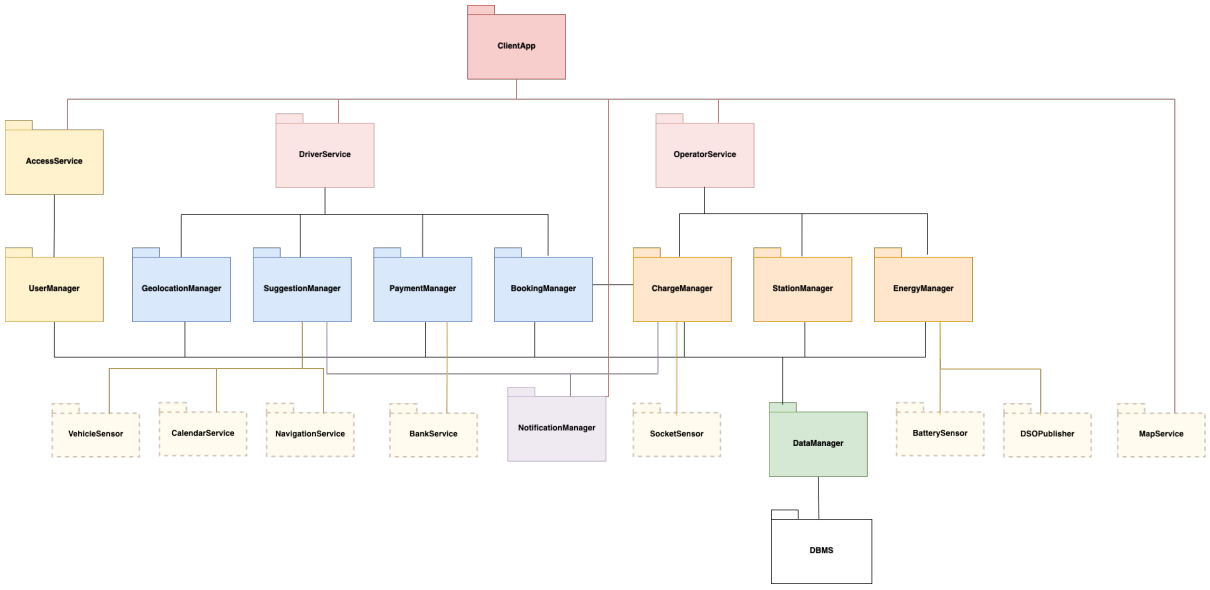

Figure 40: Integration of all components with ClientApp

5.4 System Testing

Once all system components have been integrated, it is time for System Testing. This process aims at verifying functional and non-functional requirements and must take place in a testing environment which is as close as possible to the production environment. Especially, the e-Mall System will be subject to the following tests:

- Functional testing:
  verifying if all requirements written on RASD are satisfied and if there will be any possible missing functions.

- Performance testing:
  necessary to check how the system is efficient in respect with response time, utilization, throughput by identifying bottlenecks. Establishing a performance baseline and compare between different versions of the same product or a different competitive product.

- Load testing:
  necessary to know how the system will perform under real-life loads. Useful for adjusting problems connected to memory such as memory leaks, mismanagement of memory, buffer overflows.

- Stress testing:
  useful to know if the system can recover gracefully after failure.

6 Effort Spent

- Giulia Huang

Working Task	Time spent
Group discussions	7h
Definition of system architecture	5h
Definition of database model	3h
Definition of component interfaces	2h
Realization of User Interfaces	10h
Review of the requirements and components mapping	1.5h
Implementation and integration planning	5h
Testing planning	1h
Final review and restructuring	3h
Total	37.5h

- Zheng Maria Yu

Working Task	Time spent
Group discussions	7h
Document organization	1.5h
Definition of system architecture	8h
Definition of database model	1h
Realization of Runtime Views	8h
Definition of component interfaces	6h
Review of the requirements and components mapping	1h
Implementation and integration planning	1h
Final review and restructuring	3h
Total	36.5h

- Linda Zhu

Working Task	Time spent
Group discussions	7h
Definition of system architecture	6h
Definition of database model	1h
Definition of Deployment View	4h
Realization of Runtime Views	4h
Realization of Requirements Traceability	5h
Review of the requirements and components mapping	1.5h
Definition of component interfaces	3h
Implementation and integration planning	1h
Final review and restructuring	3h
Total	35.5h

7 References

- Project specification: "Assignment R&DD A.Y. 2022-2023 v3"
- Software Engineering 2 Course slides A.Y. 2022-2023
- RASD V1.0 of the eMall system
- IEEE Guide for Developing System Requirements Specifications
- Java EE Technical Documentation
- JAX-RX API Specification