eMall - e-Mobility for All

Design Document

08/01/2023

Version 1

Authors:

Brugnano Matilde

Buttiglieri Giorgio Natale

Academic Year 2022/2023

Software Engineering 2 Project

Table of Contents

1 INTRODUCTION

1.1 Purpose

Nowadays electric mobility (e-mobility) is one of the greatest challenges of our planet.

All sectors of the contemporary industry actually want to be "green": public opinion is increasingly aware of the importance of climate change and the very high risks that this implies, both for present and future generations.

According to this objective, the electric vehicle market is constantly expanding, and, in parallel with this, more and more business opportunities were revealed in the electric vehicle charging-service companies, which see eMSPs and CPOs as the main characters.

eMSP, which stands for e-Mobility Service Providers, is a company offering an electric vehicle charging service to drivers by providing them access to multiple charging points around a geographic area.

CPO, which stands for Charge Point Operator, is a company managing a pool of charging points which creates value by providing smart charging features to eMSPs.

eMSPs and CPOs are two crucial and interdependent elements of a charging network. Together, they install and manage electric vehicle charging stations capable of supporting a fast, easy-to-use, largely automated charging system.

This document aims at providing the design specifications needed for the development of e-Mall (e-Mobility for All), a system made of both eMSP and CPMS needed to manage electric vehicle charging from the energy sources acquisition to the final customers service.

1.2 Acronyms

Term	Definition
eMSP	e-Mobility Service Providers: company offering a charging service to drivers of electric vehicles. Here, eMSP is intended as the software system used by the company to manage charging processes.
eMSP app	Mobile app through which drivers of electric vehicles can interact with the services offered by eMSP.
CPO	Charge Point Operator: company that owns and manages a network of charging stations.
CPMS	Charge Point Management System: software system that manages the charge point infrastructure of a CPO company through its connection with the charging equipment.
CPMS app	Web app through which CPO operators can interact with the services offered by CPMS.
DSO	Distribution System Operators: external company responsible for distributing and managing energy from the generation sources to charging stations.

1.3 Definitions

Term	Definition
Customer	End user of eMSP app, generally intended as the electric vehicle owner.
eMSP admin	eMSP employee responsible for configuring the CPOs network (so their CPMSs' one) with which their eMSP collaborates.
CPO admin	CPO employee responsible for registering CPO operators, DSO partnerships and charging stations in the CPMS app. They also need to configure the eMSP network with which their CPMS collaborates. All the operations managed by CPO operators can be performed by CPO admins as well.
CPO operator	CPO employee which can manage CPO operations through CPMS app. They lack the permissions due to the CPO admins.
CPO employee	CPO admin or operator.
Charging station	Infrastructure made up of one or more charging points for plug-in electric vehicles.
Charging point	Piece of equipment that supplies electrical power for plug-in electric vehicles made up of one or more charging sockets. We refer to charging points making a distinction on the basis of their "mode": "in advance" or "on the fly". Charging points whose mode is "in advance" can be reserved with bookings "in advance" only; in the same way, charging points whose mode is "on the fly" can be reserved with bookings "on the fly" only. Without losing any generality, we can both refer to the mode of sockets and the mode of charging point, assuming that all its sockets have the same mode as it.
Booking a charge in advance	Booking made when a customer wants to reserve themselves a spot in a certain charging station to recharge their vehicle in a selected future timeframe. Only charging points whose mode is "in advance" will be bookable by bookings "in advance".
Booking a charge on the fly	Booking made when a customer wants to recharge their vehicle at the same moment they reach a charging station. Only charging points whose mode is "on the fly" will be bookable by bookings "on the fly".
Booking	Booking made in advance or on the fly.
Booking code	Identifier code associated with a booking by CPMS.
Pricing policy	CPO's approach to determining the price set by the company in a charging station. Each created policy contains some parameters that have to be met when prices are automatically set by CPMS (e.g. minimum/maximum percentage of cost-plus pricing added onto DSO's prices).

Payment gateway	Merchant service provided by an e-commerce application service provider that authorises credit card or direct payments processing. Here, payment gateway is intended as the third party service that handles payments between CPMS company, eMSP company and customers.
Energy switcher	Third party service that physically handles the use of charging stations' batteries and the energy acquisition from DSOs given a specific mix of energy sources by CPMS. Every charging station is managed by an energy switcher, which is able to disconnect a battery from the grid, use it as a source of energy, store energy in it, or to switch on/off the energy acquisition from a specific DSO.
Map service	Third party service that provides the map view of a certain geographical area. eMSP exploits this service to show charging stations nearby the customer.
Mail service	Third party service that provides companies (here, CPMS and eMSP) with tools to send emails.

1.4 Abbreviations

Term	Definition
UC	Use Case
SQD	Sequence Diagram

1.5 Scope

The analysis carried out within this document focuses on the design of both eMSP and CPMS softwares. The two systems are thought of as separate, but their cooperation is needed to accomplish the commissioned goals, described in the RASD document.

In some of the following sections we will refer to a single CPMS partnered with a single eMSP, but the same considerations hold for the communication between an eMSP and multiple CPMSs compliant with the same standards. In particular, an eMSP can interact with multiple CPMSs each of which is owned by a different CPO.

1.6 Revision History

- 08/01/2023: Version 1

1.7 Document Structure

This document is composed of seven sections, detailed below.

The first section summarises the purpose of the project and provides all the acronyms, definitions and abbreviations used in this document. A more detailed description of the

application domain and the goals of the project can be found in the corresponding RASD document.

The second section describes the architectural design of the system, from the component view to the deployment view. Moreover, in this section you can find detailed sequence diagrams related to the most relevant use cases presented in the RASD document. Lastly, component interfaces and other design decisions are provided.

The third section supplies an overview of the user interfaces of the system by providing the main mock-ups of the mobile app designed for eMSP and the web app for CPMS.

The fourth section explains how the requirements defined in the RASD document map to the presented design elements.

The fifth section provides the schedule planned to implement, integrate and test all the subcomponents of the system.

The sixth section reports each project member's contribution to this document.

The seventh section includes the bibliography and sitography referred to in this document.

2 ARCHITECTURAL DESIGN

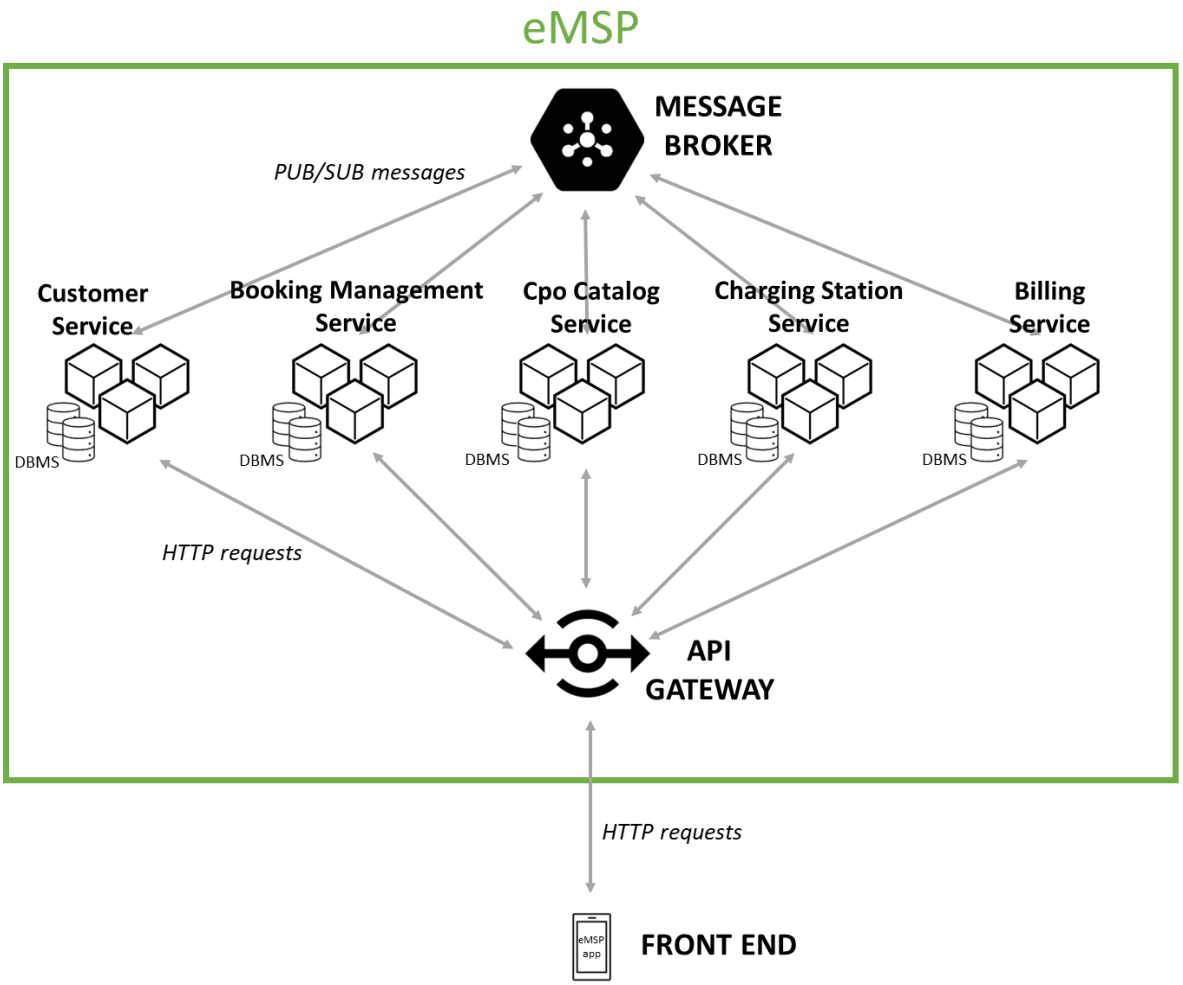
2.1 Overview: High-level components and their interaction

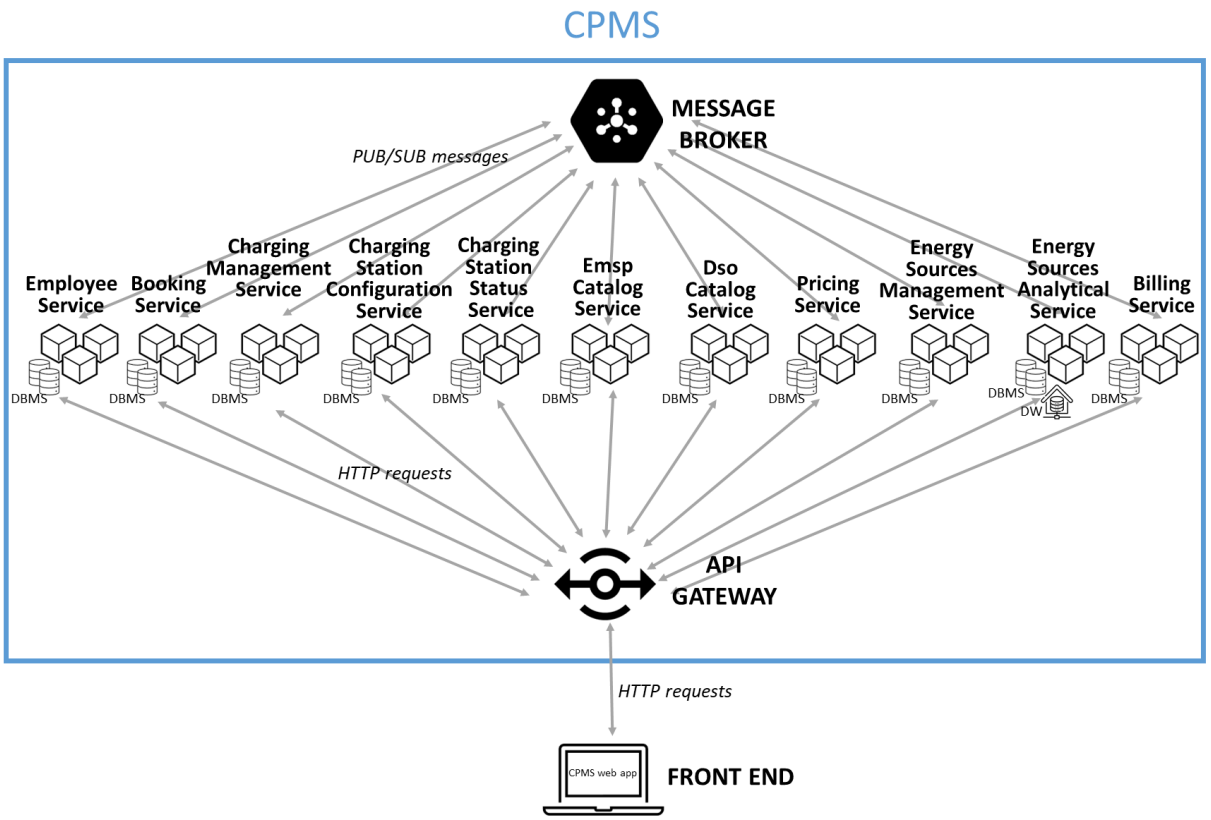
Electric vehicles are becoming more and more popular, and consequently there is a really high demand for electric charging services. The volatile and unpredictable electricity demand challenges businesses interested in delivering charging services. Businesses need to invest in efficient and scalable infrastructures that allow them to keep up with this kind of demand.

For this reason, we propose a microservice architecture for both the eMSP and the CPMS systems. Thanks to this infrastructure, all components of our systems can dynamically scale horizontally and deal with increasingly high computational requests.

Moreover, since microservices are loosely coupled, they can be scaled independently and therefore face the heterogeneity in computational demand of the different components of the systems.

Here is a high-level overview of the eMSP and CPMS systems:





As the diagram shows, both eMSP and CPMS are composed of microservices.

Each microservice is composed of stateless replicated instances of an application server and a data store. Microservices expose HTTP interfaces (following the REST style) and can communicate with one another also through a message broker.
An API Gateway unifies access to the systems and enforces cross-cutting concerns such as authentication and authorization.

2.2 Component view

2.2.1 API Gateway

An important component of both the eMSP and the CPMS systems is the API Gateway. It is responsible for providing a unified view of the system hiding its internal structure.

It is crucial for this component to be highly available and fault-tolerant because it might become a bottleneck since all traffic toward the system goes through it.

The API Gateway is also responsible for handling authentication and authorization.

Authentication is provided by the validation of a JWT token that needs to be present in all requests that require authentication.

Authorization is based on claims contained in the JWT token so that all the participants can access only the portion of the API they are authorized to.

Further authorization criteria are enforced in the specific microservices.

2.2.2 Message Broker

Another important component used both in the eMSP and in the CPMS is the message broker.

Our choice of message broker has been Apache Kafka.

Even though Apache Kafka is not only a message broker, different motivations have contributed to this choice:

- The message broker needs to support a huge volume of requests from different microservices. It needs to be highly available and keep into account the different processing speeds of each consumer. Kafka is highly available by design and also it is based on a pull model, which is optimal when consumers have different message processing speeds. It is not possible for a slow consumer to be overwhelmed by the rate of message delivery of the broker, since the speed of consumption is decided by the consumer itself.
- Microservices use the message broker also for data integration. We want to have guarantees on the persistence of messages published on the brokers. Once again Kafka by design provides different persistence guarantees: messages can be stored in Kafka topics for a specified amount of time or forever.

2.2.3 Microservices

Even though each microservice has been designed to solve the specific problem it is responsible for, the structure of the application service of all the microservices is consistent.

These are the main common components:

- Controllers:

  Responsible for exposing the HTTP API of the microservice. They handle basic validation of the input, enforcing the compliance of the request with the defined API schemas. They also invoke the related use cases.

- Use Cases:

  Implementation of the use case that the microservice participates in.
  The Use Cases rely on Managers to interact with the aggregates.

- Managers:

  Components responsible for managing the aggregates. They enforce the inter-entity invariants and provide the API to change the state of an aggregate. They rely on repositories to retrieve the interested aggregates and perform queries.

- Repositories:

  Components responsible for abstracting the data access. From the point of view of the clients (Managers) they are a collection of entities that provide insertion, removal and query primitives over the tracked entities. They interact with the data store.

- Model:

  The core of the microservice. It is composed of the aggregates that describe the domain of interest.

In the paragraphs below we describe each microservice by providing a description and the relative component diagram. For each microservice it is also specified the type of the selected data store.

In each component diagram we show the inner components responsible for implementing the business processes associated with the microservice.

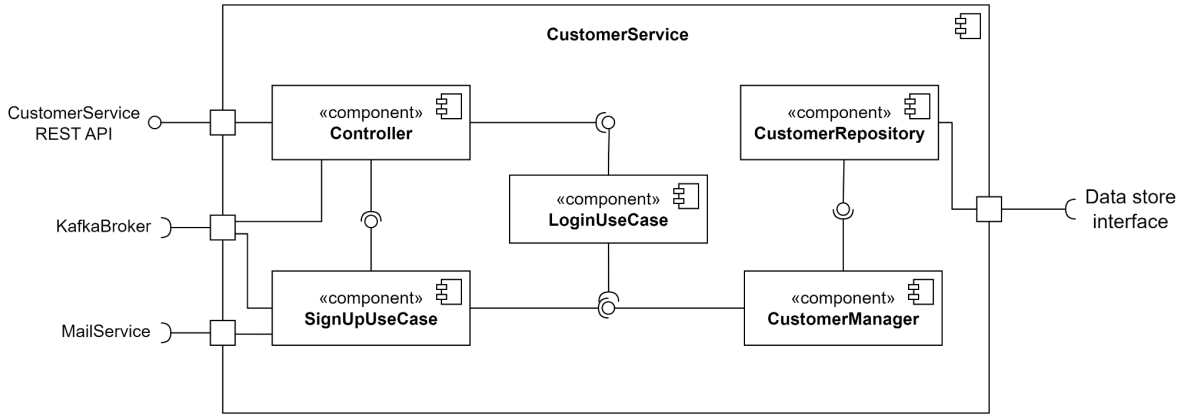
When use cases are mentioned, the associated code in the RASD document is provided between square brackets.

eMSP

CUSTOMER SERVICE

This microservice is responsible for handling the creation and management of customers' personal data. It assumes an important role in the "User signs up for the eMSP" [UC.1] and "Customer logs into the eMSP" [UC.3] use cases.

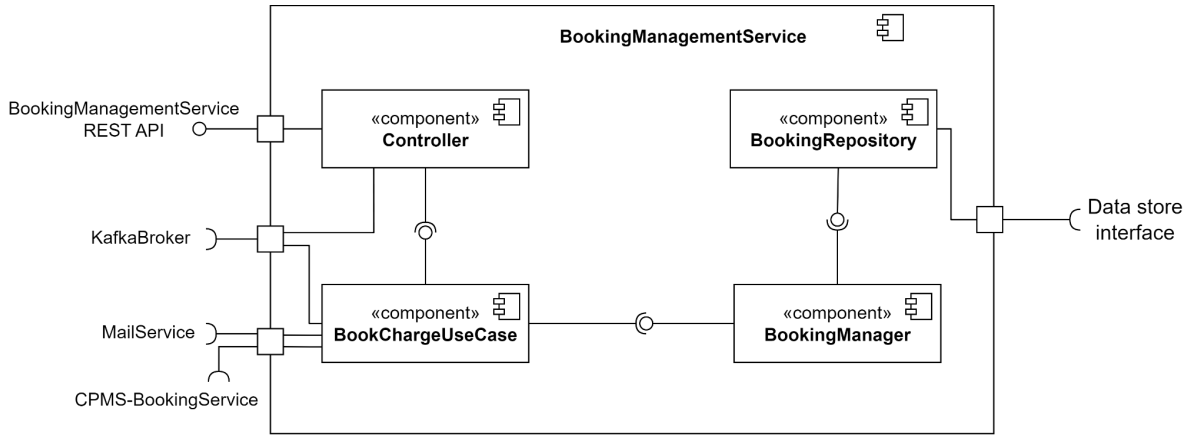
This microservice handles simple data with a stable format. We chose a relational database as the data store.



BOOKING MANAGEMENT SERVICE

This microservice is responsible for handling the creation and management of the customers' bookings, allowing customers to start or delete the planned bookings. It assumes an important role in the "Customer books a charge" [UC.5], "Customer starts charging their vehicle" [UC.7] and "The delivery of energy to a vehicle stops" [UC.8, UC.9, UC.10, UC.11] use cases.

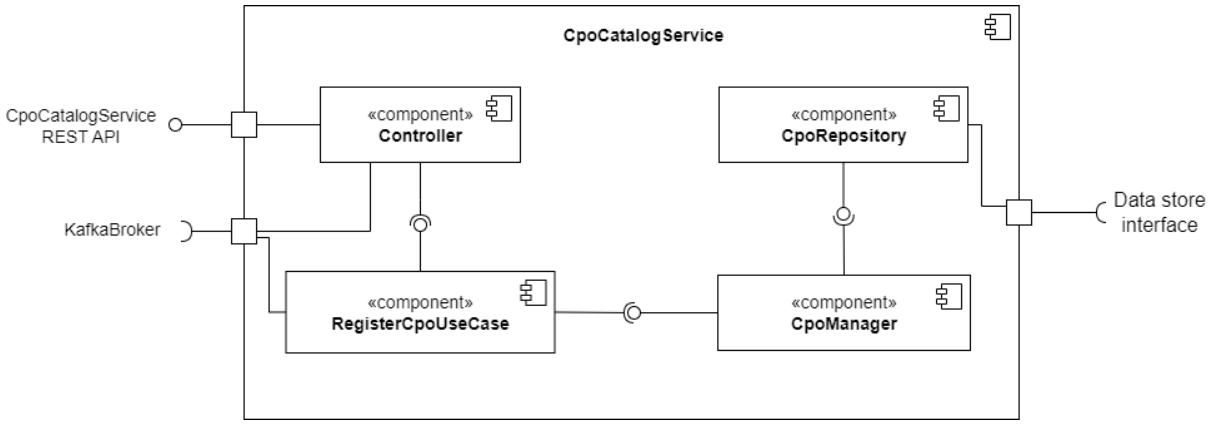
Since the change in the state of a booking is a critical process, we want to have consistency guarantees. We chose a relational database as the data store.



CPO CATALOG SERVICE

The microservice is responsible for handling the creation and management of the information about the CPOs that the eMSP is partnered with. For each CPO, this microservice is responsible to store all information needed to communicate with the relative CPMS.

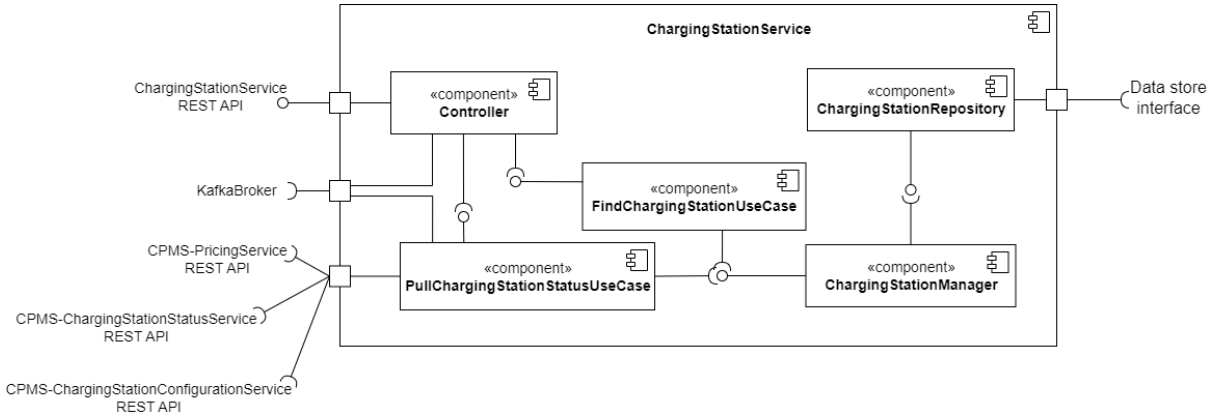
Changes in the data owned by this microservice are sporadic and simple, so we don't need strong consistency guarantees. We chose a document-based database as the data store.



CHARGING STATION SERVICE

The microservice is responsible for keeping updated the information (location, current price, current offers) about the charging stations of the partnered CPOs. It assumes an important role in the "Customer finds a charging station" [UC.4] use case.

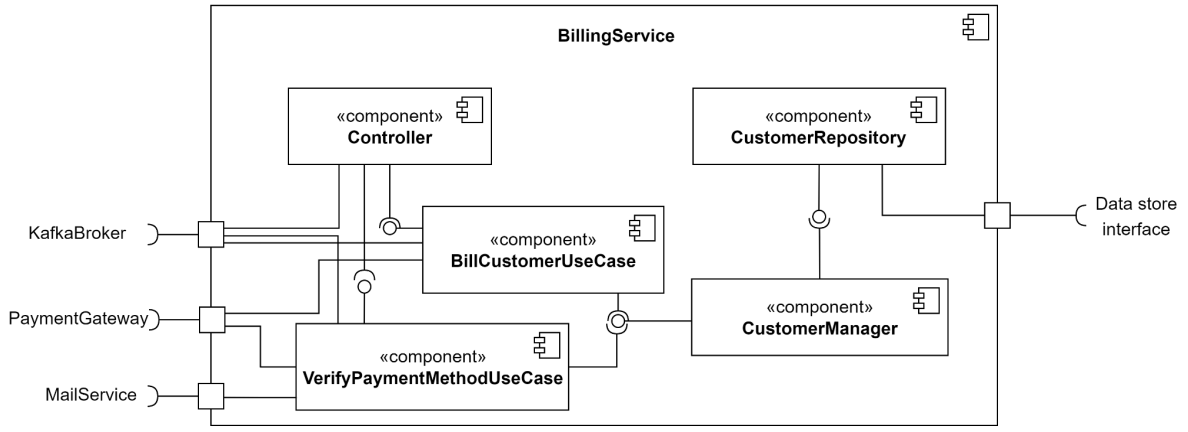
The data managed by this microservice is highly complex and it is important to run queries on big volumes of entries. We chose a relational database as the data store.



BILLING SERVICE

The microservice is responsible for billing the customer based on their bookings. It assumes an important role in the “eMSP bills a customer” [UC.6] use case.

The business processes performed by this microservice are crucial and need strong transactional consistency. We chose a relational database as the data store.

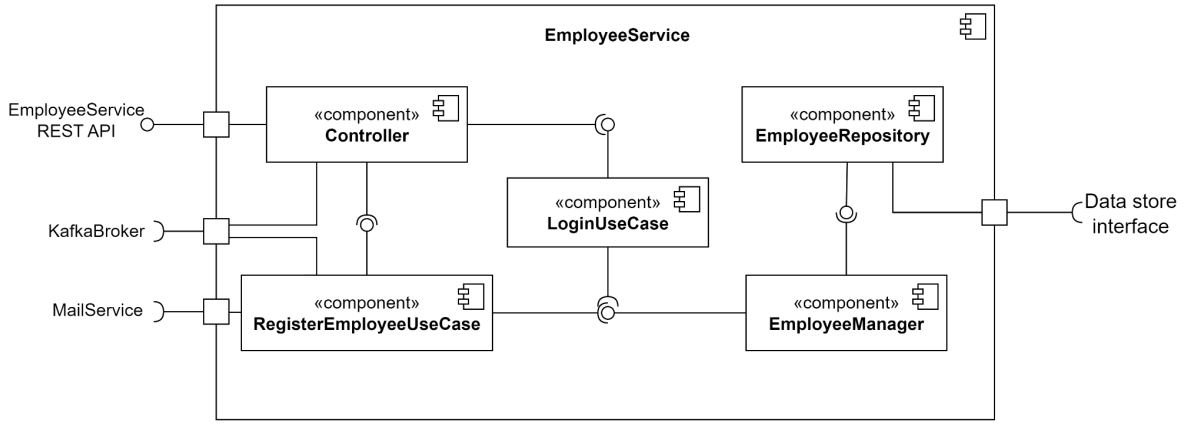


CPMS

EMPLOYEE SERVICE

This microservice is responsible for handling the creation and management of CPO employees’ data. It assumes an important role in the “CPO Admin registers a CPO Operator” [UC.2] use case and allows employees to log in.

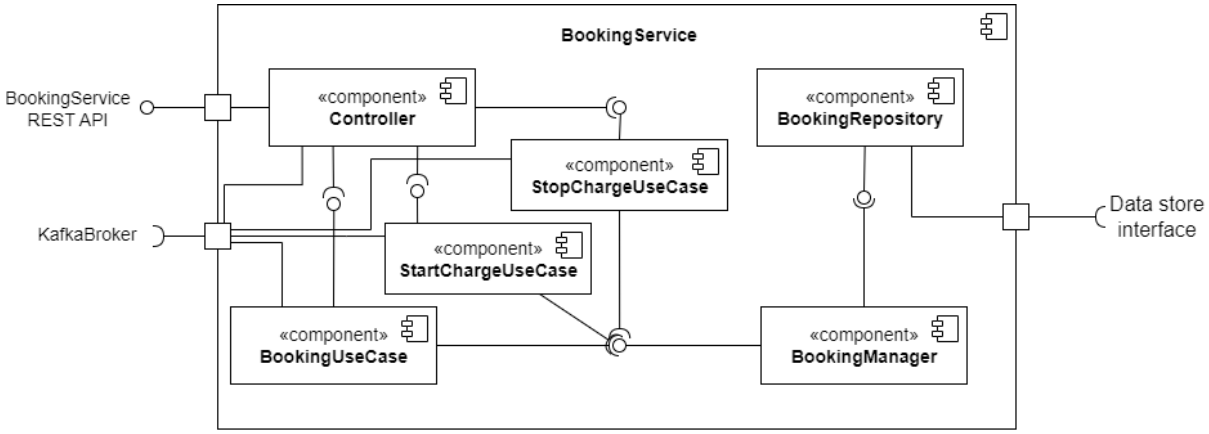
This microservice handles simple data with a stable format. We chose a relational database as the data store.



BOOKING SERVICE

This microservice is responsible for handling the creation and management of the booking requested by customers. It assumes an important role in the “Customer books a charge” [UC.5], “Customer starts charging their vehicle” [UC.7] and “The delivery of energy to a vehicle stops” [UC.8, UC.9, UC.10, UC.11] use cases.

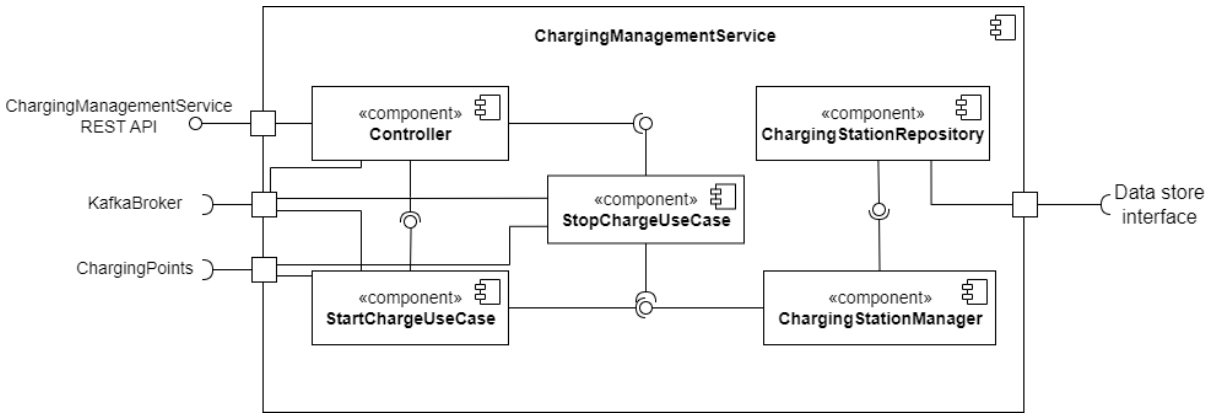
Since the change in the state of a booking is a critical process, we want to have consistency guarantees. We chose a relational database as the data store.



CHARGING MANAGEMENT SERVICE

This microservice is responsible for handling the charging process. It directly communicates with the charging points and coordinates them from the start of a charge until its completion. The microservice assumes an important role in the “Customer starts charging their vehicle” [UC.7] and “The delivery of energy to a vehicle stops” [UC.8, UC.9, UC.10, UC.11] use cases.

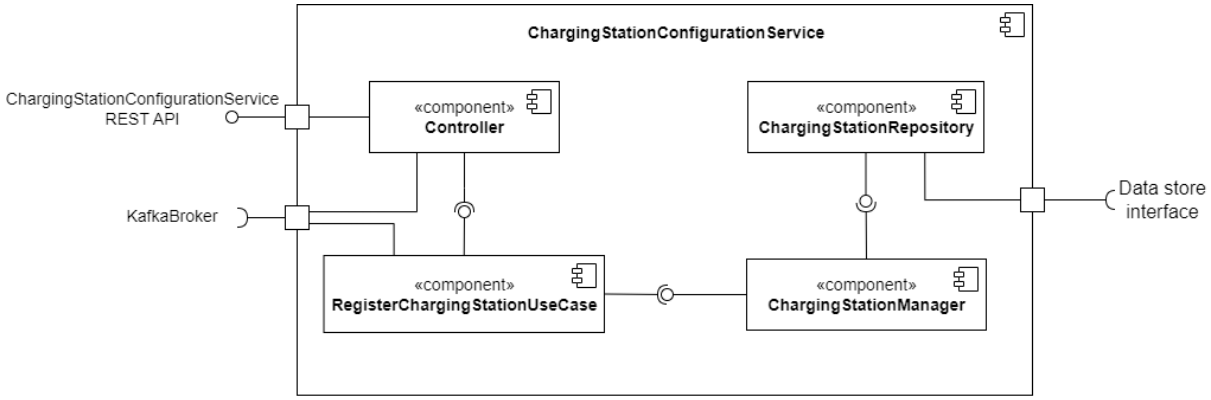
The processes handled by this use case need strong consistency. We chose a relational database as the data store.



CHARGING STATION CONFIGURATION SERVICE

This microservice is responsible for configuring the equipment of each charging station (charging points and their sockets, battery if present) along with information about the charging station itself (name, location).

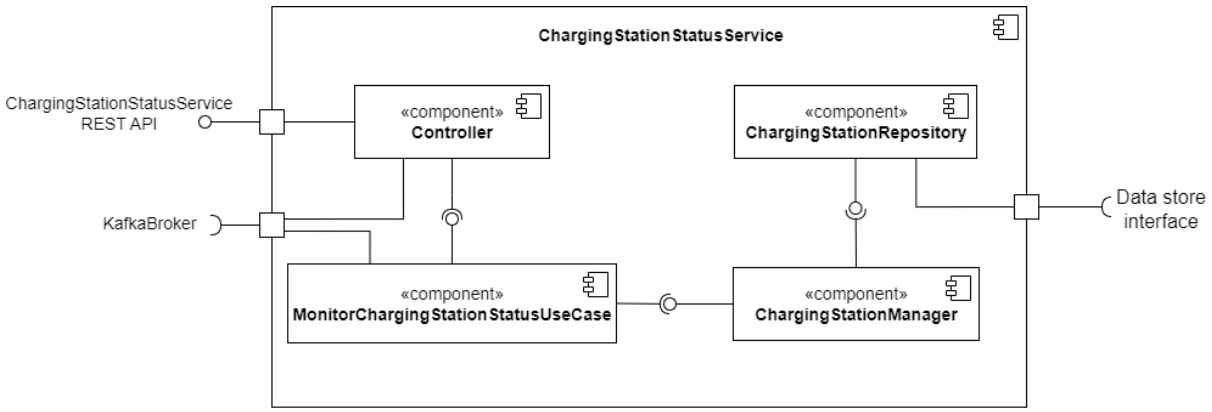
The data handled by this microservice is rarely changed and naturally grouped by charging station. There is no need for strong transactional consistency because updates are local to each charging station. We chose a document-based database as the data store.



CHARGING STATION STATUS SERVICE

This microservice is responsible for providing an overview of the status of the charging stations by consuming events published by the Charging Management Service on the broker.

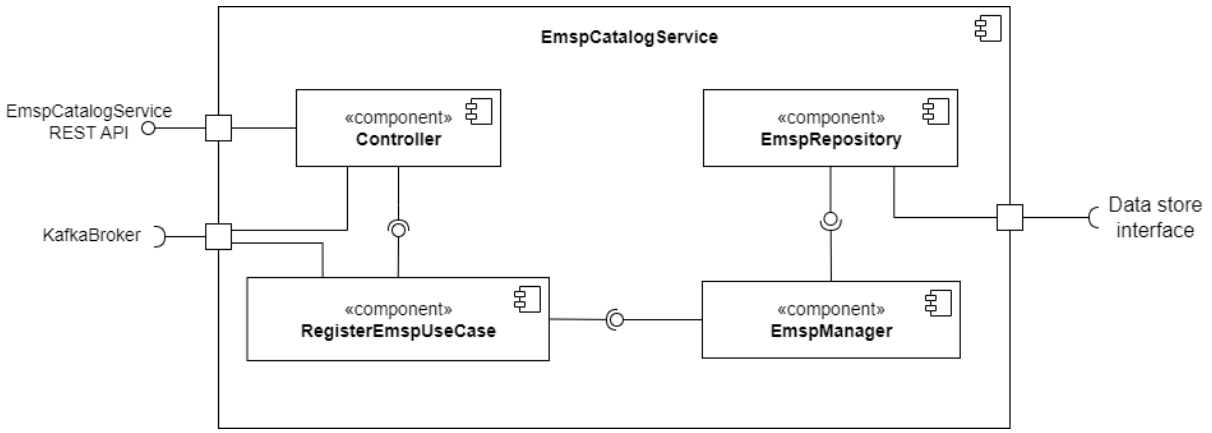
The data managed by this microservice needs to face complex queries. We chose a relational database as the data store.



EMSP CATALOG SERVICE

The microservice is responsible for handling the creation and management of the information about the eMSP that the CPO is partnered with. This microservices is responsible for storing all information needed to communicate with the associated eMSPs.

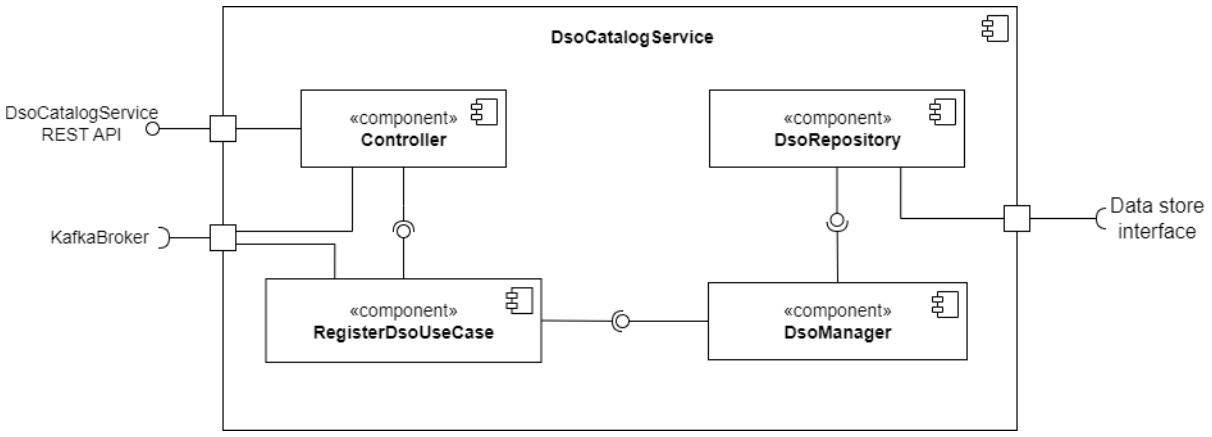
Changes in the data owned by this microservice are sporadic and simple, so we don't need strong consistency guarantees. We chose a document-based database as the data store.



DSO CATALOG SERVICE

The microservice is responsible for handling the creation and management of the information about the DSOs that the CPO is partnered with. This microservices is responsible for storing all information needed to communicate with the associated DSO.

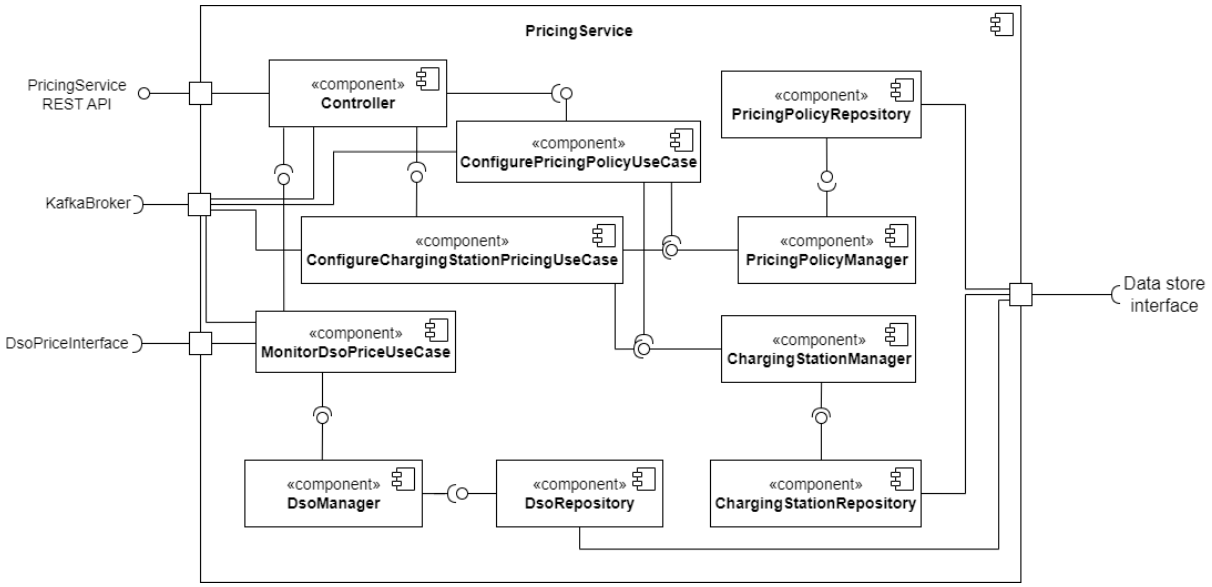
Changes in the data owned by this microservice are sporadic and simple, so we don't need strong consistency guarantees. We chose a document-based database as the data store.



PRICING SERVICE

The microservice is responsible for handling the automatic computation of the price of energy and enabling CPO employees to add discounts through offers. The computation of the price is automatically handled by the microservice, based on pricing policies imposed by the CPO employees. This microservice assumes an important role in the “CPMS automatically sets price for a charging station” [UC.13] and “CPO employee sets offers for a charging station” [UC.14] use cases.

The data managed by this microservice needs to face complex queries. We chose a relational database as the data store.

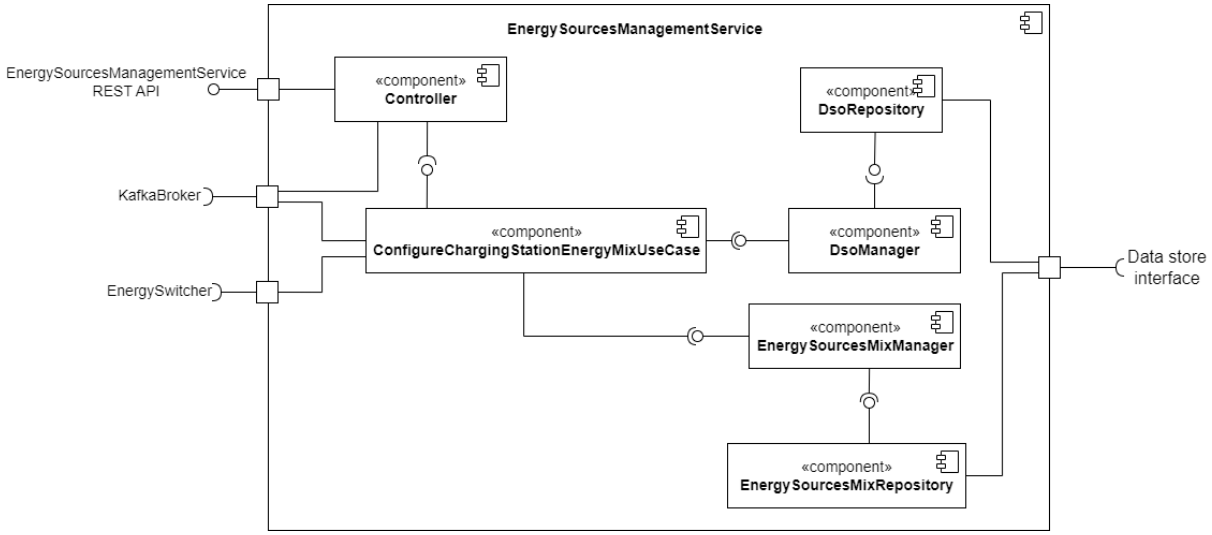


ENERGY SOURCES MANAGEMENT SERVICE

This microservice is responsible for handling the change in the mix of sources of energy. It automatically computes the best mix given the prices of the partnered DSOs and allows the CPO employees to manually change the mix. This microservice communicates with the energy switcher and makes sure that the configured mix is actually applied in the real world.

This microservice assumes an important role in the “CPO Employee manually changes the mix of energy sources for a charging station” [UC.15], “CPMS automatically changes the mix of energy sources for a charging station” [UC.16] and “CPMS operates a change in the mix of energy sources for a charging station through the Energy Switcher” [UC.17] use cases.

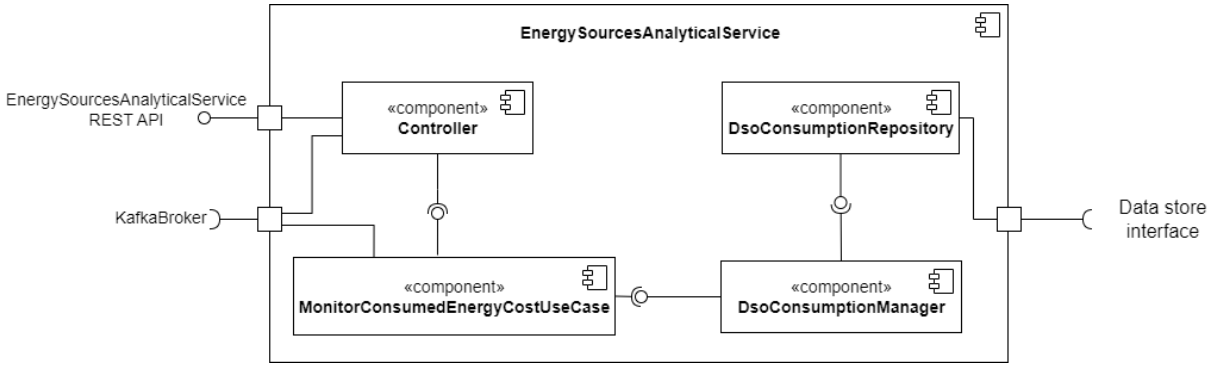
The processes handled by this microservice are complex and need strong consistency guarantees. We chose a relational database as the data store.



ENERGY SOURCES ANALYTICAL SERVICE

This microservice is responsible for keeping track of the consumption of energy from each of the sources, consuming events published by ChargingManagementService on the broker. It offers the CPO employees an overview of the cost of the consumed energy.

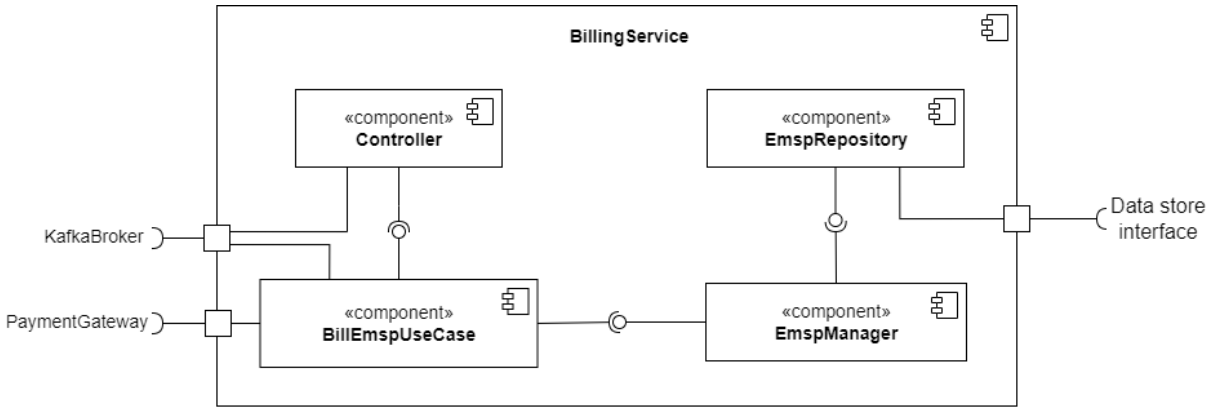
This microservice needs to store a large volume of historical data and perform complex analytical queries on it. We chose a relational database as the operational data store, and a data warehouse in which historical data are imported from the operational database. Complex analytical queries will be performed in the data warehouse.



BILLING SERVICE

The microservice is responsible for billing the eMSPs based on the charges of the customers.

The business processes performed by this microservice are crucial and need strong transactional consistency. We chose a relational database as the data store.



2.3 Deployment view

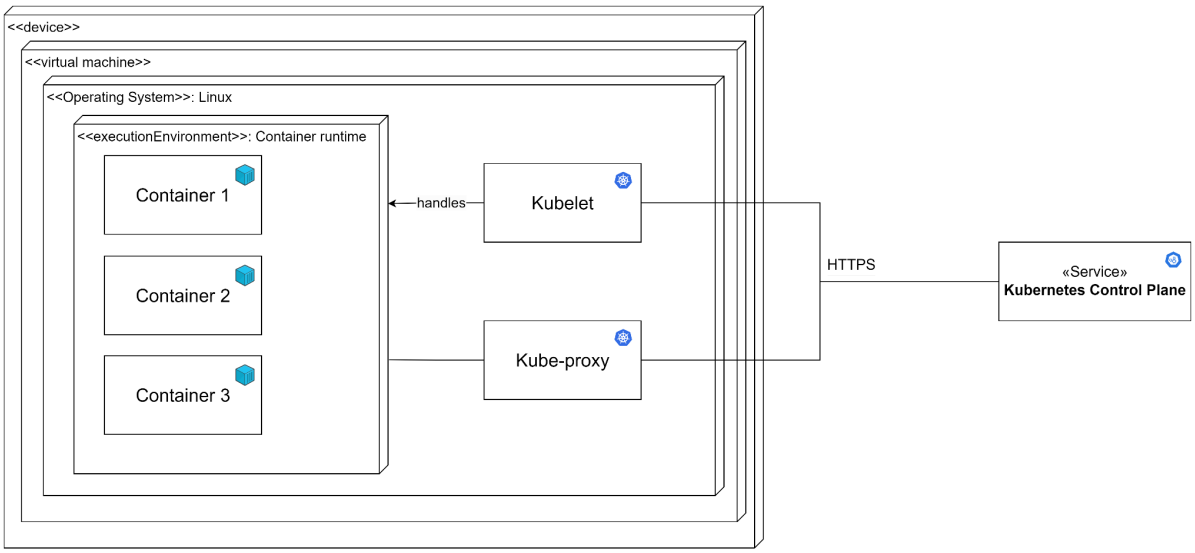
2.3.1 Introduction

As already mentioned in section 2.1, our architecture needs to be dynamically scalable. To achieve this requirement, we decided to deploy our systems on the Cloud.

We employed an IaaS model, letting the provider handle the storage, networking, and virtualization infrastructure.

Furthermore, we decided to employ Kubernetes, leveraging automatic deployment, scaling, and management of containerized applications.

The computational nodes of our deployment architecture are homogeneous. They only act as nodes in the Kubernetes cluster:



Each node runs a set of containers and hosts the kubelet and kube-proxy processes. These processes communicate with the Kubernetes Control Plane and constitute the link between the computational resource provided by the IaaS provider and the concept of a node in Kubernetes.

Once this link is enforced, we can configure Kubernetes resources to handle the deployment of our systems.

2.3.2 Kubernetes resources

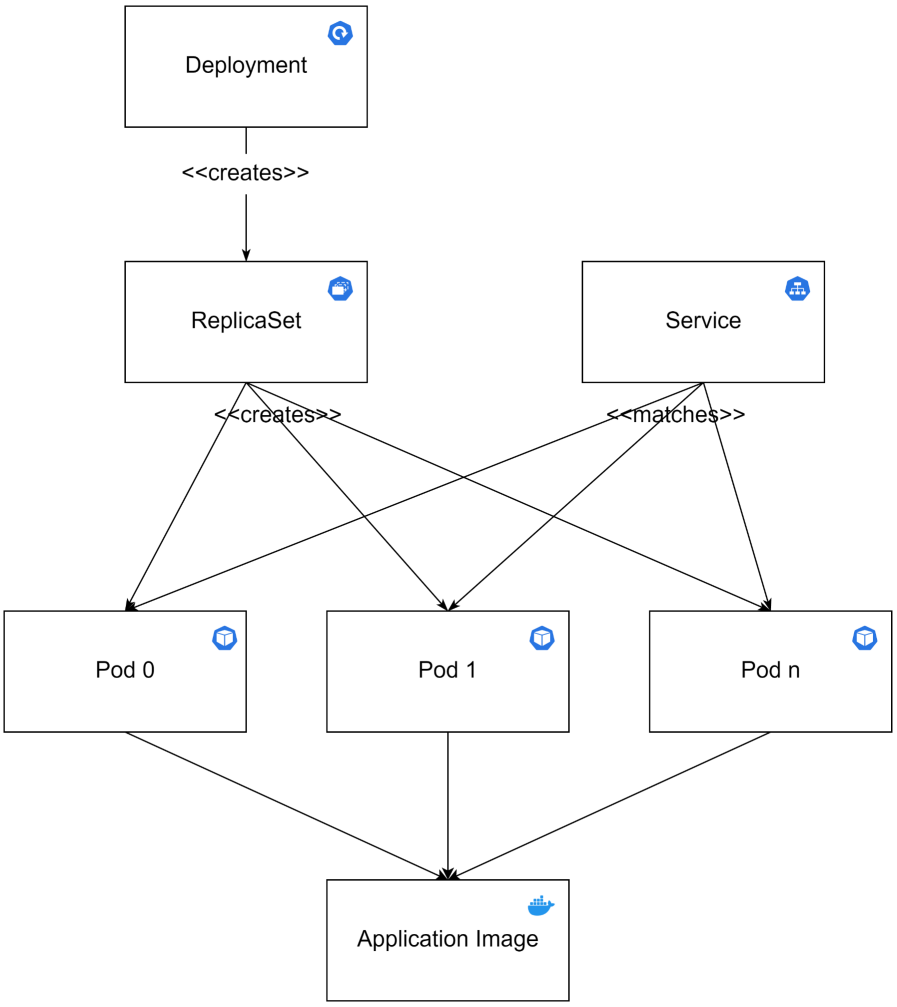
In the following paragraphs, there is a description of the Kubernetes resources that will be created in order to deploy the various parts of our systems.

Stateless applications deployment

The microservices will be deployed as stateless applications. A Deployment creates all the other necessary resources to deploy the microservice containerized application.

To access the application, a Service must also be created.

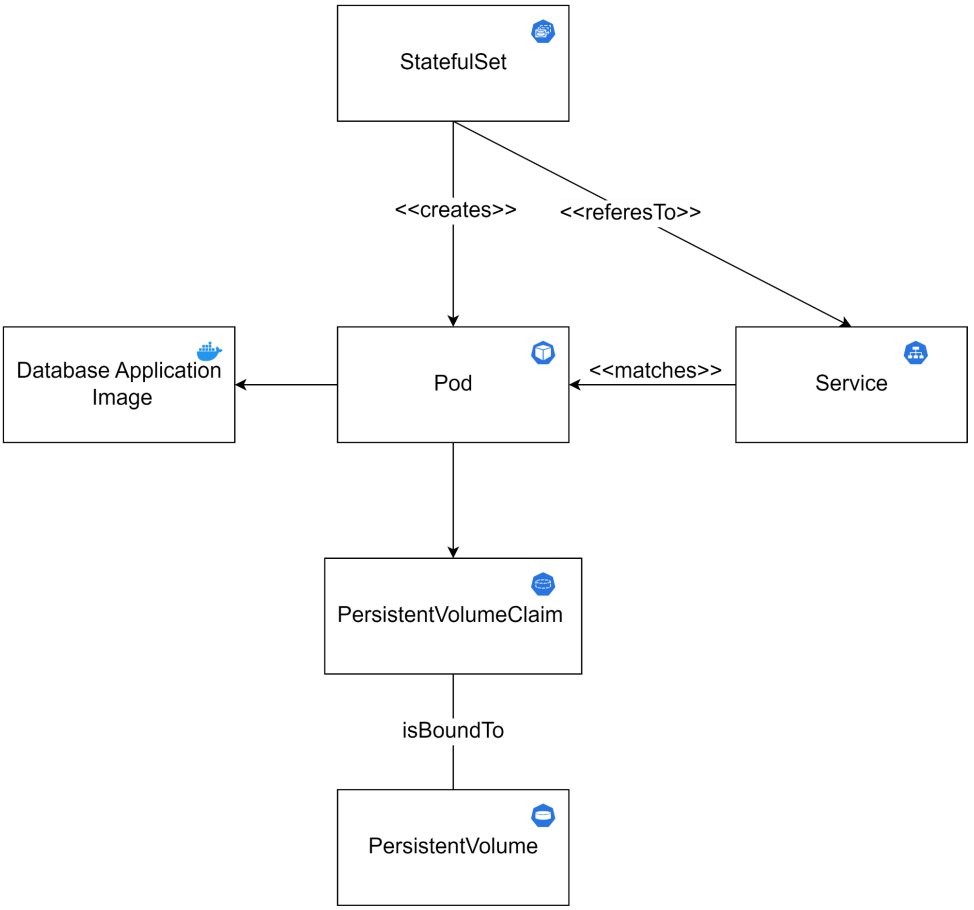
We use the same resources to deploy the instances of Nginx as well (only present for the CPMS). They will serve the static files necessary for the CPMS web app.



Data store deployment

The various data stores will be deployed as stateful applications. A StatefulSet creates all the other necessary resources to deploy the data store.
The pods running the management system application will access the distributed persistent storage by means of the PersistentVolume bound to their PersistentVolumeClaim.

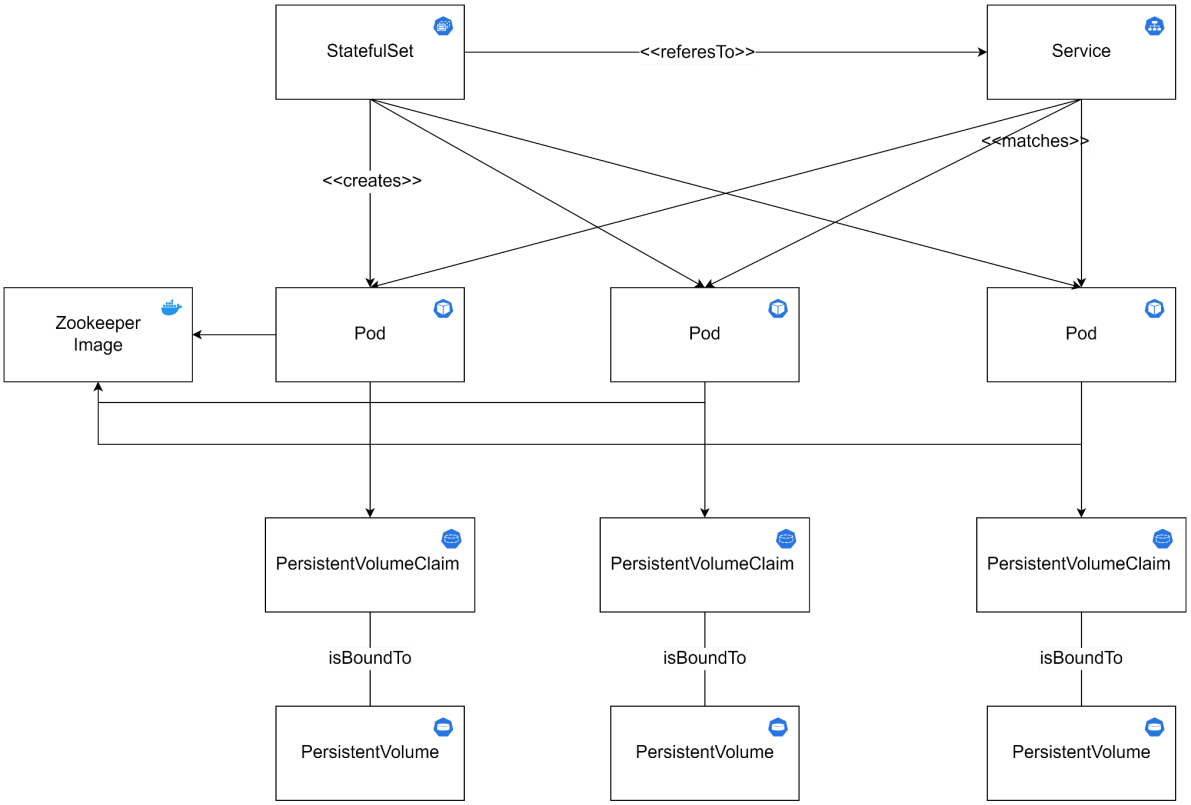
To access the application, a Service is also created.

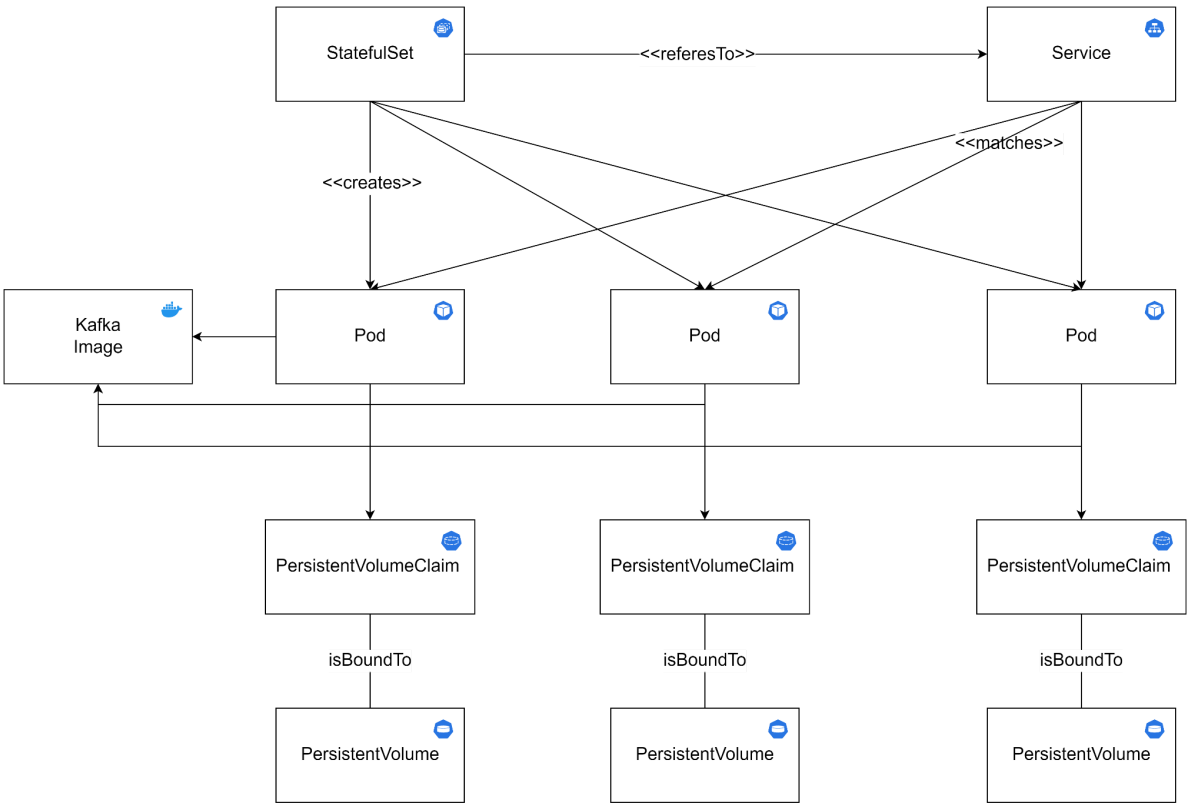


Kafka broker

Kafka brokers need ZooKeeper, a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

The deployment of ZooKeeper and Kafka is structurally similar. They both are stateful applications, so we use the StatefulSet resource. They are also distributed systems organized in clusters. The StatefulSet resource is a good fit for these systems because it allows it to carefully handle multiple instances, stick identities to each instance and gradually add or remove instances when scaling up or down.


StatefulSet
<<referesTo>>
Service
<<creates>>
<<matches>>
Zookeeper Image
Pod
Pod
Pod
PersistentVolumeClaim
PersistentVolumeClaim
PersistentVolumeClaim
isBoundTo
isBoundTo
isBoundTo
PersistentVolume
PersistentVolume
PersistentVolume


StatefulSet
<<referesTo>>
Service
<<creates>>
<<matches>>
Kafka Image
Pod
Pod
Pod
PersistentVolumeClaim
PersistentVolumeClaim
PersistentVolumeClaim
isBoundTo
isBoundTo
isBoundTo
PersistentVolume
PersistentVolume
PersistentVolume

2.4 Runtime view

In this section we explain some general flows of interactions supporting processes that are common among all microservices.

Moreover, we show some of the meaningful interactions between the main components of eMSP and CPMS systems.

2.4.1 Common interactions

State propagation

In a microservice architecture, each microservice is a component with restricted responsibility and highly specialized on the tasks it needs to perform. Even though the microservices should be loosely coupled, all of them participate in the reaching of the goals of the overall system. In particular, they need mechanisms to share state and knowledge about events happening in the system.

Sharing data store between microservices is not an option, since they would be highly coupled between one another and they could hardly be independently developed, deployed and scaled. Moreover different microservices might need different data store requirements (as in the case at hand).

To solve this problem, we leverage the message broker shared between all microservices. When a microservice that owns a piece of information needs to propagate a change, it publishes an integration event on the broker. All interested microservices will listen to the event and will update their local representation of the shared information. This way, each microservice will use its data store also as a cache of data owned by other microservices and will update its local cache using the integration events.

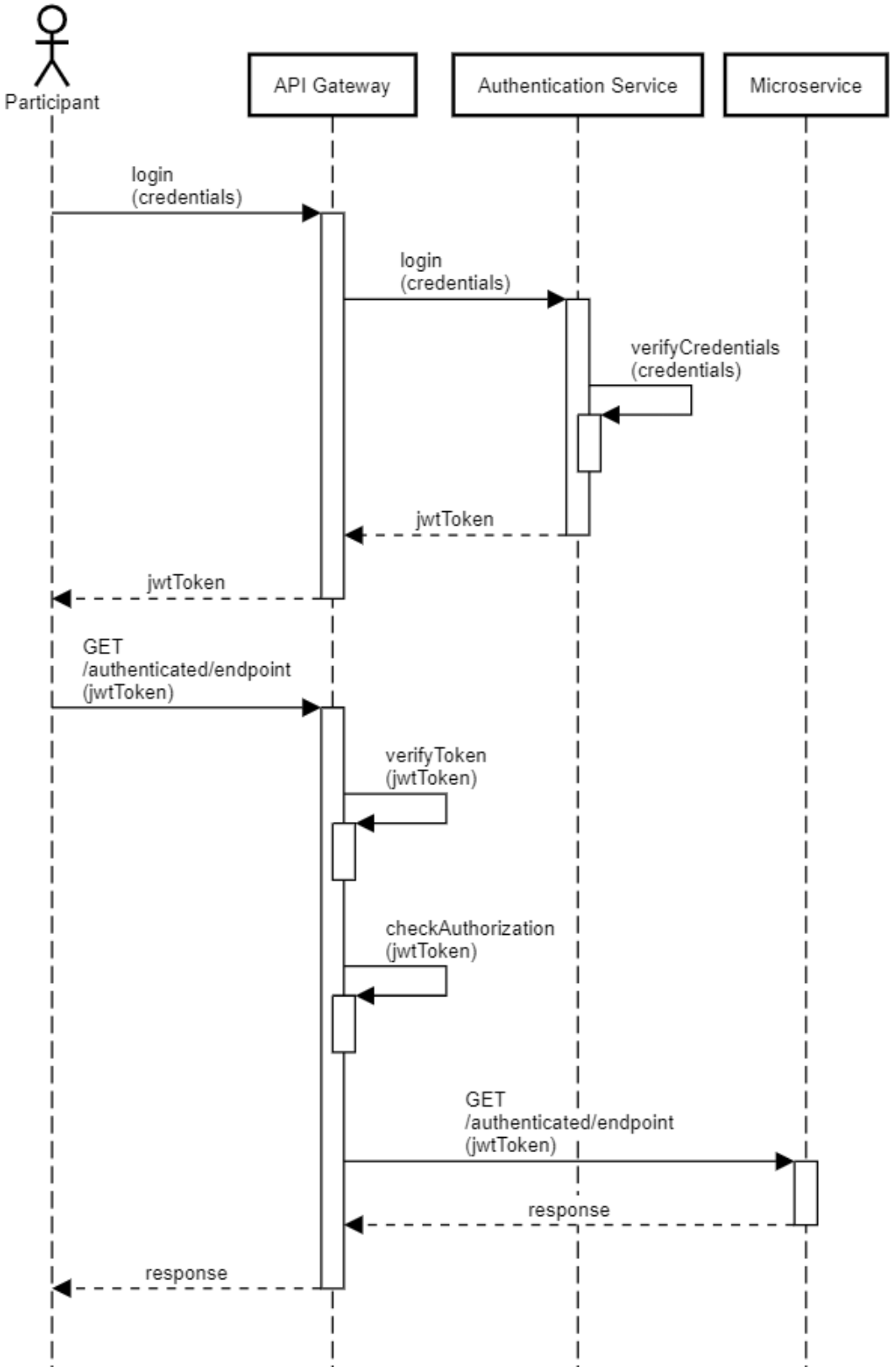
The biggest benefits of this approach are the low coupling between microservices and the higher availability due to the fact that when a microservice needs a piece of information owned by another microservice, it can fetch it from its own data store without having to rely on an HTTP call to the owner.

The downside of this approach is that data owned by some microservices could be stale in the local caches of other microservices. For this reason, this mechanism must be accompanied by direct HTTP calls in the case of stronger consistency needs (at the cost of availability).

Authentication and authorization workflow

Another common workflow is the one related to authentication and authorization. The diagram below shows a typical interaction.

In the sequence diagrams associated with the use cases, such interaction is omitted.

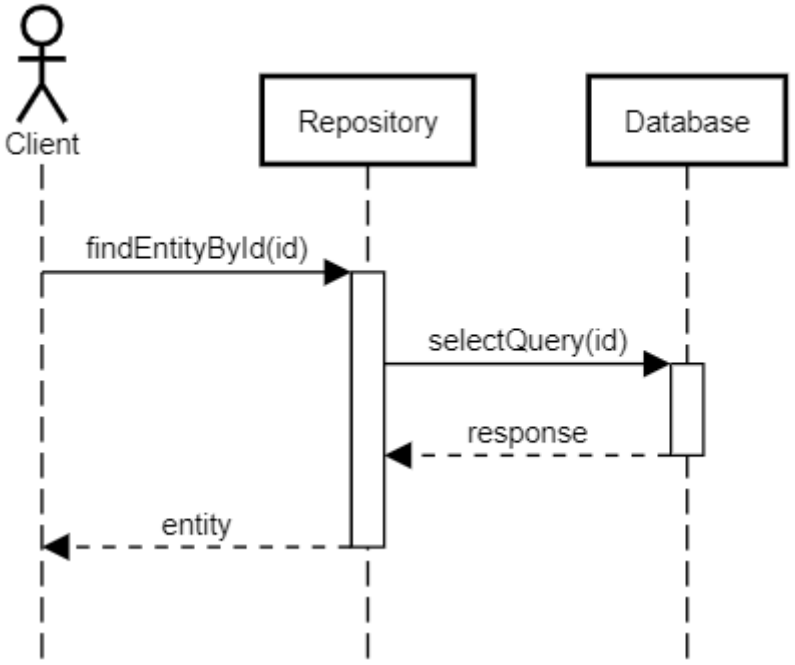

Participant
API Gateway
Authentication Service
Microservice
login
(credentials)
login
(credentials)
verifyCredentials
(credentials)
jwtToken
jwtToken
GET
/authenticated/endpoint
(jwtToken)
verifyToken
(jwtToken)
checkAuthorization
(jwtToken)
GET
/authenticated/endpoint
(jwtToken)
response
response

Data access workflow

We used the repository pattern to abstract the data access layer away.

Here is a simple example of an interaction between a repository and the associated data store.

In the sequence diagrams associated with the use cases, these interactions are omitted.



Synchronous vs asynchronous calls

Both synchronous and asynchronous calls are used in the interactions between components.

Our conventions about synchronous and asynchronous calls are:

- Synchronous calls are the ones for which the caller participant, before continuing with its task, needs to wait until the callee participant terminates the execution;
- Asynchronous calls are the ones for which the caller participant can continue its task concurrently with the execution of the callee participant;
- In the following diagrams synchronous calls are always accompanied by an explicit return message (optionally empty);
- In the following diagrams asynchronous calls are not accompanied by an explicit return message. We assume that there is an implicit return message associated with the acknowledgement of the HTTP call. This means that, after having received the acknowledgement, the caller participant can proceed with its execution concurrently with the callee execution, without waiting for the result of the callee computation.

eMSP and CPMS systems know how to communicate with other parties because they both have components responsible for keeping the linking between other parties and the associated systems.

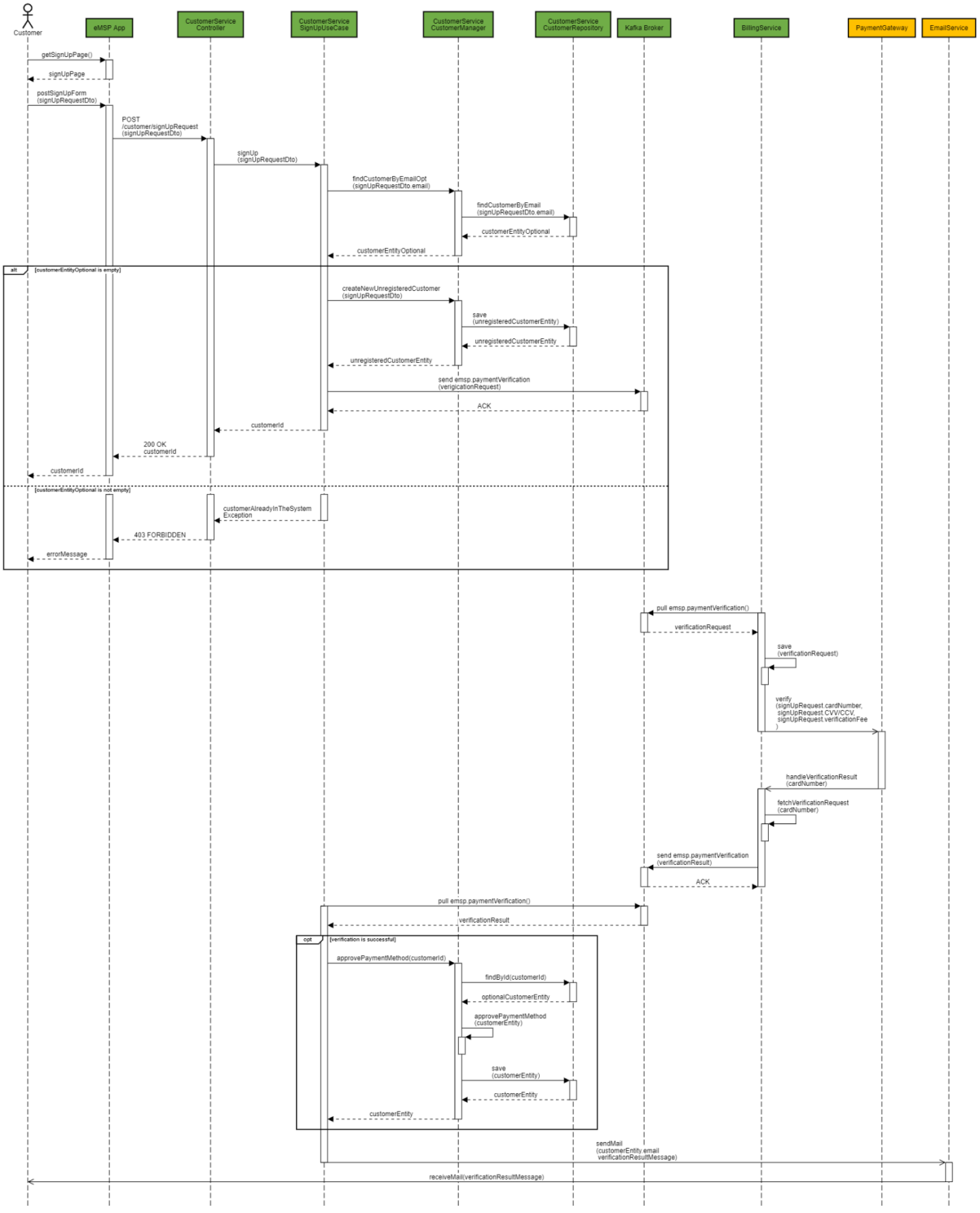
In particular, eMSP has the CpoCatalogService that is responsible for keeping the association between the CPO identifiers and the CPMS associated with each CPO.

CPMS system has the EmspCatalogService and DsoCatalogService that are responsible for keeping the association between the eMSPs and the DSOs and their associated systems.

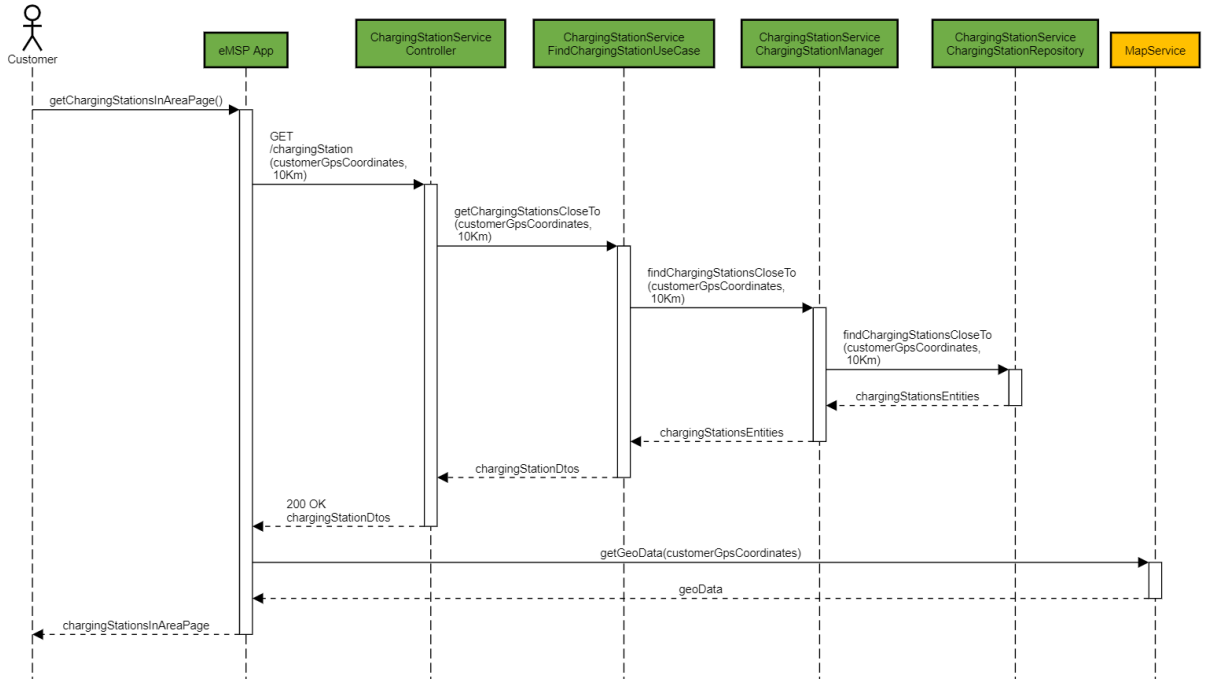
In this way, both eMSP and CPMS components can connect with other parties knowing the other parties' identifiers. In the sequence diagrams we will assume that each component knows the exact endpoint to call to communicate with the correct external system.

2.4.2 Main use case interactions

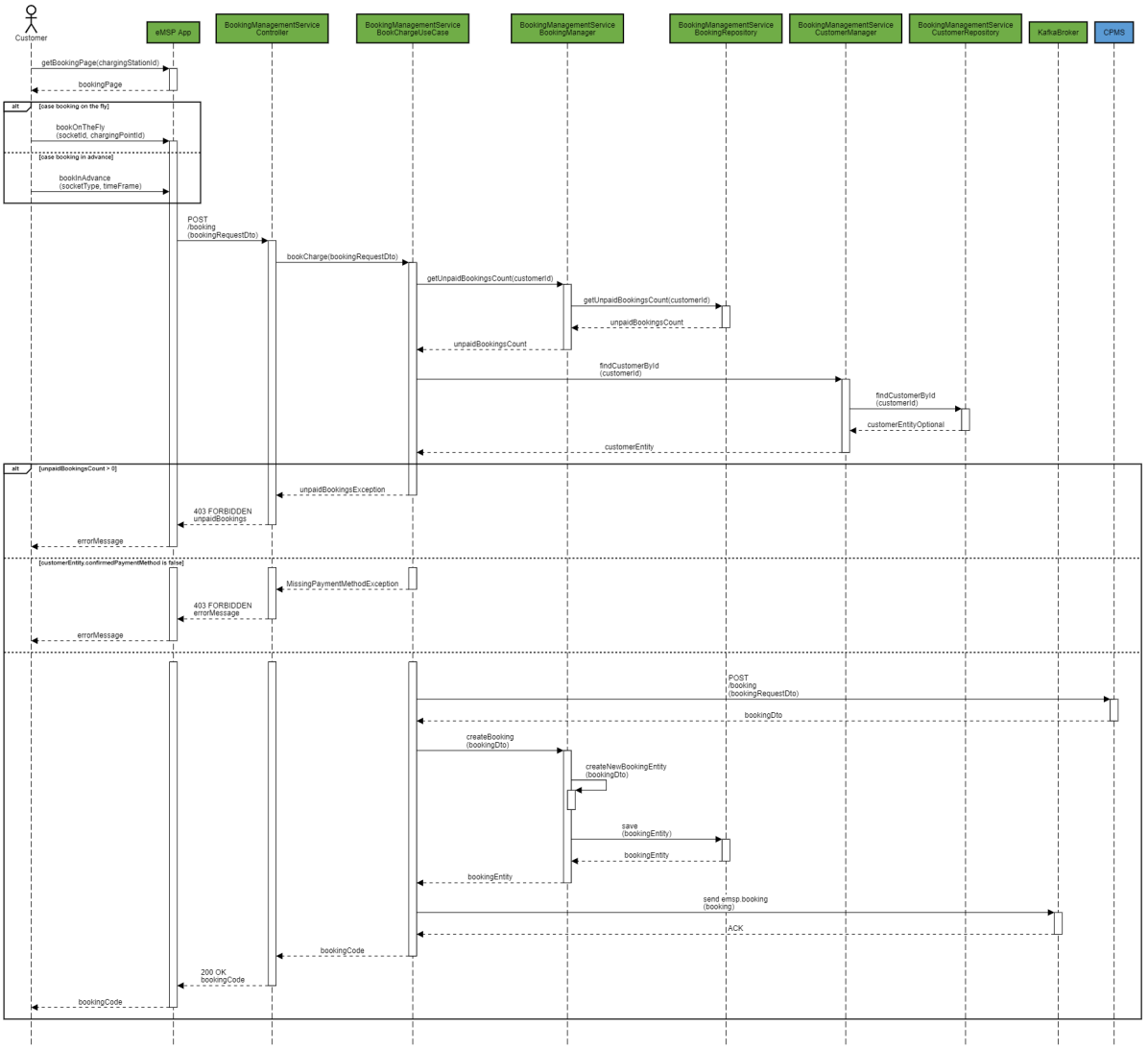
User signs up for the eMSP [UC.1, SQD.1]

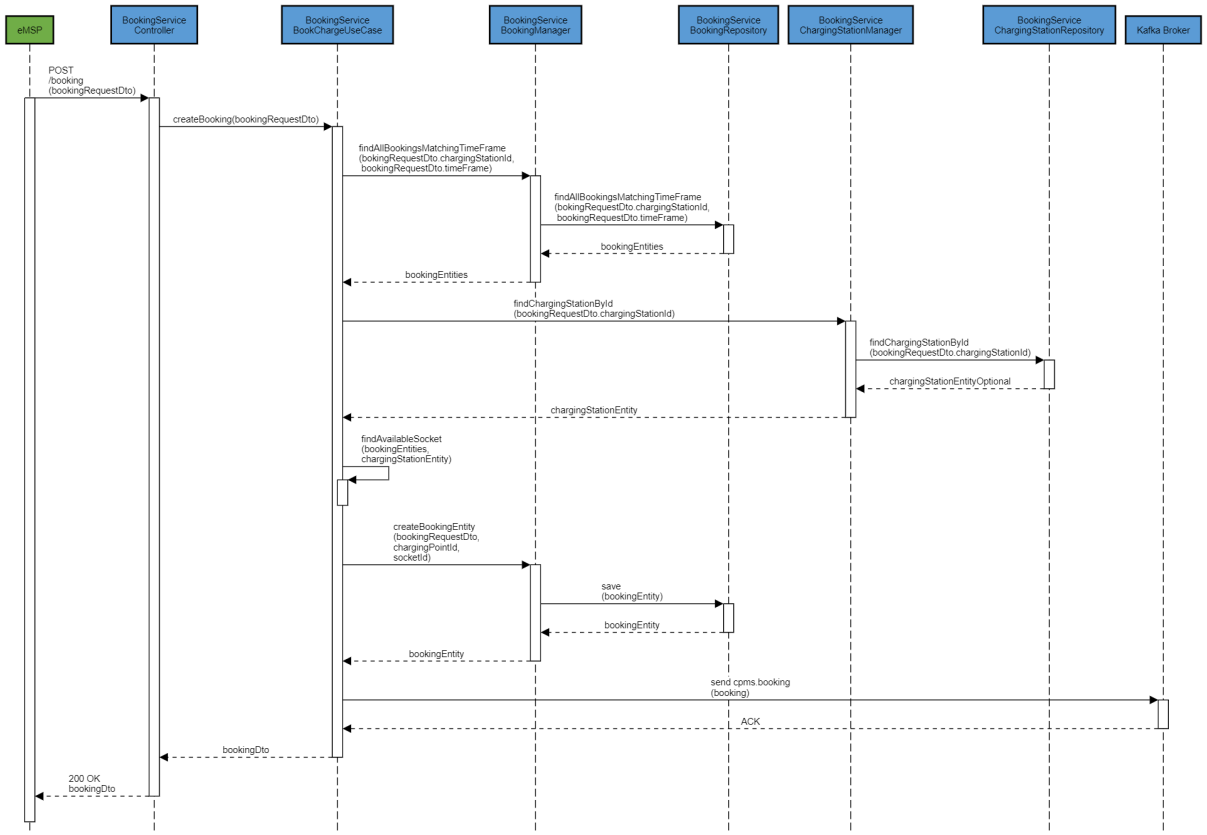


Customer finds a charging station [UC.4, SQD.4]

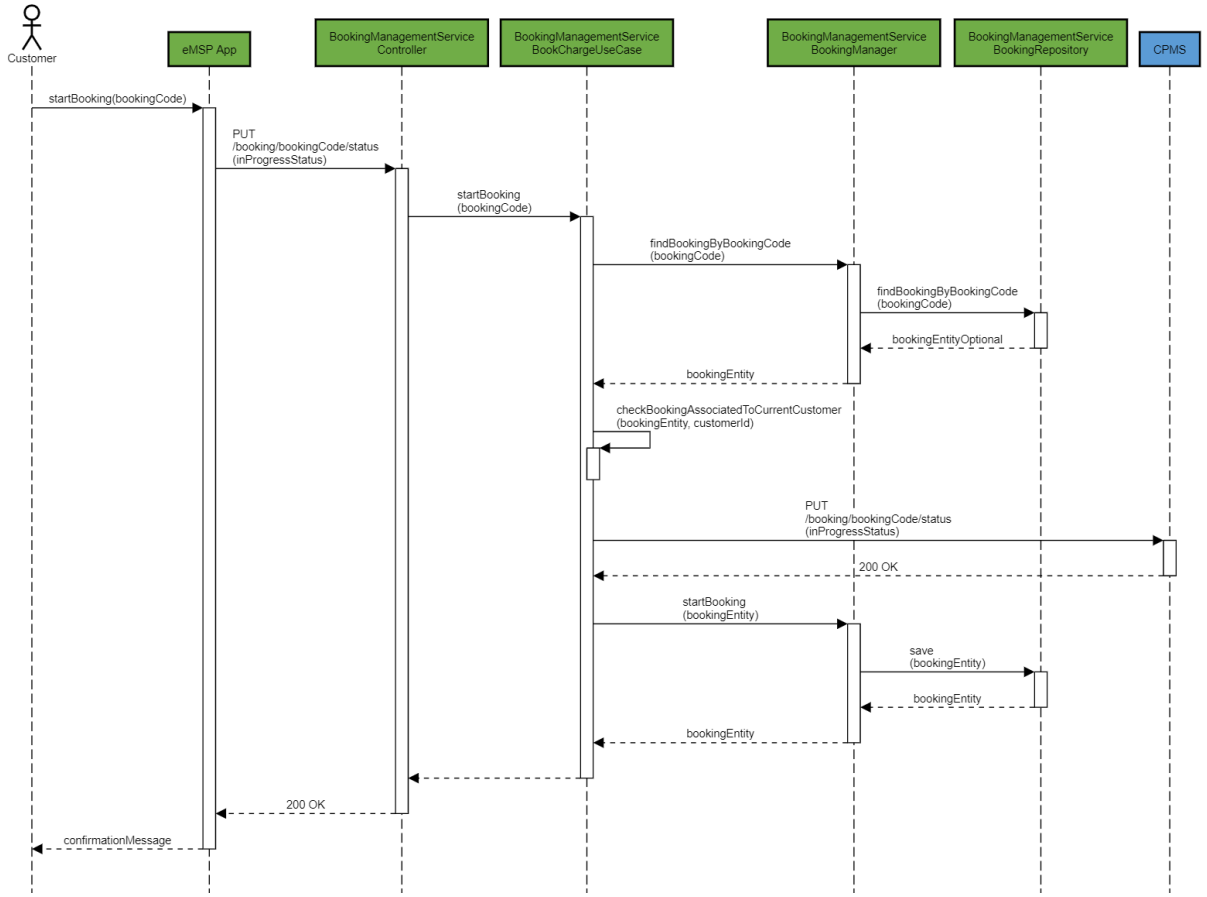


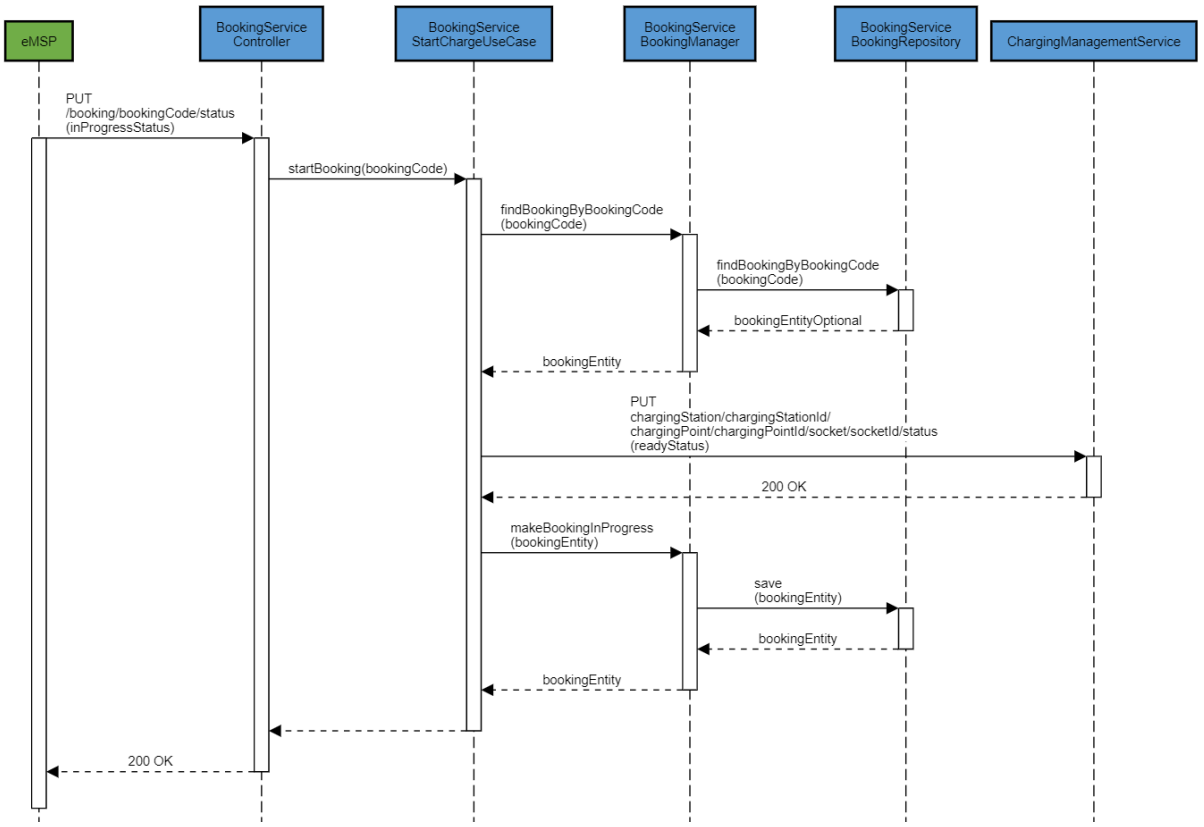
Customer books a charge [UC.5, SQD.5]

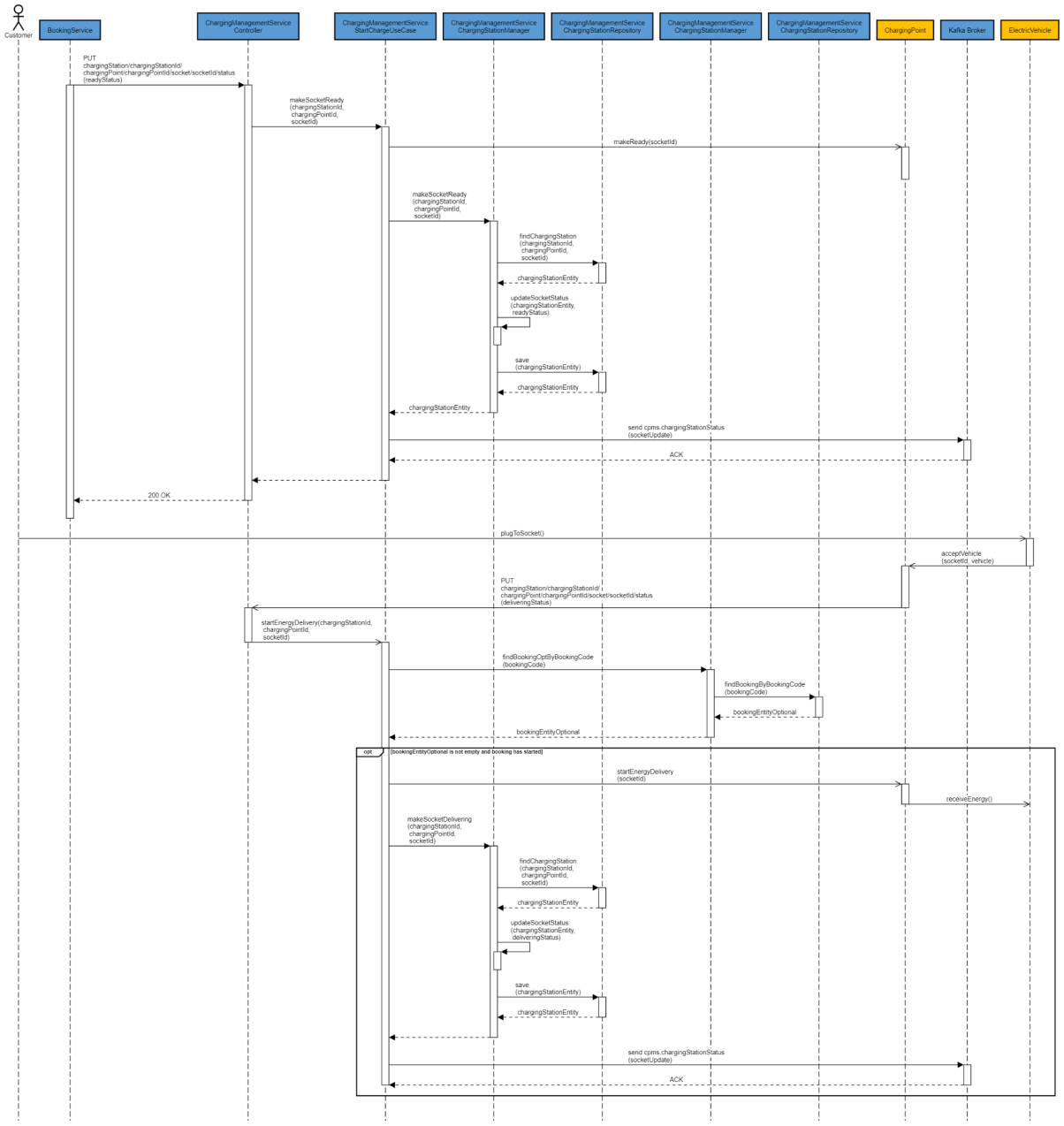



eMSP
BookingService Controller
BookingService BookChargeUseCase
BookingService BookingManager
BookingService BookingRepository
BookingService ChargingStationManager
BookingService ChargingStationRepository
Kafka Broker
POST /booking (bookingRequestDto)
createBooking(bookingRequestDto)
findAllBookingsMatchingTimeFrame (bokingRequestDto.chargingStationId, bookingRequestDto.timeFrame)
findAllBookingsMatchingTimeFrame (bokingRequestDto.chargingStationId, bookingRequestDto.timeFrame)
bookingEntities
bookingEntities
findChargingStationById (bookingRequestDto.chargingStationId)
findChargingStationById (bookingRequestDto.chargingStationId)
chargingStationEntityOptional
chargingStationEntity
findAvailableSocket (bookingEntities, chargingStationEntity)
createBookingEntity (bookingRequestDto, chargingPointId, socketId)
save (bookingEntity)
bookingEntity
bookingEntity
send cpms.booking (booking)
ACK
bookingDto
200 OK bookingDto

Customer starts charging their vehicle [UC.7, SQD.7]



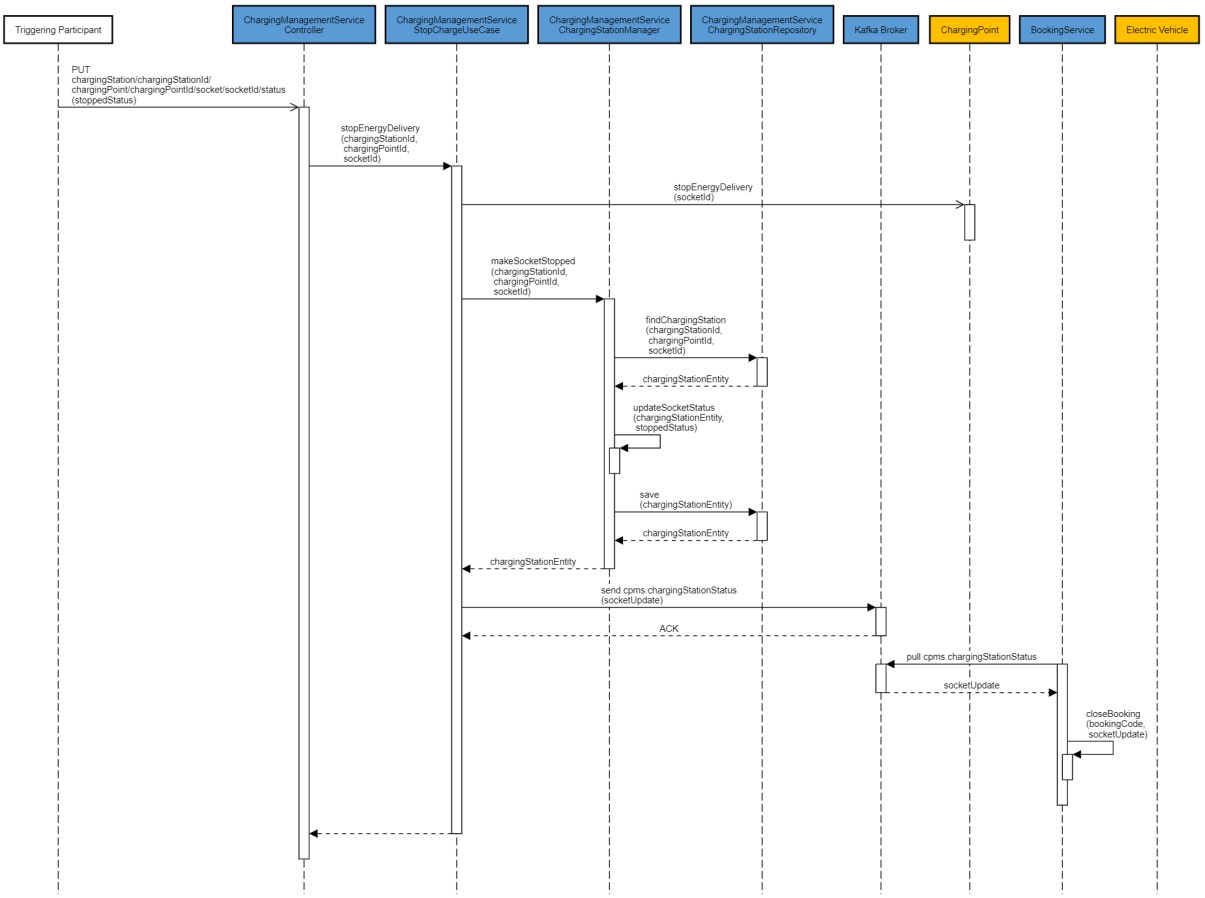

eMSP
BookingService Controller
BookingService StartChargeUseCase
BookingService BookingManager
BookingService BookingRepository
ChargingManagementService
PUT /booking/bookingCode/status (inProgressStatus)
startBooking(bookingCode)
findBookingByBookingCode (bookingCode)
findBookingByBookingCode (bookingCode)
bookingEntityOptional
bookingEntity
PUT chargingStation/chargingStationId/ chargingPoint/chargingPointId/socket/socketId/status (readyStatus)
200 OK
makeBookingInProgress (bookingEntity)
save (bookingEntity)
bookingEntity
bookingEntity
200 OK

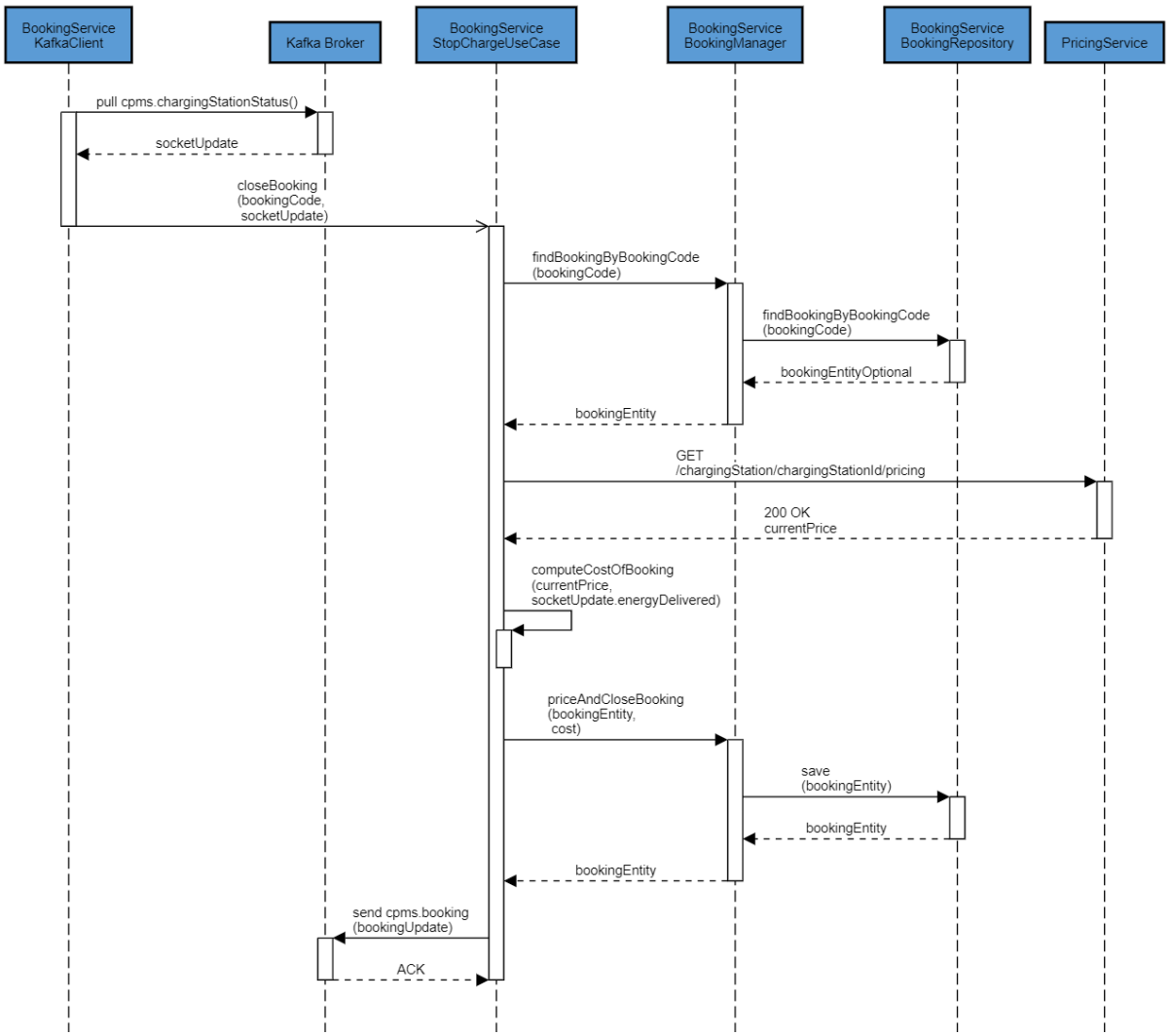


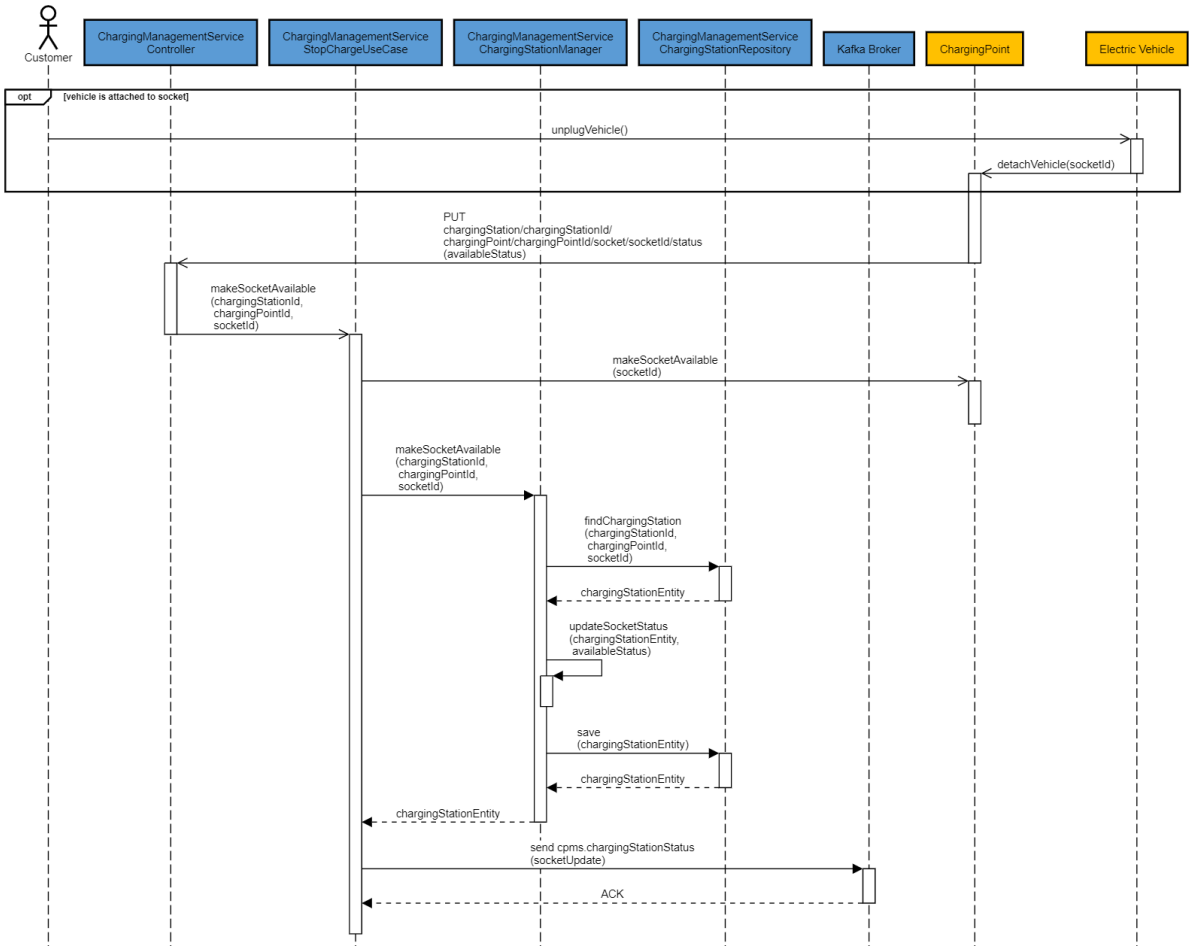
The delivery of energy to a vehicle stops [UC.11, SQD.11]

The Triggering participant of this interaction abstracts away the cause of the ending of the charge session. The causes could be:

- The battery of the electric vehicle is fully charged [UC.8];
- The timeframe selected in the booking has expired [UC.9];
- The socket that was charging has been unplugged from the electric vehicle [UC.10].


Triggering Participant
ChargingManagementService Controller
ChargingManagementService StopChargeUseCase
ChargingManagementService ChargingStationManager
ChargingManagementService ChargingStationRepository
Kafka Broker
ChargingPoint
BookingService
Electric Vehicle
PUT chargingStation/chargingStationId/ chargingPoint/chargingPointId/socket/socketId/status (stoppedStatus)
stopEnergyDelivery (chargingStationId, chargingPointId, socketId)
stopEnergyDelivery (socketId)
makeSocketStopped (chargingStationId, chargingPointId, socketId)
findChargingStation (chargingStationId, chargingPointId, socketId)
chargingStationEntity
updateSocketStatus (chargingStationEntity, stoppedStatus)
save (chargingStationEntity)
chargingStationEntity
chargingStationEntity
send cpms.chargingStationStatus (socketUpdate)
ACK
pull cpms.chargingStationStatus
socketUpdate
closeBooking (bookingCode, socketUpdate)


BookingService KafkaClient
Kafka Broker
BookingService StopChargeUseCase
BookingService BookingManager
BookingService BookingRepository
PricingService
pull cpms.chargingStationStatus()
socketUpdate
closeBooking (bookingCode, socketUpdate)
findBookingByBookingCode (bookingCode)
findBookingByBookingCode (bookingCode)
bookingEntityOptional
bookingEntity
GET /chargingStation/chargingStationId/pricing
200 OK currentPrice
computeCostOfBooking (currentPrice, socketUpdate.energyDelivered)
priceAndCloseBooking (bookingEntity, cost)
save (bookingEntity)
bookingEntity
bookingEntity
send cpms.booking (bookingUpdate)
ACK



2.5 Component interfaces

In this section we list the REST API interfaces offered by the microservices of the eMSP and CPMS systems.

eMSP

Customer Service

- /customer/login (POST)
- /customer/signUpRequest (POST)
- /customer/{customerId}/signUpRequest (GET)
- /customer/{customerId} (GET, PUT, DELETE)

Booking Management Service

- /customer/{customerId}/booking (GET, POST)
- /customer/{customerId}/booking/{bookingCode} (GET, PUT, DELETE)
- /customer/{customerId}/booking/{bookingCode}/status (GET, PUT)

Cpo Catalog Service

- /cpo (GET, POST)
- /cpo/{cpoId} (GET, PUT, DELETE)

Charging Station Service

- /chargingStation (GET)
- /chargingStation/{chargingStationId} (GET)

CPMS

Employee Service

- /employee/login (POST)
- /employee (POST, GET)
- /employee/{employeeId} (GET, PUT, DELETE)

Booking Service

- /booking (GET, POST)
- /booking/{bookingCode} (GET,PUT, DELETE)
- /booking/{bookingCode}/status (GET, PUT)

Charging Management Service

- /chargingStation/{chargingStationId}/chargingPoint/{chargingPointId}/socket/{socketId}/status (GET, PUT)

Charging Station Configuration Service

- /chargingStation (GET, POST)
- /chargingStation/{chargingStationId} (GET, PUT, DELETE)

Charging Station Status Service

- /chargingStation/{chargingStationId}/status (GET)

Emsp Catalog Service

- /emsp (GET, POST)
- /emsp/{emspId} (GET, PUT, DELETE)

Dso Catalog Service

- /dso (GET, POST)
- /dso/{dsoId} (GET, PUT, DELETE)

Pricing Service

- /chargingStation/{chargingStationId}/pricing (GET, PUT)

- /pricingPolicy (GET, POST)
- /pricingPolicy/{pricingPolicyId} (GET, PUT, DELETE)
- /dso/{dsoId}/pricing (GET)

Energy Sources Management Service

- /chargingStation/{chargingStationId}/mix (GET, PUT)

Energy Sources Analytical Service

- /chargingStation/{chargingStationId}/consumptionReport (GET)
- /chargingStation/consumptionReport (GET)
- /dso/consumptionReport (GET)
- /dso/{dsoId}/consumptionReport (GET)

2.6 Selected architectural styles and patterns

This section shows the main architectural styles and patterns employed. These patterns have already been explored throughout the whole document, but are here listed and summarized.

2.6.1 Microservices architecture

As already mentioned in section 2.1, both the eMSP and the CPMS systems are solicited by a huge volume of requests and they need to support business processes that have heterogeneous requirements. It is important to be able to independently scale different components of both systems. Moreover, the volatile and unpredictable charging demands require the ability to dynamically scale the systems.

Microservices are a natural choice for these kinds of requirements.

2.6.2 Ports and adapters architecture

Each microservice has been designed taking inspiration from the ports and adapters architecture.

The core of the microservice is the model of the domain, which is composed of domain entities and use case services. The core is solicited through REST API interfaces and by events consumed by the topics to which the microservice is subscribed. The state of the core is persisted through interfaces that abstract the persistence layer.

Business-related concerns (enforced by the core) are hence separated from the interaction with external systems (consumers of the API, message broker and data store).

This allows an easier introduction of new external participants and the update of the existing ones.

2.6.3 API style

Microservices expose their functionalities through REST interfaces. REST stands for REpresentational State Transfer, a software architectural style that describes a uniform interface between components.

The choice of this style was guided by different motivations:

- REST style is the industry standard for web application, so it is easier to interact with other external systems such as charging point devices and DSO systems;
- Due to the great complexity of the system to be developed, multiple teams need to cooperate to build it. REST provides semantic homogeneity of the outer interfaces of the components of the system. It is universally understood and hence favours a faster development.

2.6.4 API Gateway pattern

The API Gateway pattern hides the complexity of the internal components of the system offering to external participants a unified view of the system.

In the case at hand, it is also responsible for consolidating cross cutting such as authentication and authorization.

2.6.5 Publish-subscribe pattern

One possible means for state integration between microservices is the message broker, in our case Kafka.

Kafka is logically organized in topics. At a logical level, a topic is a log of events associated with a name. Components can publish events on some topics, and the components that are subscribed to those topics will consume the associated messages.

The publish-subscribe pattern allows decoupling of microservices, thanks to the fact that the interaction between them is dependent only on the message published on topics and not on the implementation of the single participating microservices.

2.7 Other design decisions

In this section we will present other design patterns and decisions that are not specifically architectural.

2.7.1 Domain modelling

At the core of each microservice is a model representing the piece of the business domain the microservice is interested in. Microservices with articulated business processes need to carefully design their model to tackle the intrinsic complexity of the interactions they participate in.

In these cases (e.g. microservices related to booking and charging processes), we adopted the concept of entities, value objects and aggregates.

Entities are types of objects (classes) for which their instances cannot be identified only by the value of their attributes: they need an identity. For example, the instances that represent the different state of the same booking (e.g. planned, in progress, ended), have different values for their attributes, but they are all associated to the same booking: they need an identity in order to be matched together.

Value objects, on the other hand, are types of objects for which their instances can be identified by the value of their attributes. For example two instances of TimeFrame that have different attributes always represent two different time frames.

An aggregate is a cluster of types of objects that we treat as a unit.

An entity within an aggregate needs to be selected as the aggregate root. Only the aggregate root is exposed outside the aggregate. This way inter-class invariants between the objects inside an aggregate are enforced by the aggregate root.

2.7.2 Circuit breaker

When microservices need to communicate with other participants, problems in communications might occur. Some of these problems might be just temporary and caused by the temporary unavailability of other participants or by sporadic network latency.

One possible way to face these issues is retrying the requests until they succeed (assuming idempotent parties and/or naturally idempotent requests).

The problem with this approach is that if naively implemented, it could aggravate network faults by flooding the communication medium with duplicated requests.

The circuit breaker pattern can be seen as a proxy of the callee participants that regulates the rate at which retries of the caller participant are actually sent.

When the number of subsequent failed retries towards the callee participant crosses a threshold, no more requests are sent to the callee. Retries will be sent again after a delay. If after the delay requests still fail, another try will be performed after a bigger delay and so on.

3 USER INTERFACE DESIGN

The user interfaces required by this project are:

- eMSP app for customers;
- CPMS app for CPO employees;
- eMSP admin panel for eMSP admin;
- CPMS admin panel for CPO admin.

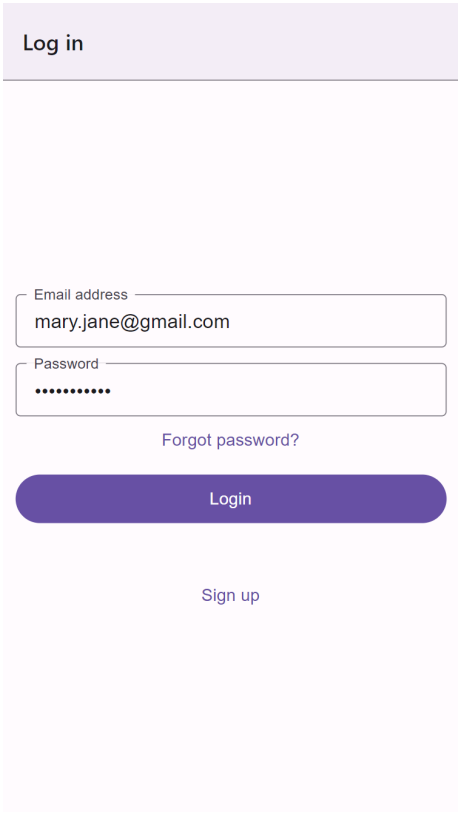
In the scope of this document, only the two first interfaces will be detailed. The admin panels required for configuration would indeed have a really standard interface design, which would let users fill in forms but also import data from files (e.g. .xlsx, .csv).

3.1 eMSP mobile app

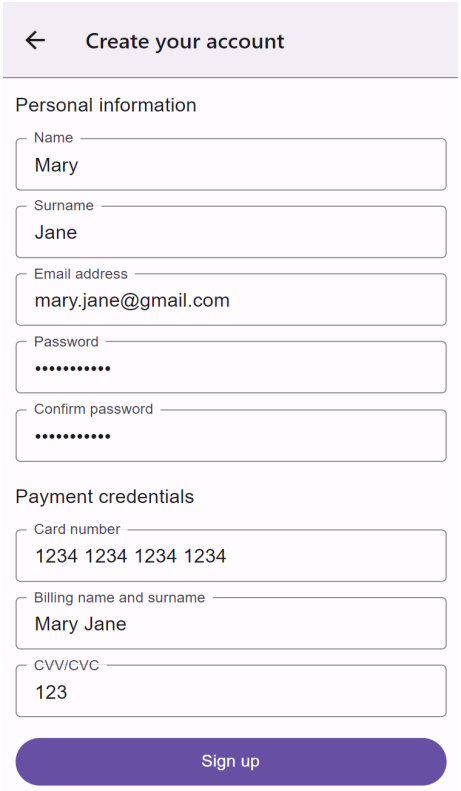
To favour the access of a very wide public, a mobile app will be implemented as eMSP app. This way, customers will be able to access the system with the only requirement of having a mobile phone with Internet connection.

The following mockups show all the pages that will compose the mobile app.

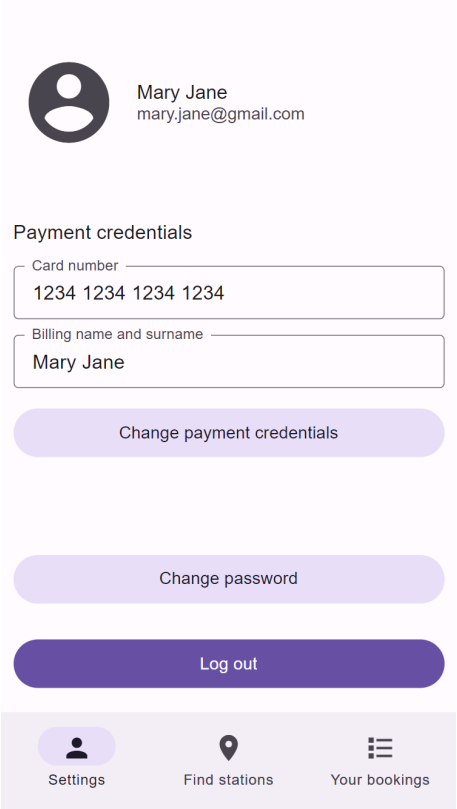
Login page



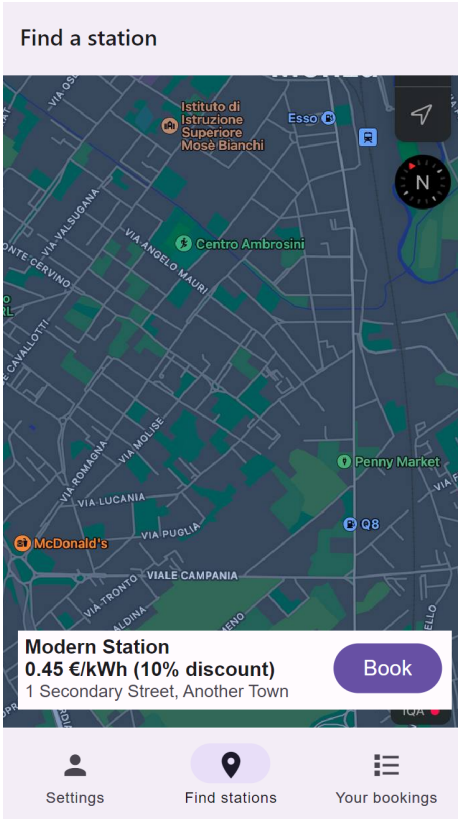
Signup page



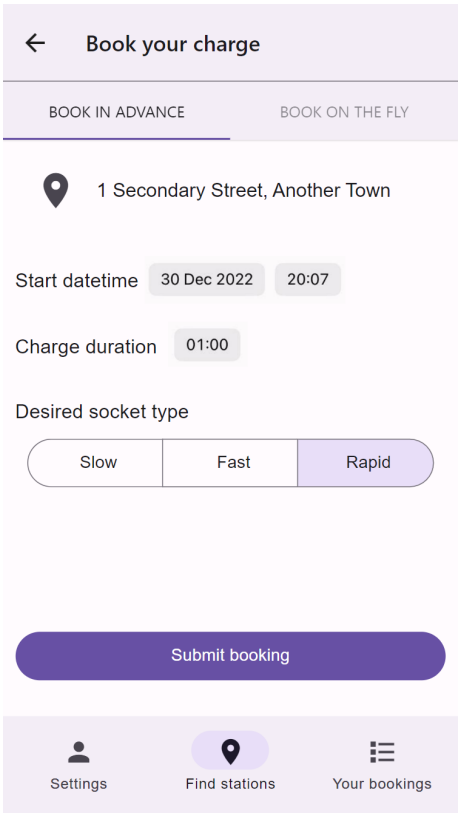
Profile settings page



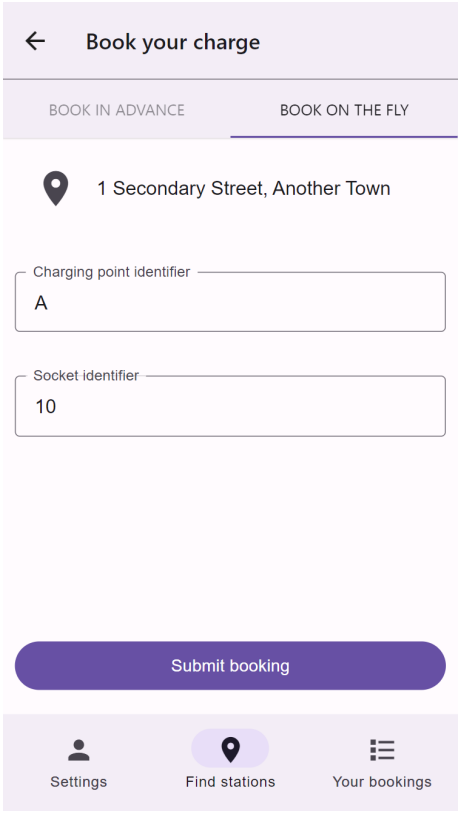
Find stations page



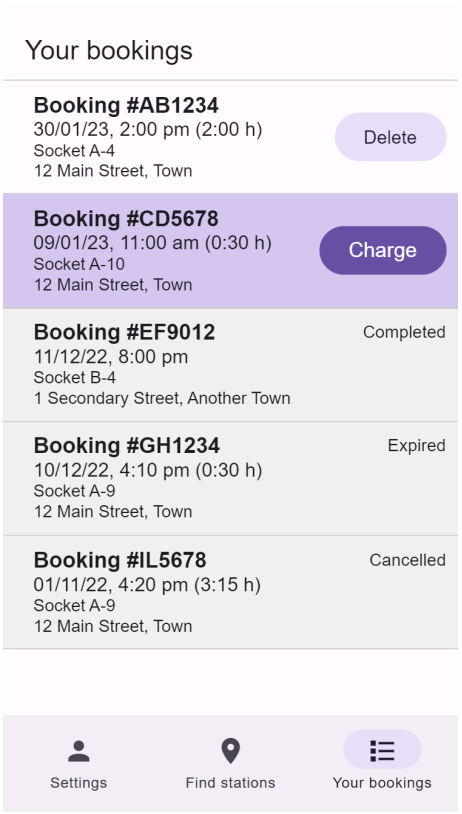
Book charge in advance page



Book charge on the fly page



Booking list page



3.2 CPMS web app

Most access to CPMS will be made by employees through their tablets or computers at the office, so a desktop web app will be implemented as CPMS app.

The following mockups show the main pages that will compose the web app.

Charging station list page

Charging station status page

< Back to Charging station list

Go Electric Station

Via Vittorio Veneto, 8 - Milan

Charging points in advance

A			
1	Available		
2	Power absorbed Expected time left	10 kWt 00:50 h	
3	Power absorbed Expected time left	80 kWt 00:40 h	
4	Power absorbed Expected time left	10 kWt 01:30 h	

B			
1	Available		
2	Available		
3	Power absorbed Expected time left	30 kWt 00:30 h	
4	Available		

C			
1	Power absorbed Expected time left	95 kWt 00:05 h	
2	Available		
3	Power absorbed Expected time left	10 kWt 00:20 h	
4	Power absorbed Expected time left	50 kWt 00:30 h	

D			
1	Available		
2	Power absorbed Expected time left	10 kWt 03:50 h	
3	Available		
4	Available		

E		
1	Available	
2	Available	
3	Available	
4	Available	

F		
1	Available	
2	Available	
3	Available	
4	Available	

Charging points on the fly

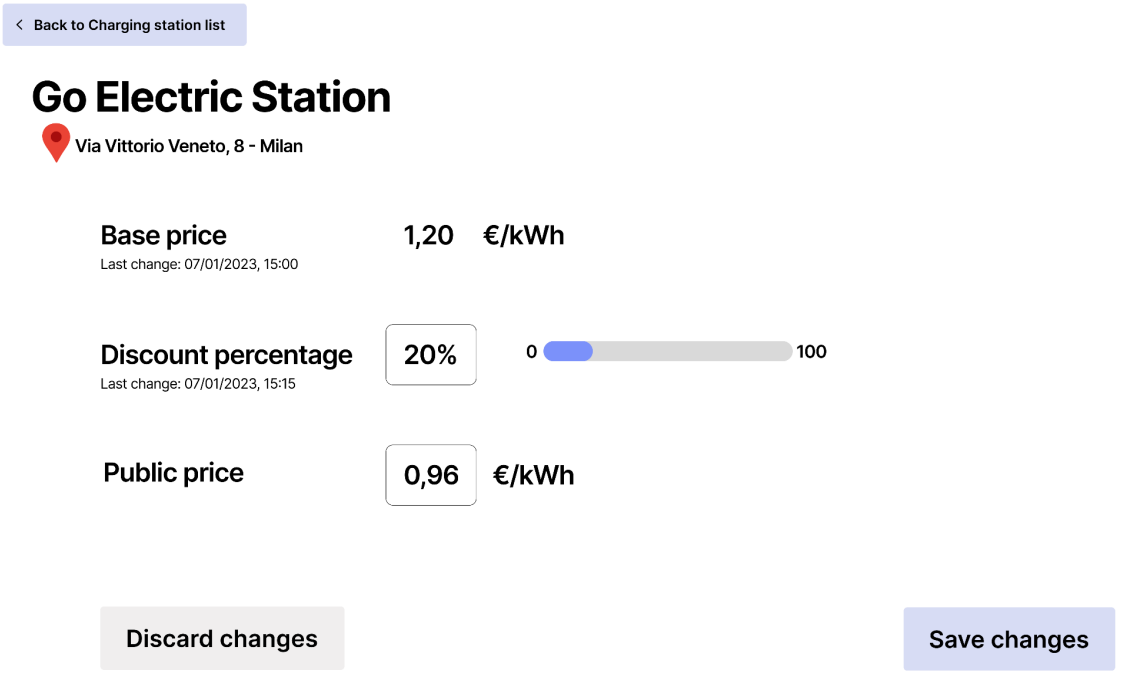
G			
1	Available		
2	Power absorbed	10 kWt	
3	Power absorbed	80 kWt	
4	Available		

H		
1	Available	
2	Available	
3	Available	
4	Available	

I			
1	Available		
2	Available		
3	Power absorbed	10 kWt	
4	Available		

J			
1	Available		
2	Power absorbed	100 kWt	
3	Available		
4	Power absorbed	5 kWt	

Charging station pricing page



Charging station energy sources page

< Back to Charging station list

Go Electric Station

Via Vittorio Veneto, 8 - Milan

Activate manual changes in the mix of energy sources

Battery

72%

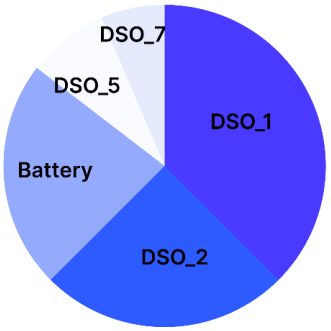
Total capacity: **10000 kWh**

Use battery as source of energy	Store energy in the battery	Disconnect the battery from the grid

Mix of energy sources

Last change: 07/01/2023, 08:30

Battery	30	%
DSO_1 Price: 0,10 €/kWh	40	%
DSO_2 Price: 0,11 €/kWh	20	%
DSO_5 Price: 0,15 €/kWh	5	%
DSO_7 Price: 0,15 €/kWh	5	%



Discard changes

Save changes

4 REQUIREMENTS TRACEABILITY

This section maps the designed microservices to the requirements defined in the RASD document: each requirement is associated to all the microservices which cooperate to its satisfaction.

eMSP microservices	Mapped requirements
Customer Service	R.1, R.2
Booking Management Service	R.18, R.19, R.20, R.21, R.23, R.26, R.27, R.28, R.29, R.30, R.36
Cpo Catalog Service	R.7
Charging Station Service	R.8, R.14
Billing Service	R.46

CPMS microservices	Mapped requirements
Employee Service	R.3, R.4, R.5
Booking Service	R.22, R.24, R.25
Charging Management Service	R.11, R.31, R.32, R.33, R.34, R.35
Charging Station Configuration Service	R.9, R.10
Charging Station Status Service	R.10, R.11, R.12, R.13, R.16, R.17
Emsp Catalog Service	R.6
Dso Catalog Service	R.37
Pricing Service	R.38, R.39, R.40, R.41, R.42
Energy Sources Management Service	R.43, R.44, R.45
Energy Sources Analytical Service	R.48, R.49, R.50
Billing Service	R.47

5 IMPLEMENTATION, INTEGRATION AND TEST PLAN

5.1 Introduction

In this section we provide an overview of the implementation, integration and test plan for the discussed project. Planning the software development lifecycle is a crucial phase of the software-to-be design because it is both used to align the team with the necessities and expectations of the stakeholders and to better support internal coordination during the development.

Stakeholders' priorities and development times are the main elements which guided our planning, but we have also decided to focus on the most critical elements of the application logic first. Conducting extensive testing on them will indeed allow us to find system-breaking bugs as early as possible and so to reduce bugfix cost.

The following table summarises the functionalities (separated into eMSP's and CPMS's ones) described in the RASD document and highlights for each of them the importance for the customer and the estimated difficulty of their implementation.

Each rating will be indicated with one of the following levels: low, medium, high.

eMSP's features	Stakeholders' priority	Implementation's difficulty
Customer login	Low	Low
Customer signup	Low	Low
Find a station	Medium	High
Book a charge	High	Medium
Start a charge	High	High

CPMS's features	Stakeholders' priority	Implementation's difficulty
Employee login	Low	Low
Employee signup	Low	Low
Manage bookings	High	Medium
Manage the charge	High	High
Manage charging station's status	Medium	High
Manage charging station's pricing	Medium	Medium
Manage charging station's mix of energy sources	Medium	Medium
Monitor charging station's energy consumption and costs	Low	Medium

The architecture designed for this project enables the parallel development of microservices by multiple independent teams, each of which is responsible for building its own service and the needed functionality. To enable these teams to quickly acquire the needed knowledge about development and deployment procedures, all the microservices share a common pipeline structure.

Our plan solution has its bases on the most common CI/CD practises: faster release cycles are indeed one of the major advantages of microservices architectures and are considered to be the best way to meet the stakeholders' favour.

5.2 Planning

Due to the relatively long estimated development times (in order of months), we decided to proceed with implementation with an incremental approach which could provide stakeholders with multiple incremental but stable versions of the software. In particular, our strategy is based on a feature by feature development which requires the parallel development of eMSP and CPMS in order to deliver the whole functioning system to the stakeholders.

Considering the priorities previously mentioned and the level of microservices inter-communication, we provide the following plan.

Each cycle includes the participation of both eMSP and CPMS features and is arranged in order to require approximately the same amount of time and resources. All the listed microservices can be developed in parallel or sequentially, depending on the availability of teams.

Cycle	eMSP microservices	CPMS microservices
1	Booking Management Service	Booking Service Charging Management Service
2	Charging Station Service	Charging Station Status Service Pricing Service
3	Customer Service Billing Service	Employee Service Energy Sources Management Service Billing Service
4	Cpo Catalog Service	Emsp Catalog Service Charging Station Configuration Service Dso Catalog Service Energy Sources Analytical Service

In the first three cycles, the eMSP mobile app interfaces and the CPMS web app interfaces can be developed in parallel with the development of the listed microservices, as the following table shows.

Cycle	eMSP mobile app interfaces	CPMS web app interfaces
1	Booking list page Book charge in advance page Book charge on the fly page	Charging station list page
2	Find stations page	Charging station status page Charging station pricing page
3	Login page Signup page Profile settings page	Charging station energy sources page

The fourth cycle could be used by the front end development teams to develop the interfaces required by eMSP admins to configure partnered CPOs and the ones required by CPO admins to configure partnered eMSPs, DSOs and charging stations.

For each microservice, the following phases are required to be performed by the assigned teams:

- development, possibly in parallel, of the back-end business logic and the corresponding front-end modules;
- unit testing, which is held in parallel with the development;
- component testing, in which the microservice is integrated with its own persistence layer and the whole unit is treated as a black box to test the interface's behaviour: when needed, some drivers/stubs can be implemented;
- deployment into the staging environment;
- end-to-end testing, in which the corresponding front-end modules, if any, are integrated with the microservice's APIs.

After the development of each cycle, the microservices are added progressively in the staging environment in the order established above, and the corresponding previously used drivers/stubs are removed.

6 EFFORT SPENT

Section	Total effort spent
Section 1 - Introduction	5 hours
Section 2 - Architectural design	80 hours
Section 3 - User interface design	25 hours
Section 4 - Requirements traceability	5 hours
Section 5 - Implementation, integration and test plan	5 hours

Team member’s contributions:

Brugnano Matilde 60 hours

Buttiglieri Giorgio Natale 60 hours

7 REFERENCES

1. “Assignment RDD AY 2022-2023_v3”
2. RASD
3. Deployments | Kubernetes
4. Service | Kubernetes
5. StatefulSets | Kubernetes
6. Persistent Volumes | Kubernetes
7. Apache ZooKeeper