Hochschule Konstanz
Fakultät Informatik

Parkspace - Team VW

Cloud Application Development – Winter Term 2024/25

Maren Franke - ma452fra - 313188
Elisha Leoncio - el871leo - 309900
Nico Riedlinger - ni911rie - 311667

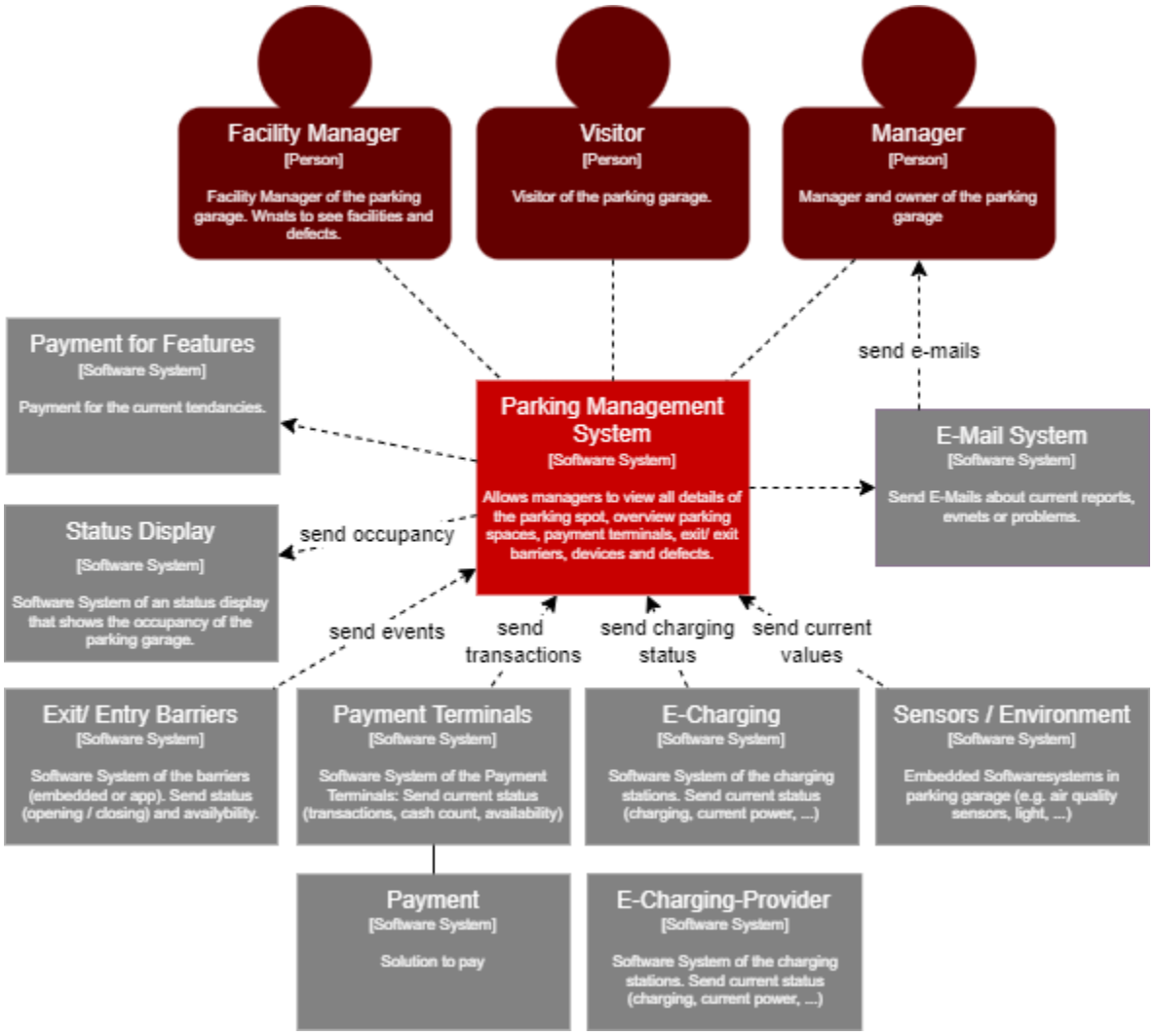
GitHub: https://github.com/msi-cad-vw

1 Requirements

Parkspace - an application to manage parking garages. From monitoring your own parking garage to facility management and detailed analyses of events in the parking garage, everything is possible for the parking garage manager. But not only for the owner of the garage this application brings advantages, but also for the end user, who can get an overview of the parking garages, see the current status of the garage via Occupancy Signs and get a better parking experience.

1.1 System Context

Shown below is the system context diagram of the application with external systems and users.



External Systems inside the parking garage:

- **Entry / Exit Barriers**: Software System of barriers, that send the parking-events and check if a visitor is allowed to drive in and out of the parking garage
- **Payment Terminals**: Terminals to pay the visit. They send requests to the API and validate the tickets. The Payment itself is fulfilled by an additional external Payment-Service.

- **E-Charging-Stations**: E-Charging-Stations, that are able to send start and stop events from the charging station, the payment and other requirements are provided by an external E-Charging service
- **Sensors**: Measurement systems that are able to send the current status.
- **Status Display**: Sign to show the current occupancy. Only requests the current status from time to time.

Other external systems

- **Payment solution**: System to fulfill online payments, like PayPal, …
- **E-Mail-System**: Send and receive E-Mails from and to customers to generate a knowledge transfer (e.g. to notify creation of new tenants, reports problems, …)

1.2 Use Case Overview

Various use cases and the stakeholders involved are described below. There are two different Use-Case-Diagrams provided. One for the system context with the different actors and one for the external components of the parking garage, that need to communicate with the system.

1.2.1. Actors

Manager = Owner and Manager of the parking garage and needs:

- Good Overview of the parking garage
- Manage parking garages (Create, Delete, Configure)
- See current status of parking garage and other reports
- User-Management to add other users to parking garage

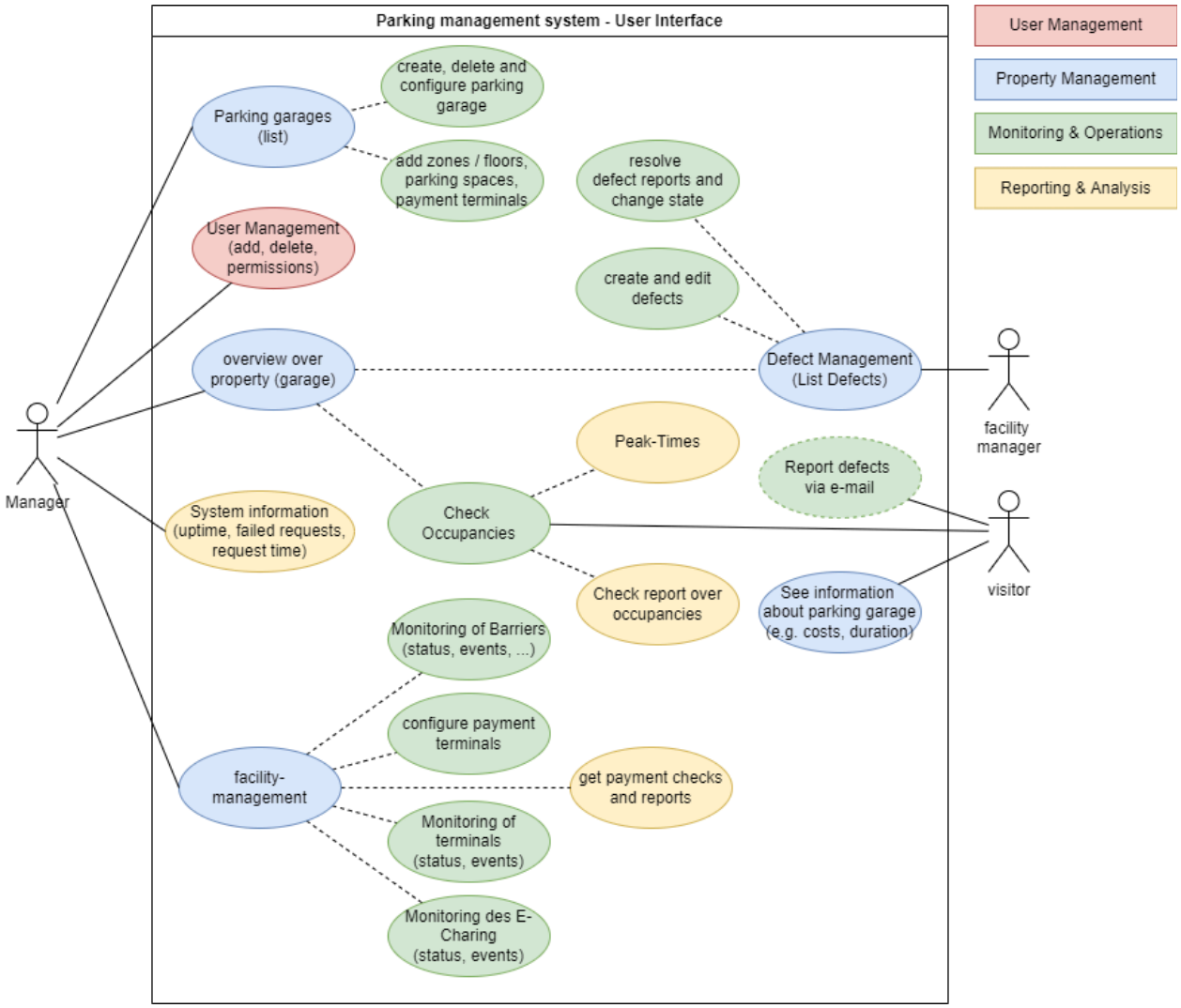
Facility Manager = Facility Manager of a parking garage and needs:

- Add, resolve and configure Defects
- Get overview over existing defects

Visitor = not logged in visitor of the parking garage and needs:

- Information about occupancy of parking garage
- See available parking garages

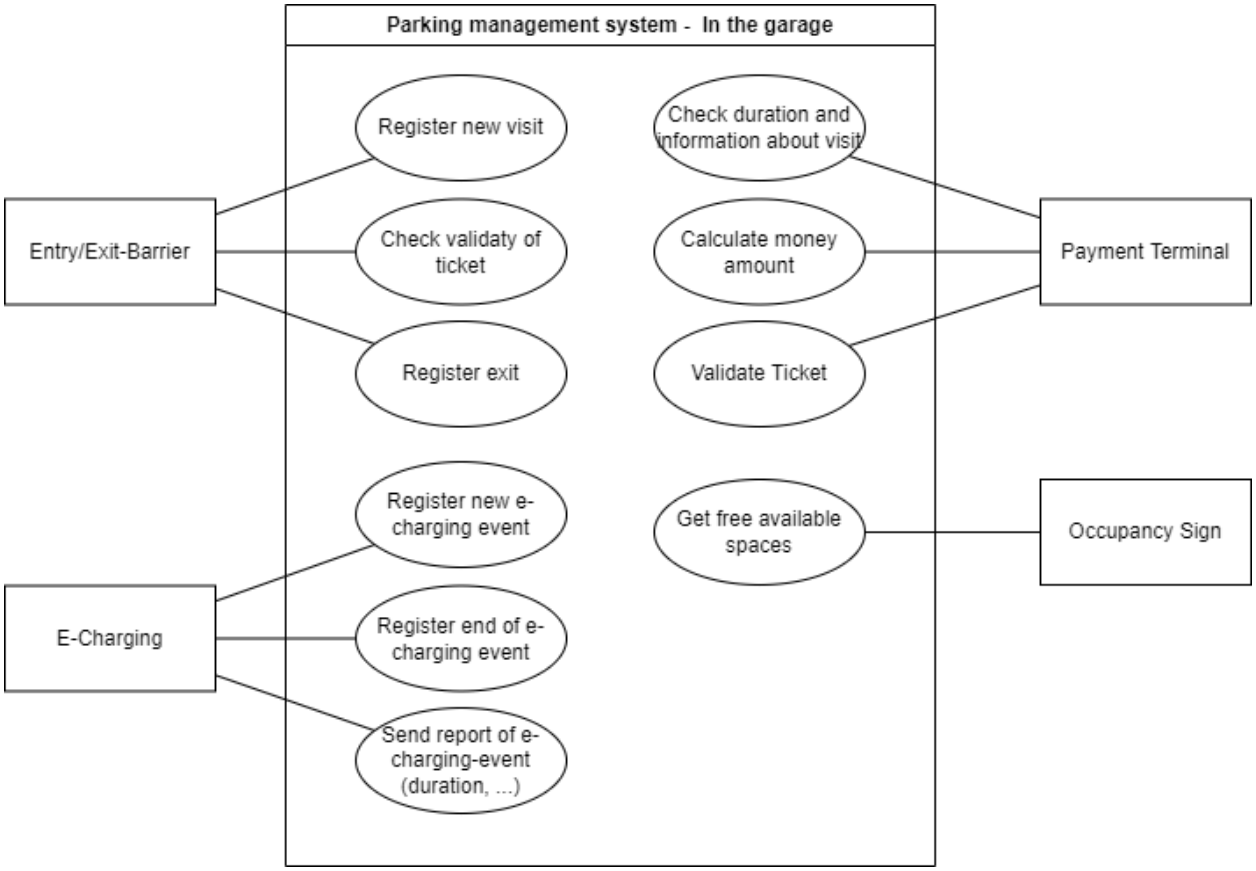
1.2.2 Use Case Application



1. As Manager, I want to get an overview of my parking garages. I want to be able to see all of them, add new parking garages to the current management and add parking spaces and similar things to the parking garage.
2. As Manager, I want to handle users of my parking management system. I want to add new ones and grant permissions.
3. As Manager, I want to get a good overview of all my properties. Additionally, I want to see if there are defects.
4. As Manager, I want to get information about the system (e.g. uptime, failed requests, request time).
5. As Facility Manager, I want to get an overview of the defects and resolve them.
6. As Manager, I want to see all parking spaces, the current occupancy (including peak times and reports) and reserve, manage or configure the spaces.
7. As Manager, I want to get an overview of the exit / entry barriers and see the current status and usage amount.

8. As Manager, I want to see my payment terminals, their status and of course the earned money. Additionally, I want to configure my terminals.
9. As Manager, I want to see the E-Charging terminal list and the current status.
10. As a Visitor, I want to see the costs of the parking garage and the current occupancy.
11. As a Visitor, I want to be able to report defects.
12. As a logged-in Visitor, I want to reserve parking spots.

1.2.3. Use Case - External Systems



In this case, the visitor wants to do the following things:

1. As a visitor, I want to be able to enter the garage and receive a ticket from the entry barrier. Therefore, my visit needs to be stored within the parking management service.
2. As a visitor, I want to be able to pay for my visit. Therefore, the spent time and the resulting costs need to be calculated, I have to pay, and the result will be stored in the database.
3. As a visitor, I want to leave the parking garage after my visit. Therefore, my ticket needs to be checked.
4. As a visitor, I want to see the current occupancy of the parking garage on a display.

1.2.4 Additional Requirements

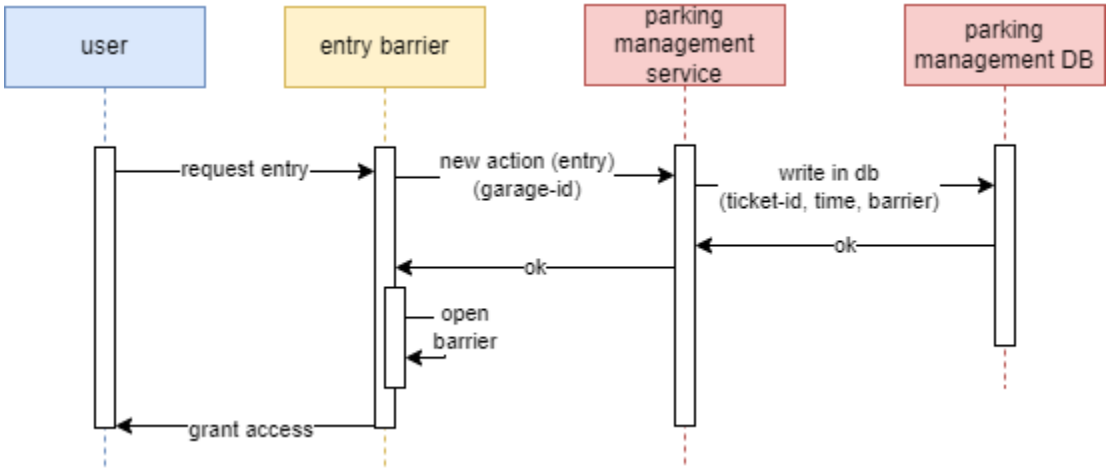
1. Authentication and Authorization
2. Self-Registration of Tenants with tenant management

3. Customization for each Tenant
4. Pricing models for the tenants
5. Security with IAM (e.g. one service account per tenant/service)
6. CI/CD-Pipeline
7. Separate Staging and productive environment

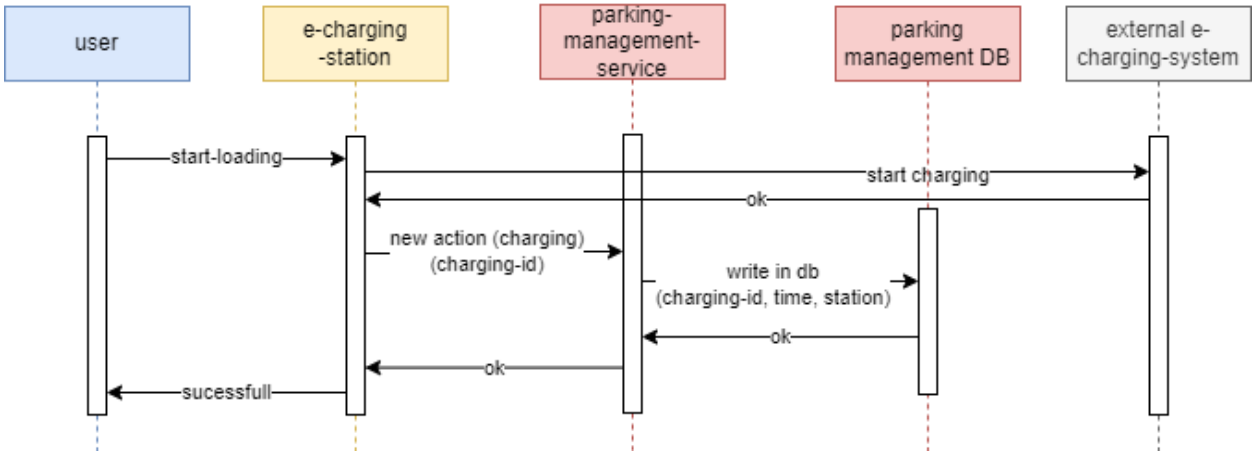
1.3 Sequence Diagrams

1.3.1 Synchronous Workflows

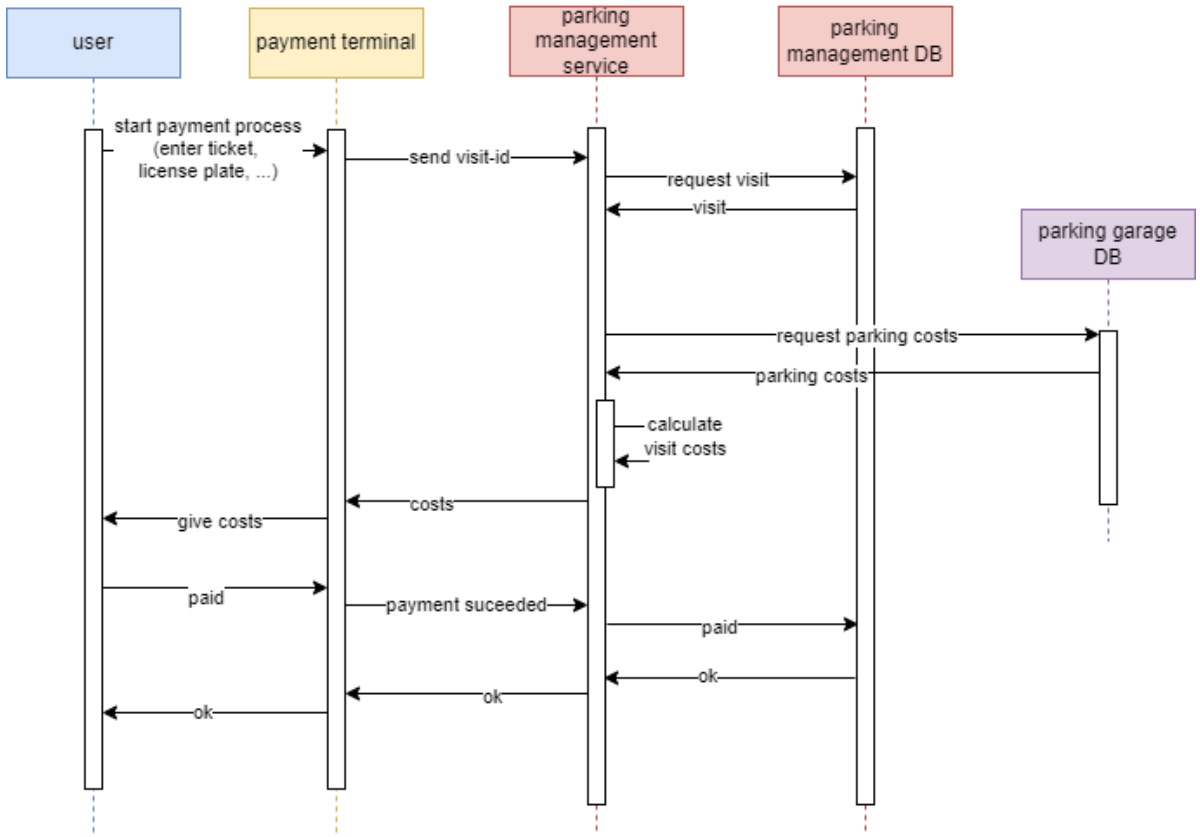
Entry of visitor (Drive through entry barrier)



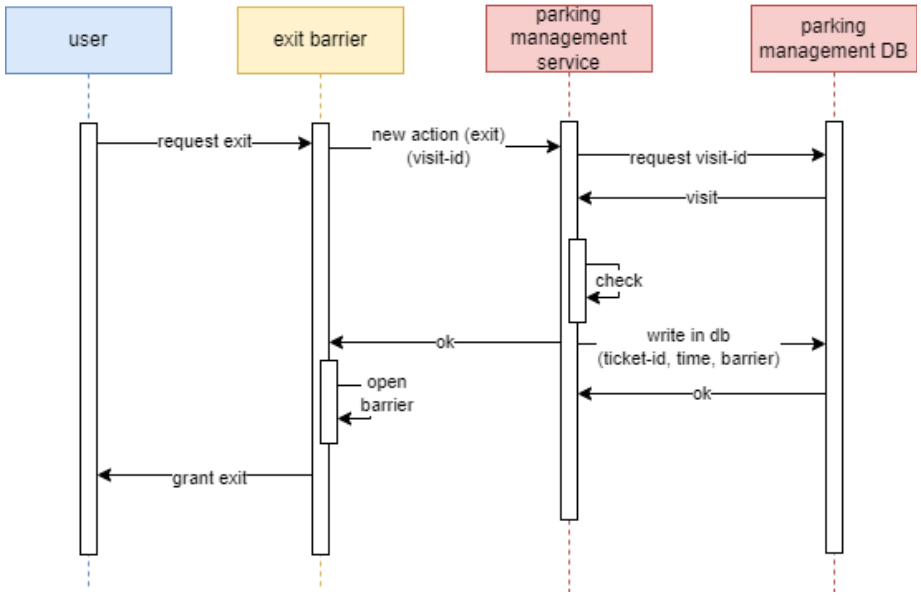
E-Charging of user (Start and Stop E-Charging)



Payment of visitor (Use the payment terminal)

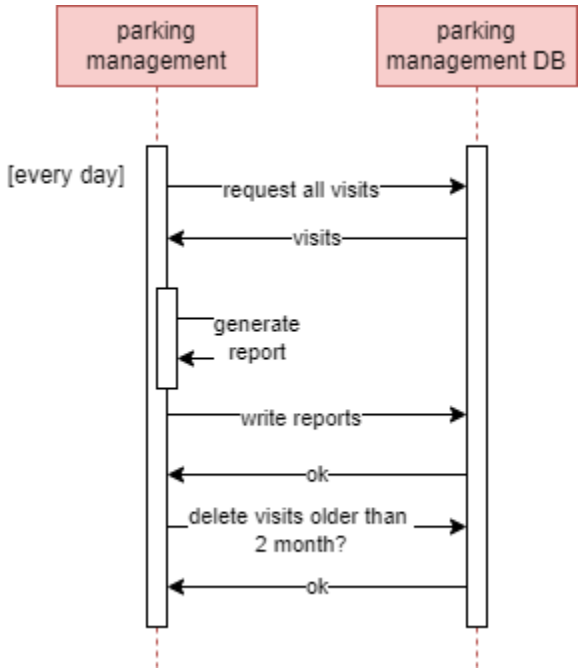


Exit of visitor (Use the exit barrier)

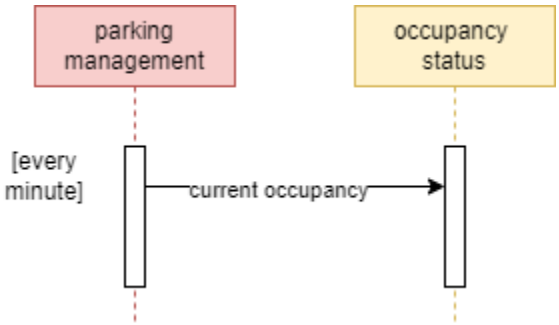


1.3.1 Asynchronous Workflows

Create report (Once per day)



Update Occupancy Sign



2 Development View

2.1 Software Components

2.1.1. Source Code

We use a Multi-Repo approach for the organization of source code repositories. Each repository serves a single microservice. Since all repositories are located at GitHub, they use Git as their Version Control System (VCS). There are the following repositories.

- **Backend Repositories** (written in C#)
  - Facility-Management
  - Defect-Management
  - Parking-Garages
  - Parking-Management
  - User-Tenant-Management
- **Frontend-Repository** (written in ReactJS)
- **CI/CD**
  - DevOps
  - Workflow
- **Other**
  - Backend-Microservice-Template-Repository
  - .github

All Backend Repositories are implemented using C# and the ASP.NET web framework. They contain logic for the microservices.

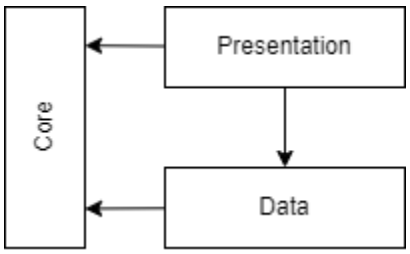
The Frontend repository acts as a singleton and contains all frontend logic. It is a ReactJS application and calls the API endpoints of the respective backend services for data.

All CI/CD-related logic and data is stored in the DevOps and Workflow repositories. The DevOps repository handles the management of the whole infrastructure using Terraform setup scripts. Furthermore, management pipelines as well as all Helm charts to set up the containers are included here.

There are also some other repositories. For easier creation of new backend microservices, a template repository is used. With this, a new microservice can easily be set up with all default configuration within a few clicks on GitHub. The *.github* repository is the default repository for documentation in a GitHub Organization. Thus, we used this for general documentation purposes, like the whole system architecture, as well as setup scripts.

2.1.2. Programming Languages

Backend - C#
The Microservices are implemented using C# and the ASP.NET web framework. To reduce dependencies between classes inside the services, a layered system structure is used. Every microservice has three layers, as depicted on the right-hand side.



The Data layer is the lowest. This bundles the data model as well as data requests to other software components via for example REST requests. Top-most of the structure is the Presentation, which also handles all incoming requests from the API in controllers and processes the business logic. A separate business logic layer was not used to reduce the complexity of each microservice.

In order for the layered system to work, one layer may only call functions from the layer directly beneath. A call to another layer may never be made across more than one layer. Vice versa, a layer may also only accept function calls from the layer directly above itself.

The only exception to this system is the Core layer, which takes over a special role compared to the other layers. Because it only contains general functions, enums and extension methods, as well as configuration options, it is detached from the layer stack and can be accessed from all layers. It is important that the Core layer does not have any dependencies to other layers, but instead is complete in itself.

Other libraries used in the backend include the various Google Cloud packages for features like Cloud Object Storage and FirestoreDB. These had to be used to connect to the database and populate it with data.

Another external library used is the Magick.NET library for image size scaling. It was used to down-scale images associated with defects inside the defect management microservice in order to save on storage capacity.

Frontend - JS

The Frontend is built using JavaScript. It serves as the foundation for building dynamic and interactive user interfaces for the project. To structure the application, React.js is used. React.js is a JavaScript library for building component-based, reusable, and efficient user interfaces. The modular architecture of React promotes code reusability and simplifies the management of complex UI states. Each UI element or feature is encapsulated in its own component, making the codebase easy to maintain and extend.

For styling, an extension of CSS, called SCSS, is used. SCSS brings additional features like variables, nesting and mix-ins, which simplify styling and enable better code organization. In keeping with best practices, SCSS files are maintained separately from the JavaScript files. This separation ensures cleaner code and facilitates easier collaboration within a team. Each SCSS file corresponds to a specific component or feature, maintaining modularity in both functionality and design.

Additionally, React's ecosystem, combined with JavaScript, allows for a seamless integration with other libraries and tools, enhancing the overall development experience. Some of the libraries used in the project include:

- **React and React DOM:** Forming the core foundation of the project, enabling the creation and rendering of component-based user interfaces.
- **React Router DOM**: For seamless client-side routing and navigation between different views or pages.
- **Chart.js**: To create interactive and visually appealing charts for data visualization.
- **React Color**: To integrate a customizable color picker, enhancing user experience for selecting colors dynamically.

- **React Icons**: Providing a comprehensive collection of icons to enhance the visual appeal of the user interface.
- **Sass**: For writing modular, reusable, and maintainable styles using SCSS.

These tools and libraries work together to ensure the project is scalable, feature-rich and maintainable while delivering a polished user experience.

2.1.3 External Interfaces

The system deals with different external applications. As already shown in the use case diagram, two types of external systems are used - the systems inside the parking garage such as payment terminals, entry/exit barriers and E-Charging terminals, and the external systems, such as an e-mail system.

The external systems inside the parking garage can be connected via the various APIs provided by the application. Each terminal, barrier and station must send its current status to the API endpoint. The external endpoints for the different services are used as a showcase on the "Simulation Page" provided by the front end. This page simulates the different situations such as "drive in", "pay", "drive out" or "start/stop charging". With this information, the parking garage equipment can be adapted to send the requests to our system and start the integration of the current parking garage into the new application.

The external systems outside the parking garage, such as the e-mail system or the payment service, are used passively in our application. Each user can write to a designated e-mail address in case of errors in the system. In addition, each organization can provide an e-mail address that can be contacted in case of errors or other problems in the parking garage.

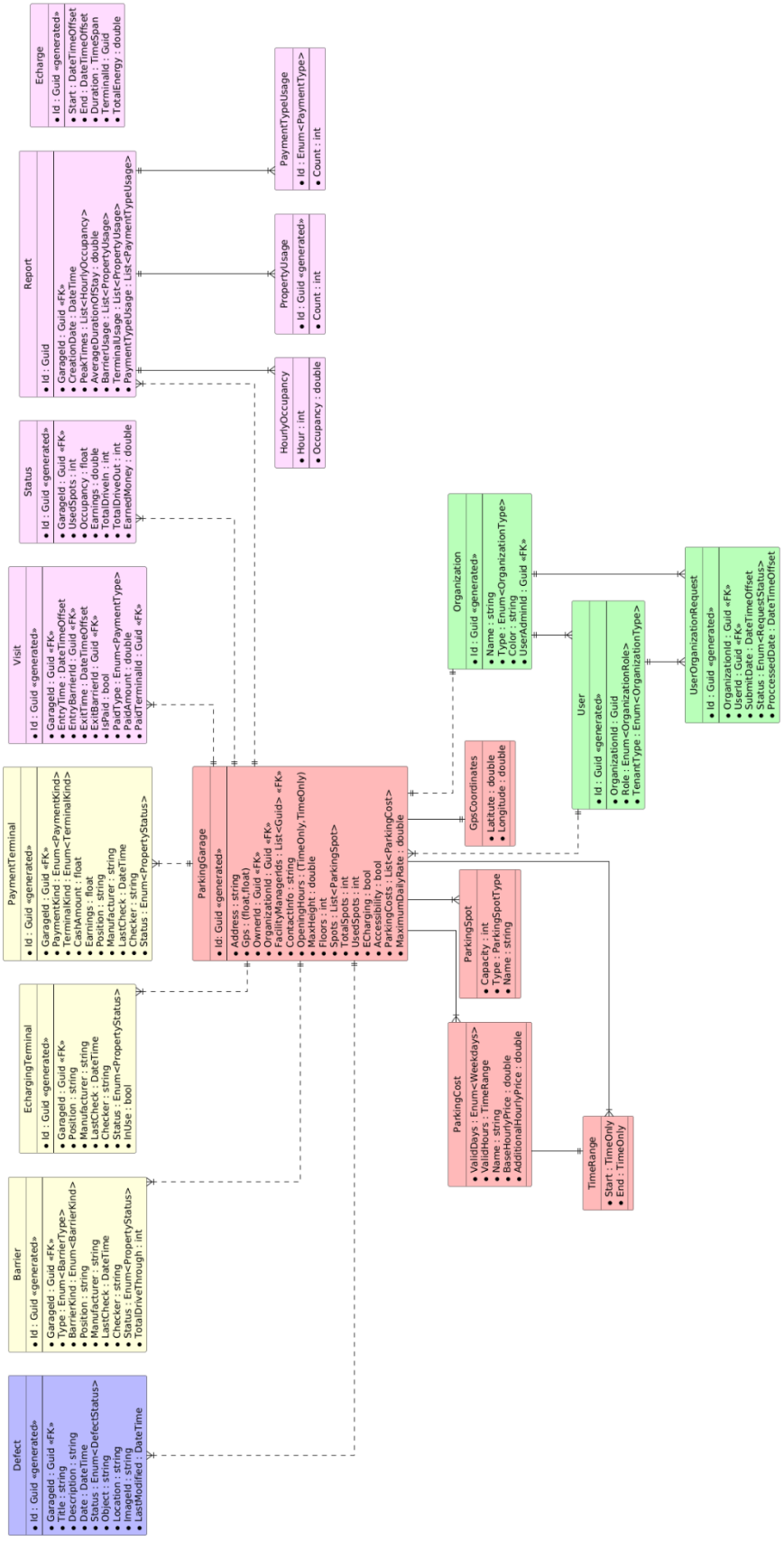
The payment system is currently a SEPA mandate, which can be extended.

2.2 Data Model

All alphanumeric data is stored inside FirestoreDB. Despite having a document-oriented NoSQL database, we defined a fixed database schema, as can be seen in the image on the next page.

Entities with the same color are inside the same microservice. Relationships inside a microservice are marked with a solid line, while relationships between different microservices are marked with a dashed line for easier reading.

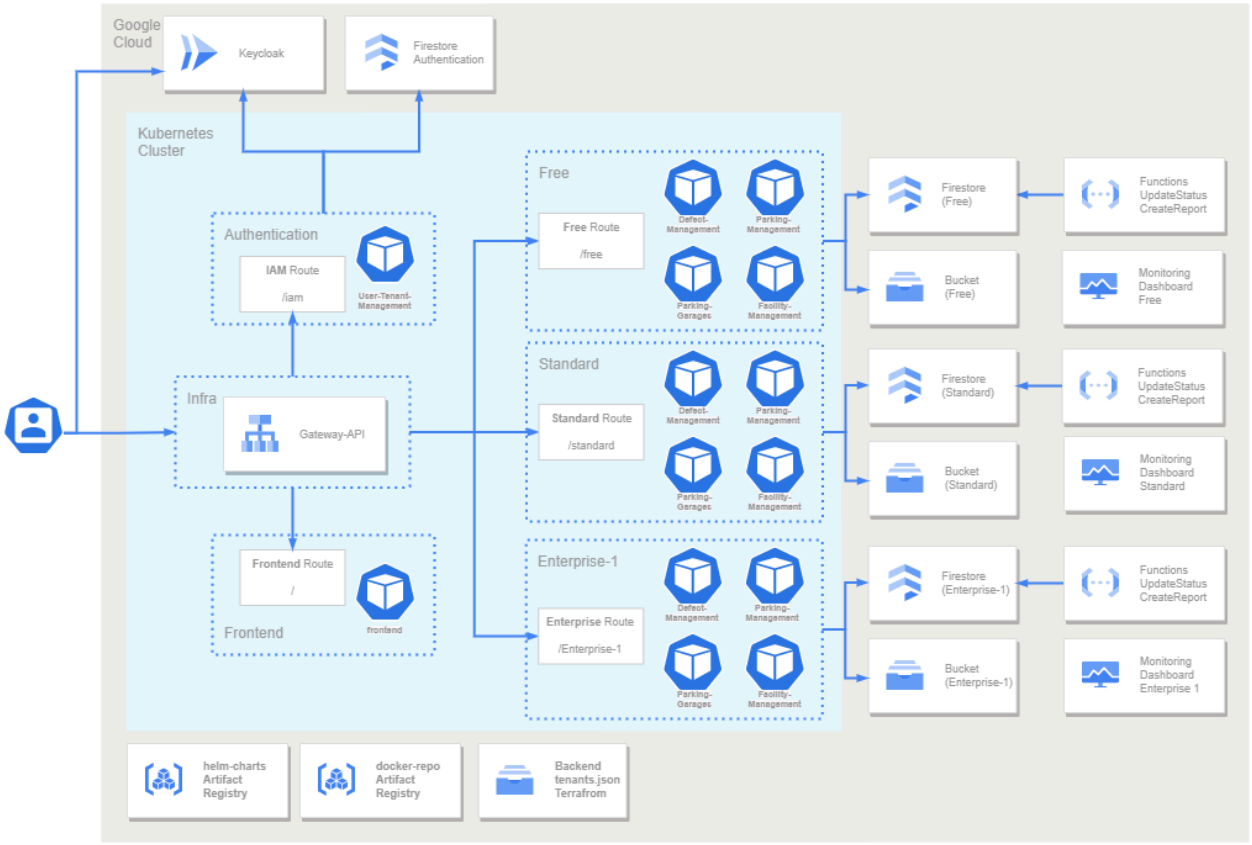
The schema does not follow a strict relational database setup, but provides more of a baseline for orientation. The “foreign keys” are highlighting a link between two objects, but are rather weak links, since no cross-validation is done between the objects. The foreign key relationships are marked with a dashed line.


Echarge
• Id : Guid «generated»
• Start : DateTimeOffset
• End : DateTimeOffset
• Duration : TimeSpan
• TerminalId : Guid
• TotalEnergy : double
Report
• Id : Guid
• GarageId : Guid «FK»
• CreationDate : DateTime
• HourlyOccupancy : List<HourlyOccupancy>
• AverageDurationOfStay : double
• BarrierUsage : List<PropertyUsage>
• TerminalUsage : List<PropertyUsage>
• PaymentTypeUsage : List<PaymentTypeUsage>
PaymentTypeUsage
• Id : Enum<PaymentType>
• Count : int
PropertyUsage
• Id : Guid «generated»
• Count : int
HourlyOccupancy
• Hour : int
• Occupancy : double
Status
• Id : Guid «generated»
• GarageId : Guid «FK»
• UsedSpots : int
• Occupancy : float
• EntryCount : int
• Earnings : double
• TotalDriveIn : int
• TotalDriveOut : int
• EarnedMoney : double
Visit
• Id : Guid «generated»
• GarageId : Guid «FK»
• EntryTime : DateTimeOffset
• EntryBarrierId : Guid «FK»
• ExitTime : DateTimeOffset
• ExitBarrierId : Guid «FK»
• IsPaid : bool
• PaidType : Enum<PaymentType>
• PaidAmount : double
• PaidTerminalId : Guid «FK»
Organization
• Id : Guid «generated»
• Name : string
• Type : Enum<OrganizationType>
• Color : string
• UserAdminId : Guid «FK»
UserOrganizationRequest
• Id : Guid «generated»
• OrganizationId : Guid «FK»
• UserId : Guid «FK»
• SubmitDate : DateTimeOffset
• Status : Enum<RequestStatus>
• ProccessedDate : DateTimeOffset
User
• Id : Guid «generated»
• OrganizationId : Guid
• Role : Enum<OrganizationRole>
• TenantType : Enum<OrganizationType>
PaymentTerminal
• Id : Guid «generated»
• GarageId : Guid «FK»
• PaymentKind : Enum<PaymentKind>
• TerminalKind : Enum<TerminalKind>
• CashAmount : float
• Earnings : float
• Position : string
• ManufacturerId : string
• LastCheck : DateTime
• Checker : string
• Status : Enum<PropertyStatus>
ParkingGarage
• Id: Guid «generated»
• Address : string
• Gps : (float,float)
• OwnerId : Guid «FK»
• OrganizationId : Guid «FK»
• FacilityManagerIds : List<Guid> «FK»
• ContactInfo : string
• OpeningHours : (TimeOnly,TimeOnly)
• MaxHeight : double
• Floors : int
• Spots : List<ParkingSpot>
• TotalSpots : int
• UsedSpots : int
• ECharging : bool
• Accessibility : bool
• ParkingCosts : List<ParkingCost>
• MaximumDailyRate : double
GpsCoordinates
• Latitute : double
• Longitude : double
ParkingSpot
• Capacity : int
• Type : ParkingSpotType
• Name : string
EchargingTerminal
• Id : Guid «generated»
• GarageId : Guid «FK»
• Position : string
• ManufacturerId : string
• LastCheck : DateTime
• Checker : string
• Status : Enum<PropertyStatus>
• InUse : bool
ParkingCost
• ValidDays : Enum<Weekdays>
• ValidHours : TimeRange
• Name : string
• BaseHourlyPrice : double
• AdditionalHourlyPrice : double
TimeRange
• Start : TimeOnly
• End : TimeOnly
Barrier
• Id : Guid «generated»
• GarageId : Guid «FK»
• Type : Enum<BarrierType>
• BarrierKind : Enum<BarrierKind>
• Position : string
• ManufacturerId : string
• LastCheck : DateTime
• Checker : string
• Status : Enum<PropertyStatus>
• TotalDriveThrough : int
Defect
• Id : Guid «generated»
• GarageId : Guid «FK»
• Title : string
• Description : string
• Date : DateTime
• Status : Enum<DefectStatus>
• Object : string
• Location : string
• ImageId : string
• LastModified : DateTime

3 Runtime View

In the runtime environment, there are a lot of components running.

The core of everything is built by the Kubernetes cluster, which contains the APIs, user authentication and the frontend. Additionally, there are a lot of other Google components, which are used in the application.



The interface to the user is the Gateway-API, which forwards requests to the HTTP-Routes to the different namespaces and the required microservices.

There are a lot of different microservices, which will be described below. The services are managed by Kubernetes and each service will run in its own pod. The scaling is done by Kubernetes and a *HorizontalLoadBalancer*. Each microservice has a maximum CPU and memory limit. The configuration for the scaling and runtime configuration is described in chapter 4.

The services are only accessible through an API-Gateway, which uses HTTPS with a self-signed certificate. To make a request to a microservice, authentication is required. Authentication and authorization follows the OpenID Connect flow where users authenticate to a central IAM system which in turn authorizes users upon successful login. To log in, users must enter a username and password. The passwords are encrypted and stored securely inside the Keycloak database. The benefit of using Keycloak over implementing our own authentication/authorization flow is that Keycloak already implements all required flows securely while also being open-source software.

A separate isolated namespace is created for each tenant. This namespace is defined via the routes as follows:

```
https://parkspace.tech/<tenant-id>/<service-name>/<path>
```

This address is forwarded via the gateway API, and it can be ensured that all requests only reach the endpoint that they are supposed to reach. Separate resources are also created for each tenant, e.g. the databases, buckets, functions and dashboards are created for each tenant.

More details are described in chapter 4.

3.1 Keycloak

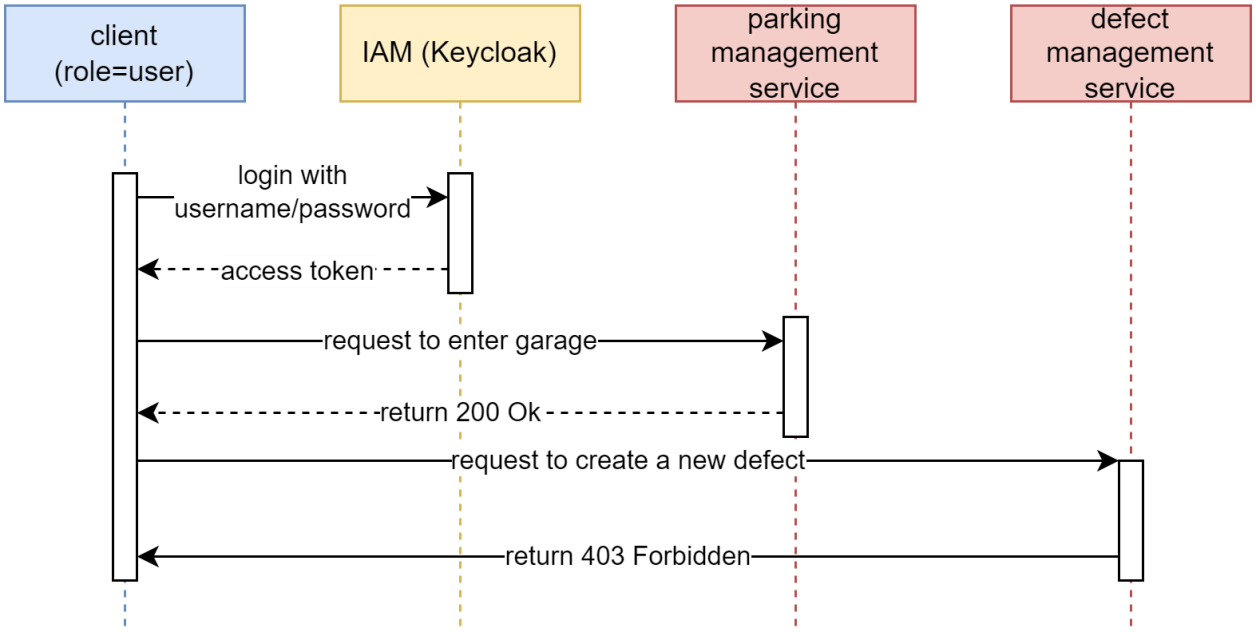
Keycloak is being used as the Identity and Access Management (IAM) system and as such handles all authentication and authorization processes. It is deployed as a container in Google Cloud Run outside the cluster. To save on costs for development, an Object Storage bucket is used in combination with a local database file. Similar to *SQLite*, Keycloak uses its own database system called “H2” for file database storage. Before going into production, this setup should be revised and a real SQL-Server should be used, as the approach described above also limits the Keycloak instances to exactly one, thereby effectively eliminating scaling possibilities. Using a real SQL-Server, Keycloak can be set up redundantly with multiple instances for fail-over and load balancing.

The Keycloak service is set up as a singleton. This means that it is not reproduced among different tenant types, but instead all users connect to the one service and all user data is stored in a so-called “realm”. This realm stores all data associated with a user, as well as their roles and privileges. Using only a single realm makes tenant migration easier, as only a user’s group must be migrated instead of the whole user.

All communication from the backend to Keycloak is handled by the User-Tenant-Management service (see section 3.2.5). To access data inside Keycloak, the official Keycloak REST API is used. A service account is set up with the necessary roles to automatically change users groups and roles.

3.1.1 Authorization flow

The client sends its user data to Keycloak for login, logout, etc. Passwords and sensitive data are stored securely in Keycloak which makes for easy authentication processes regarding privacy and data regulation. All workflows are oriented on the default OpenID Connect flow, which provides the frontend service with an access token containing all data and roles required to authenticate and authorize the user for different services. This token must be sent in the Authorization header in every request for the backend. The complete flow between client, IAM and backend services is visualized in the following sequence diagram.



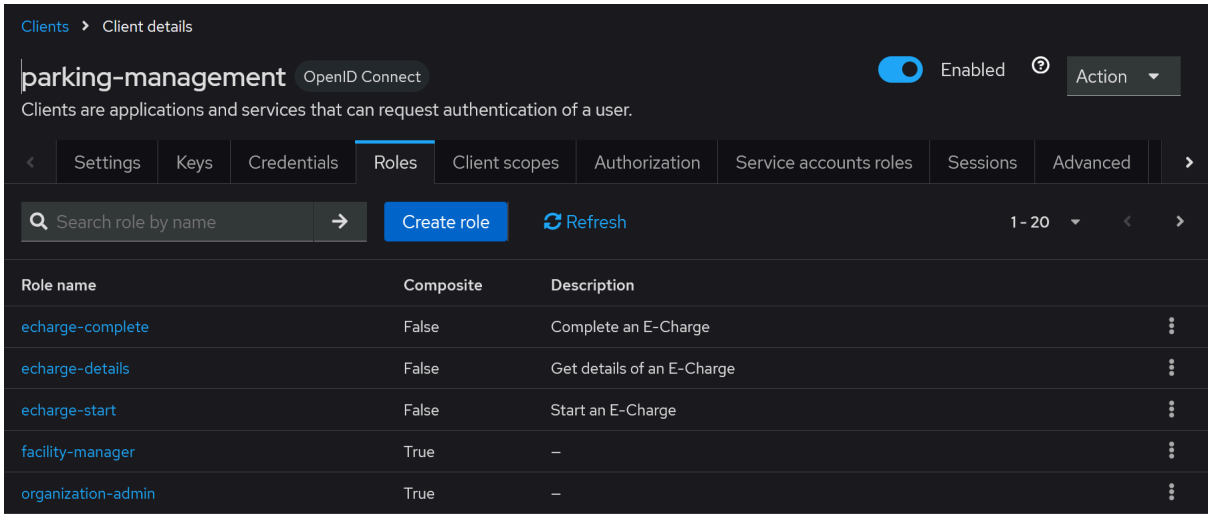
The client has a role of “user”. Regular users may enter a parking garage by requesting the *ParkingManagement* service. This can be seen as the request returns an HTTP status code of “200 OK” as a result. In contrast, a regular user is not allowed to create a new defect. Thus, the second request to the *DefectManagement* service is denied by returning a status code of “403 Forbidden”. In both requests, the acquired access token from the first step must be included in the Authorization header. Otherwise, if no access token is provided, each backend service will return a status code of “401 Unauthorized” indicating that a user must log in before.

3.1.2 Clients

To access Keycloak from any backend service, an individual *client* is created. This client has the same name as the microservice it belongs to. Every backend microservice gets its own client to enforce data separation between the services. Also, this way, a too powerful single client can be avoided in terms of security.

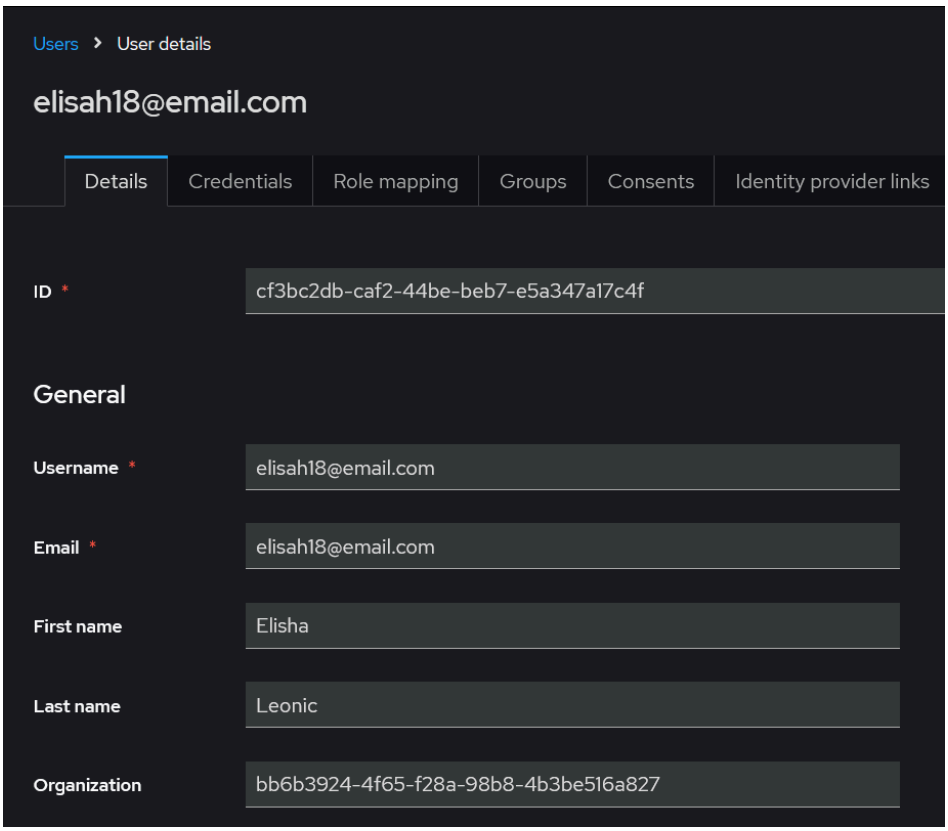
Authorization is done via the client ID (aka name) and a secret. The latter is an auto-generated, alphanumeric string of 32 characters length. Services can use the combination of these two values to authorize the Keycloak client and get information about a user’s role inside the client. The implementation of this connection in the backend services can be seen in section 3.2.1.

Inside a client, all roles for the endpoints are created. Additionally, they are grouped together as the different roles an organization member can have, namely “user”, “facility-manager” and “organization-admin”. For this, composite roles are used. They are just like normal roles, except that they also reference other roles. A user that has a composite role will automatically have all referenced roles. Part of the roles overview of the *parking-management* client can be seen in the screenshot below.

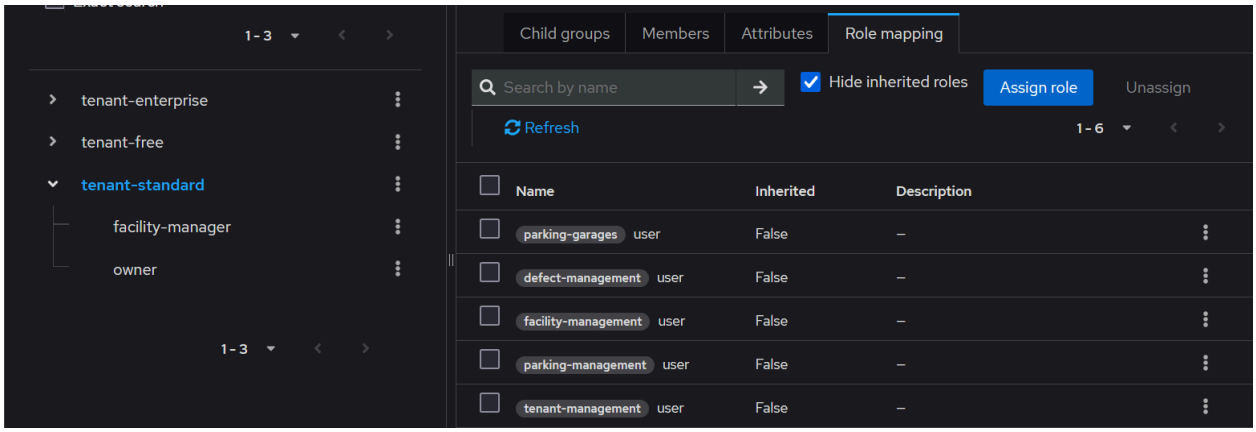


3.1.3 Users and Groups

All users are created in the realm "*parking-management*", no matter what tenant type they belong to, because the tenant type will be determined by the organization they belong to. A user may also switch organizations as they want. Affiliation to an organization is marked with a custom attribute in the user model. This can also be seen in the image below. The unique identifier of the corresponding organization is entered in the organization field.



Besides users, groups are also set up. A group can bundle multiple users into a logically connected unit. Furthermore, a group can also have roles attached. This will extend any group roles to all users that belong to it. An example can be seen in the image below with the regular “user” group for the Standard tenant type. It is also visible that only the respective “user” composite role of each client is mapped to the group in order to keep maintenance at a minimum.



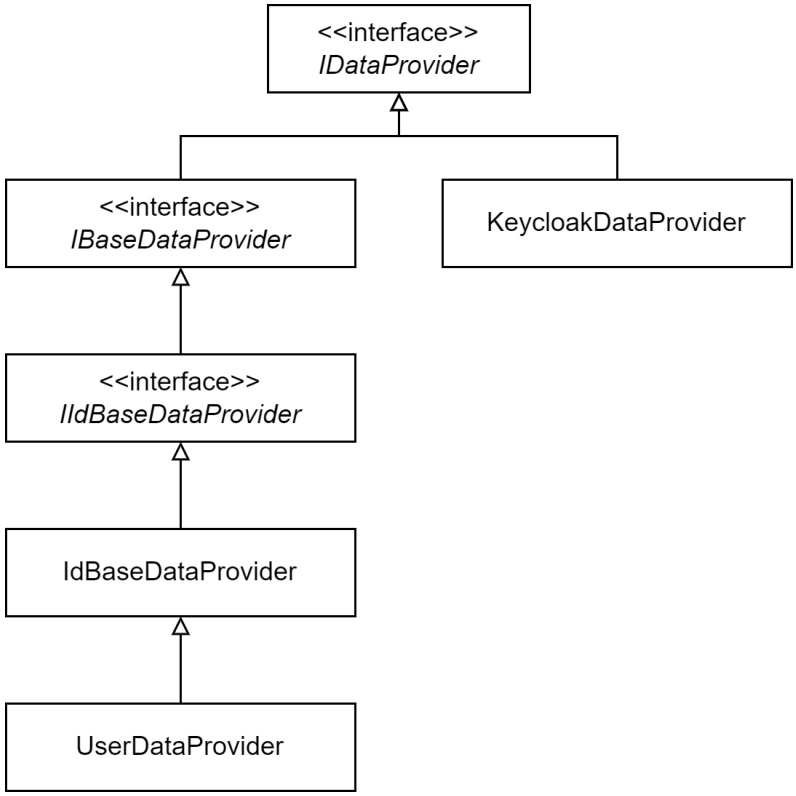
To switch a user’s role in an organization or add or remove them entirely, they only have to be added or removed from the respective group. This makes controlling the roles with the REST API fairly easy as only group memberships of a user must be changed in order for them to get a new role.

3.2. Backend Services

All backend services are implemented in C# / .NET and follow a common structure. This structure is described in section 3.2.1. The following sections only describe individual specialties of the microservices that differ from the baseline.

3.2.1 General Structure

The database used is Google FirestoreDB. It is a NoSQL document database. For data access from the microservices to the database, the official NuGet package by Google is used in the *Data* layer of each service. *NuGet* is the package manager in the C# / .NET ecosystem and functions like, for example, *NPM* in NodeJS. To make development easier, an abstraction was implemented and applied to all backend microservices. This way, *Data Provider* classes can easily be set up by inheriting the class *IdBaseDataProvider*. This inheritance structure is depicted in the image below.



Furthermore, this inheritance structure allows for an automated Dependency Injection setup. This is a concept used in C# to invert control of object instances and avoid circular dependencies, especially in bigger projects with many cross-references. With Dependency Injection, objects in constructor parameters must only be requested instead of created individually. They will be injected into the class constructor automatically by an instance sitting above all and coordinating all object instances.

Using the top-most interface `IDataProvider`, each data provider class can be registered automatically. The left column describes data providers, specifically for database entities with a unique identifier. In combination with the `IdBaseDataProvider`, a general setup for basic CRUD operations to the database is also given and must not be implemented for each data provider again. An example for this type of data provider is the `UserDataProvider` which manages all user-related operations and data accesses to the database. The right column visualizes any other data providers that do not handle database entities, but can still be registered for injection. An example for this is the `KeycloakDataProvider` which handles all communication with the Keycloak REST API.

Each service exposes a number of different endpoints from the controllers. All endpoints are secured with a custom role that a user must have to access the endpoint. This role is included in a bearer access token that must be sent along with the request inside the authorization header. To verify the token integrity, all required data is configured in the microservice options, like the correct issuer or realm name. These values come from Keycloak and must be set accordingly. Since this data is configured beforehand, no separate request must be made to Keycloak at runtime.

For a better overview of all endpoints of a service, a SwaggerUI is set up. This lists all available endpoints, as well as the required HTTP method to access it (like `GET` or `POST`) and input parameters. Furthermore, a brief description of the endpoint is provided for better usage and more context.

The microservice also provides multiple configuration options. These can be found in the *Presentation* layer inside the *appsettings.json* file. Amongst general settings for the service, like log message level and format, the two sections depicted below are the base configuration options for the service and are included in every service. They may, however, be extended in individual services, if needs be.

```
"Iam": {
  "Hostname": "https://msi-cad-vw-keycloak-768187275150.europe-west1.run.app",
  "Authority": "https://msi-cad-vw-keycloak-768187275150.europe-west1.run.app/
                realms/parking-management",
  "Realm": "parking-management",
  "ClientId": "parking-garages",
  "ClientSecret": "SfYuVu9gnIl5qUyCsEEm9CQyDumZm3XX"
},
"Options": {
  "ServiceBaseRoute": "parking-garages",
  "ProjectId": "msi-cad-vw-parking",
  "DatabaseId": "msi-cad-vw-database-standard",
  "SecretMountPath": "NONE"
}
```

The first section, `IAM`, provides settings about the IAM system. In addition to hostname and authority, the realm, client ID and secret are the most important settings in this section. Using the client id and its secret, the microservice can call Keycloak for further information about the user and request their roles.

The second section, `Options`, provides general settings for the microservice that are explained below.

- `ServiceBaseRoute`: A general prefix for all routes to endpoints. This way, all the microservices can be differentiated in the cluster.
- `ProjectId`: The Google Cloud project ID where all the Cloud services are run.
- `DatabaseId`: The ID of the Google Cloud Firestore database that the microservice should connect to. This way, different deployments of the service can use different databases, for example to separate Free, Standard and Enterprise tenant types.
- `SecretMountPath`: The path to a service account key that is mounted into the container for authentication.

All configuration options included in the *appsettings.json* file can also be set using environment variables. Therefore, the prefix `MSICAD_` must be used concatenated with the hierarchical name of the value and double underscores. For example, to set the value of `DatabaseId`, the environment variable must be named `MSICAD_Options__DatabaseId`. This way, configuration can be set dynamically at container start. This setup of the application options works across all backend microservices.

Credentials for Google Cloud services are handled with a key secret of a service account. This secret may be created in the web interface or automatically, i.e. using Terraform. It contains authentication and authorization data to use different cloud services. Google will automatically set an environment variable with a path to the secret when deployed in the Google Cloud. For local development or when deploying in a container, the secret must be set manually. This is the reason that the options include the value for `SecretMountPath`. At application startup, a small conditional clause is included to check the runtime environment and setting the environment variable, if needed. The code can be seen below.

```
if (Environment.IsEnvironment("Container"))
{
       System.Environment.SetEnvironmentVariable("GOOGLE_SERVICE_ACCOUNT_SECRET",
       Configuration.GetValue<string>("Options:SecretMountPath"));
}
```

In this case, the variable is set in the “Container” environment only. The environment is set during the build phase when all source code is compiled. The “Container” environment specifically is set when building the project using the Dockerfile. For local execution, the environment variable is set directly in Visual Studio. This approach was easier as every developer has the file at a different location on their computer. If it was included in the *appsettings.json* file and, thus, in the Git repository, it would have had to be changed every time another developer built it.

3.2.2 Parking Garages

The *ParkingGarages* microservice stores all information about parking garages. A *ParkingGarage* is the central object and stores general data about the garage.

In the database scheme from chapter 2, the red objects belong to this microservice. While there are many entities colored red, only the *ParkingGarage* object is stored in the database. All other objects are used for easier parsing and storing of corresponding data. At this point, we stray from a strictly relational data model. It is not normalized (1. Normalform) in a traditional sense. But, FirestoreDB being a document-oriented NoSQL database, even bigger objects like here can be stored inside it.

Despite the general structure and connection to FirestoreDB, the *ParkingGarages* microservice does not use any other external dependencies or interfaces.

3.2.3 Facility Management

The *FacilityManagement* microservice manages all the properties a parking garage can have. In our scenario, this includes entry and exit barriers, payment terminals and E-Charging terminals. They can be created, updated, deleted and read either one by one or all in one request. In the database scheme from chapter 2, all entities colored yellow belong to this microservice.

There is no further functionality to this microservice other than managing the properties with the FirestoreDB. Thus, it does not include any other external dependencies or interfaces.

3.2.4 Parking Management

The *ParkingManagement* microservice is for overseeing parking operations. This includes entering and exiting a parking garage, as well as paying the fee for a visit. In addition, it also keeps score of each parking garage’s status and stores daily operations reports of them. In the database scheme from chapter 2, all entities colored pink belong to the *ParkingManagement* service.

Besides FirestoreDB, the service uses Cloud Functions indirectly to create periodic data for the reports. However, since the Cloud Functions access the database directly, there is no implementation of them in any kind inside the service. More information about Cloud Functions can be found in section 3.5.

3.2.5 Defect Management

The *DefectManagement* microservice is for the management of defects. Each defect belongs to a parking garage, has some informational data and can have an image linked to it. This makes this service a little more special than the others, as storing images requires a connection to Google Cloud Object Storage buckets. The entities of this microservice are colored blue in the database scheme from chapter 2.

All textual data for a defect is stored in FirestoreDB. Any binary data is stored inside the aforementioned storage bucket with a unique identifier. Using this identifier, the data can be retrieved again. To make this step easier and access the data directly, we used signed URLs. This requires a *UrlSigner* to be set up in the service, which itself requires access to a service account that has the needed roles. In order to achieve this, we can reuse the service account key secret that was already used for the FirestoreDB connection.

To save on storage capacity, the library *Magick.NET* was used as a NuGet package. This allows us to scale images down if they exceed a certain pixel threshold. To make this threshold configurable, it is also included in the *appsettings.json* file as a new value. If either height or width of the given image exceeds the configured threshold, the image will be scaled down to the maximum dimension size while, at the same time, keeping the original aspect ratio.

Because this microservice has the bucket connection, it also has two new configuration values in the *appsettings.json* file that are described below.

- `FileUploadBucket`: The name of the Google Cloud Object Storage bucket that images should be saved in and retrieved from.
- `MaxImageSizePx`: The maximum pixel size that images may have in height or width before they are scaled down. Simultaneously functions as the target scaling size that images are scaled to.

3.2.6 User Tenant Management

The *UserTenantManagement* microservice manages all things related to users and their organizations. In addition, it also communicates with Keycloak and can automatically set different roles for a user. In the database scheme from chapter 2, all entities colored in green belong to this microservice.

This microservice primarily manages the relationship between users and organizations. Users may submit requests to join an organization, which will also be managed by this microservice. If the request is approved and the user joins an organization, it is set by this service, both in FirestoreDB and in Keycloak. Furthermore, this service is responsible for moving an organization to a new tenant type. To achieve that, all users must be migrated to the selected tenant type in Keycloak while keeping their respective roles inside the organization.

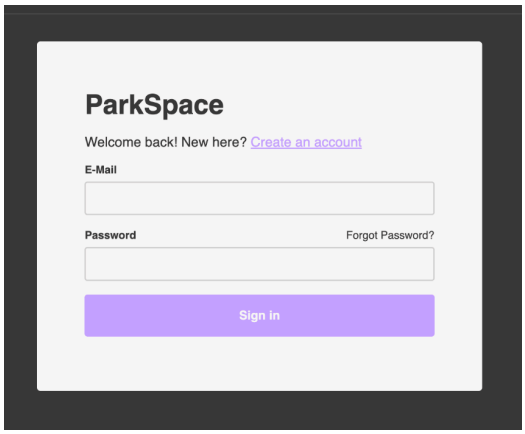
To access and change data of Keycloak, the official Keycloak REST API is used. It requires either a user or a service account for access. Based on the chosen way of access, authorization is done and the API call handled. Since microservices should never know of user passwords, and especially not admin passwords, a service account is set up for the respective client. This service account is granted the required roles for user management inside Keycloak. Only the required roles are granted, not all available, to follow the principle of least privilege.

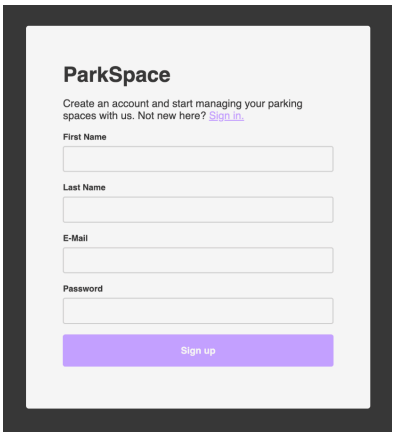
For easier usage of the Keycloak REST API, the NuGet package *Schick.Keycloak.RestApiClient* is used. It abstracts each endpoint of the API and wraps it inside a C# method that can be called. Additionally, it also creates object classes to serialize or deserialize JSON results or requests for the API. This way, the API can be consumed without much effort or time-intensive programming of REST calls. It is used in the *Data* layer of the microservice.

3.3 Frontend

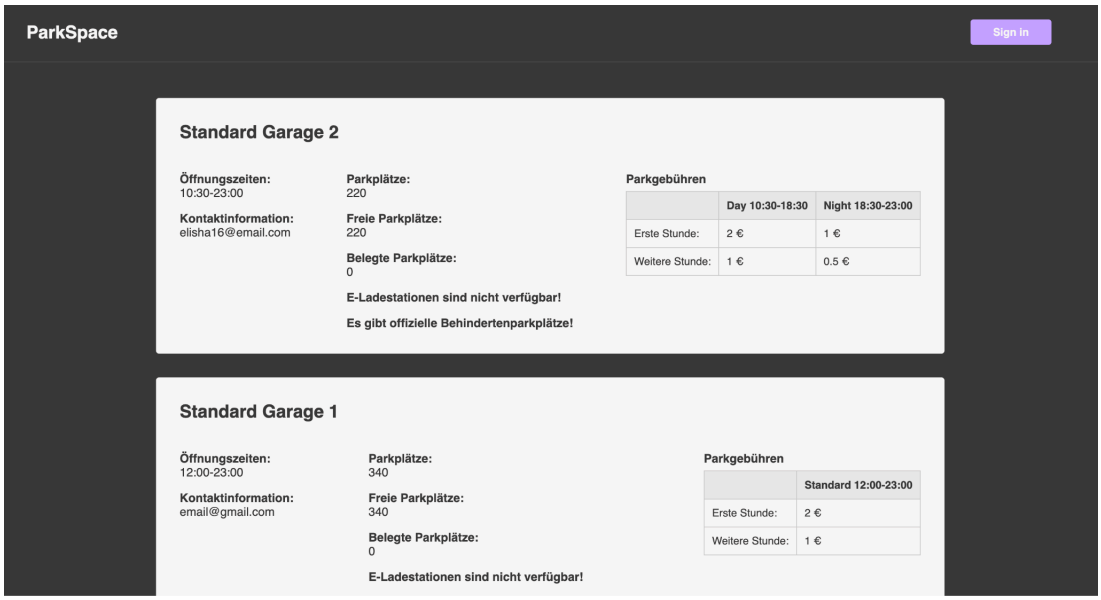
The microservice *frontend* includes all the files needed to build the user interface. Users can navigate through several pages:

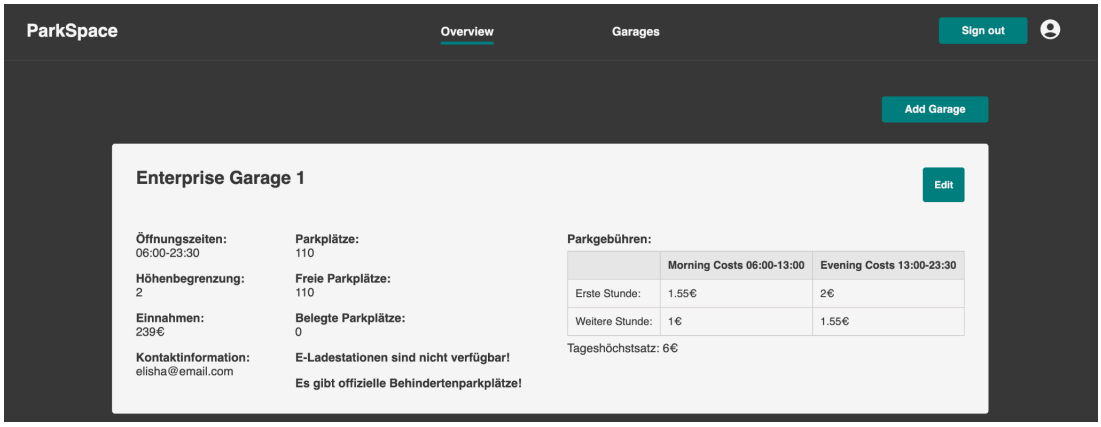
- **Login Page** and **Register Page** for authentication and account creation



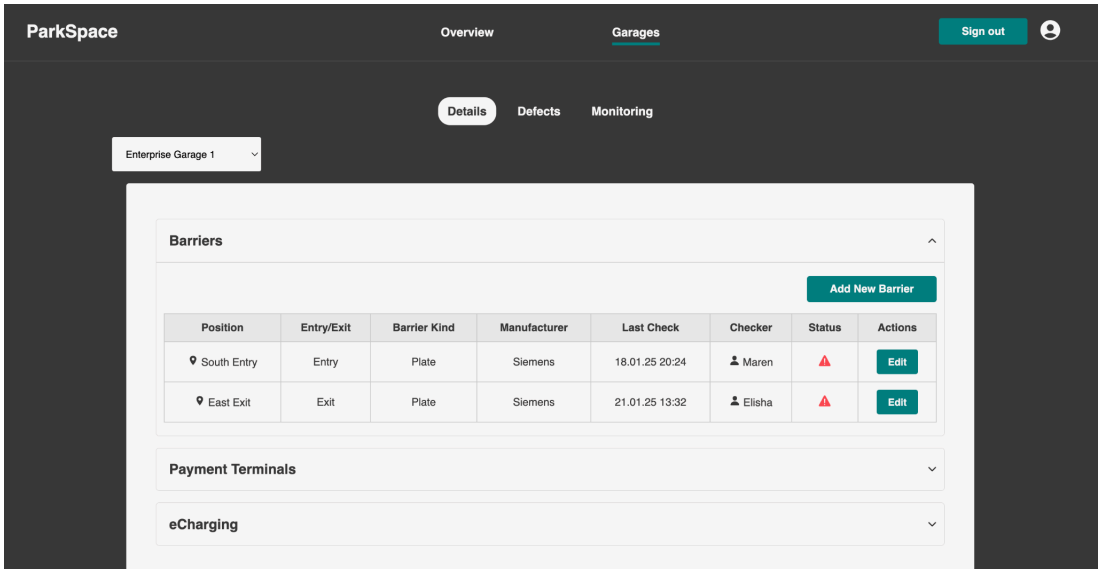


- **Landing Pages** to display all garages for non-signed-in users or personal garages for signed-in users

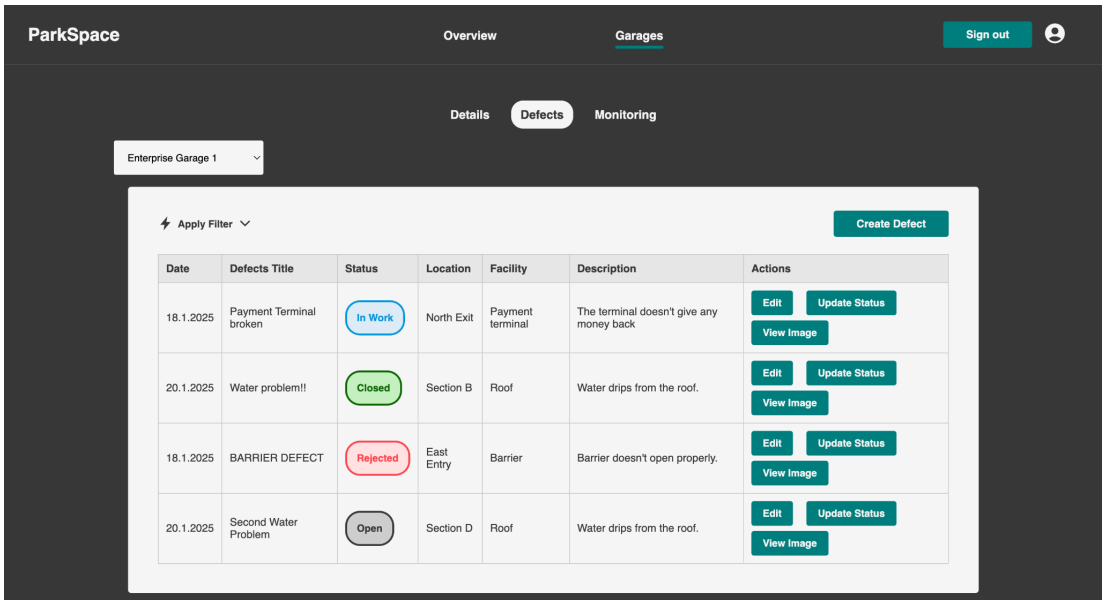




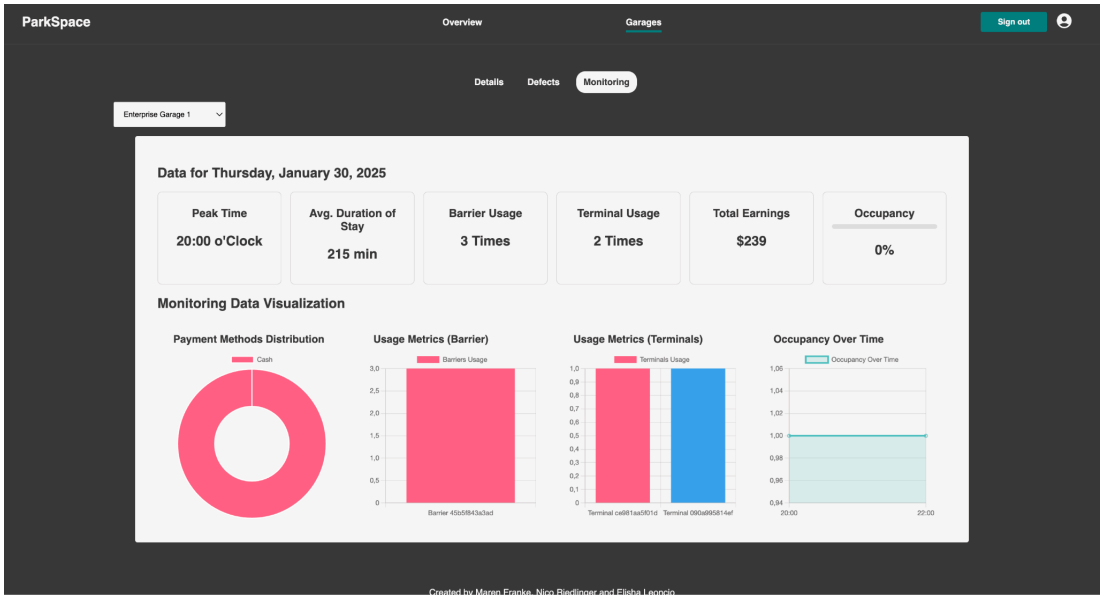
- **Details Page** to view garage facilities



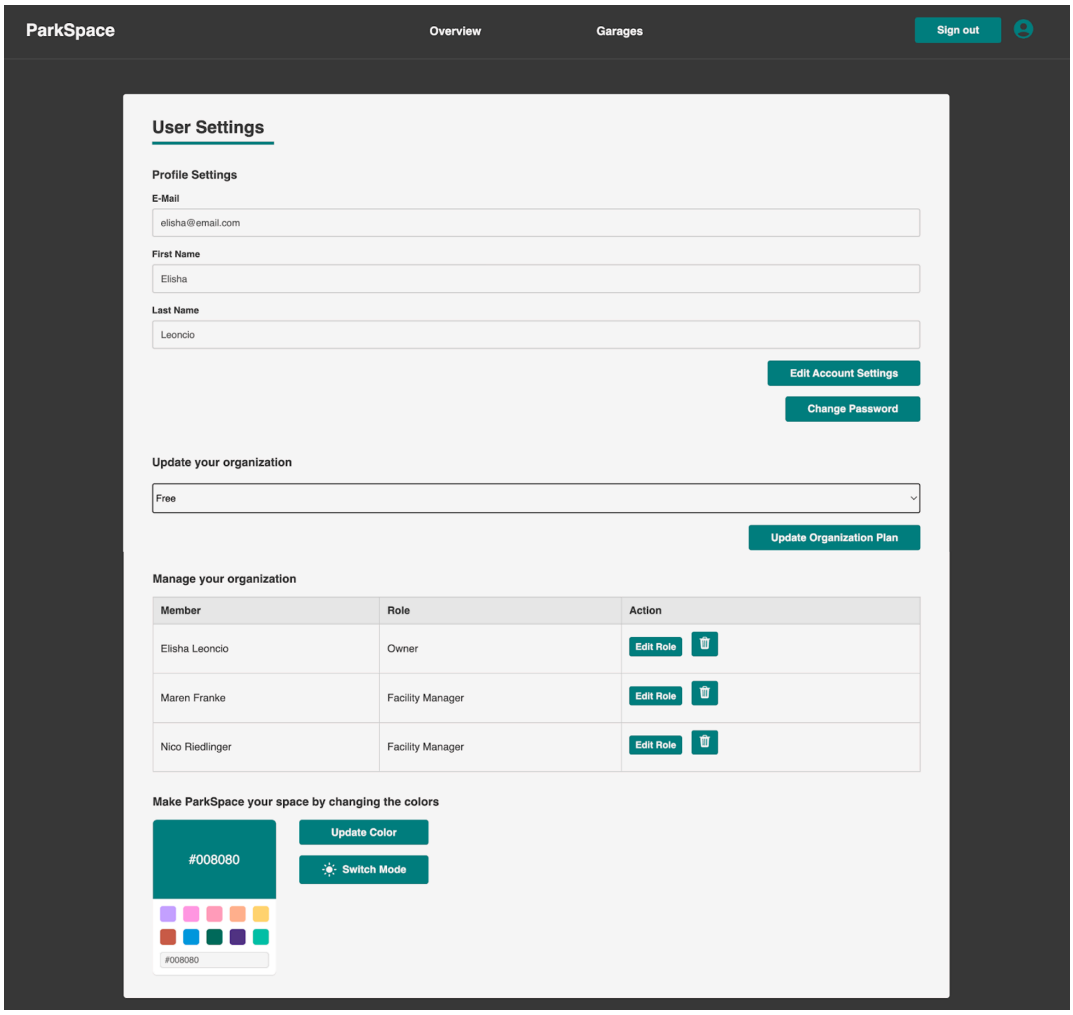
- **Defects Page** to see defects for a garage



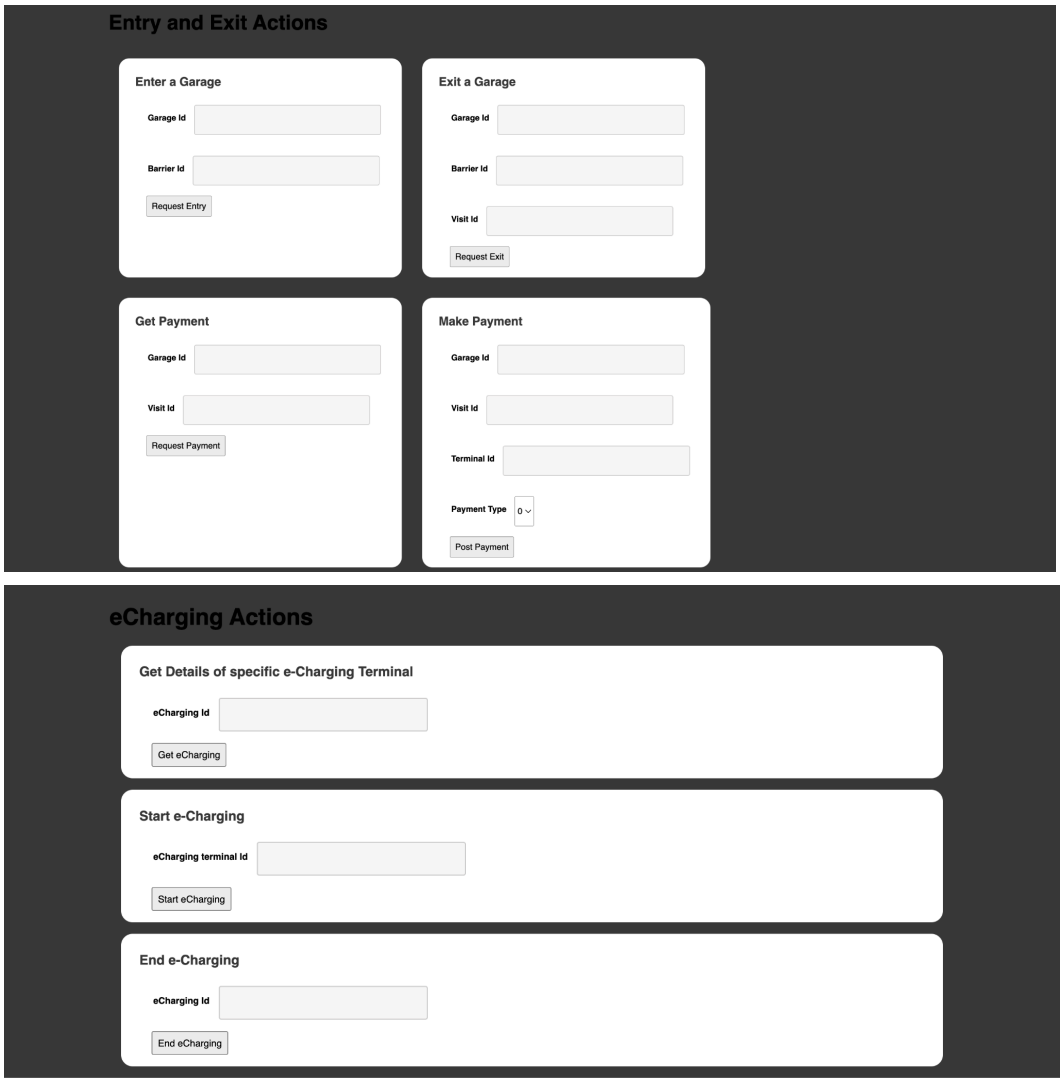
- **Monitoring Page** to monitor garages



- **User Settings Page** to manage accounts, create or join organizations, and handle organization settings



- **Simulation Page**, accessible only by manually adjusting the path, where users can simulate vehicle entries and exits for a garage



For authentication, users first create an account, with their data stored in Keycloak and Google Cloud FirestoreDB. When signing in, Keycloak sends a token to the frontend, which is decoded to verify user details. If the token lacks the required information, access to the application's features is denied. The token is also used to authenticate backend requests from the frontend.

A key customization feature allows organization owners to modify the application's color scheme for their organization, which updates the interface for all members. This color information is stored in FirestoreDB with the organization data and is retrieved to apply the new accent color via `variables.scss`

The application's backend requires dynamically constructed paths due to its use of multiple microservices and namespaces. The frontend queries the **User-Tenant Management** microservice to obtain user details and determine the appropriate namespace. For enterprise tenants, the namespace corresponds to their tenantId rather than their tenantType. Paths are then built using the format: `BASE_ADDRESS + tenantType/tenantId + MICROSERVICE`. To authenticate these backend requests, the token is sent as a Bearer token in the request headers.

3.4 Datastores

The data stores are also started and managed using the Terraform script. A FirestoreDB is created for each individual tenant, in which all information relevant to the application is stored. In addition, a bucket is created for each individual tenant to store the images for defect management.

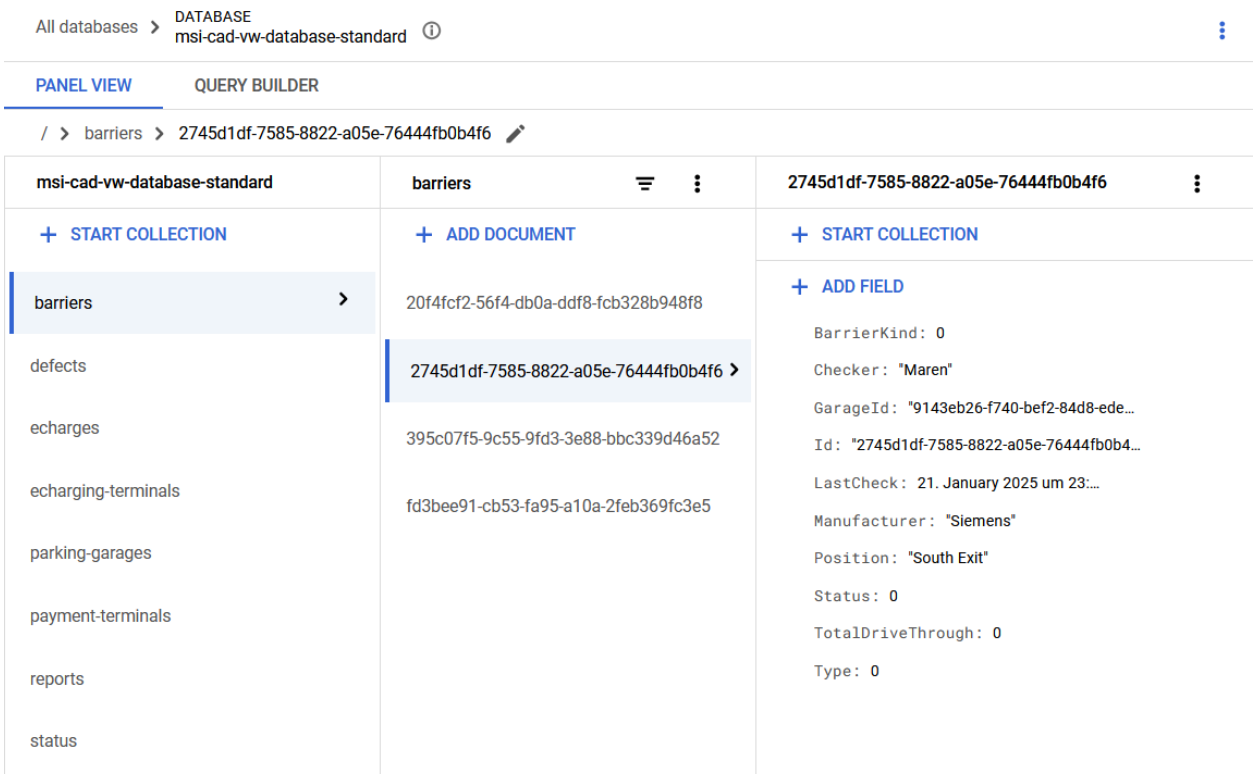
3.4.1 FirestoreDB

The database stores all alphanumeric values from primitive data types. The complete database model can be found in section 2.2. Because FirestoreDB is a document-oriented NoSQL database, the scheme functions more as a baseline for orientation. It is neither a complete E/R diagram, nor are the foreign keys validated between different collections.

A separate Firestore database is created for each tenant type to ensure that no data can be shared across different tenant types. In addition to each tenant type, the authentication services also have a separate database, because they are deployed as a singleton service in a separate namespace inside the cluster. They can be accessed from all tenant types and manage data not specific for a single tenant type. An overview of existing databases can be seen in the image below. Each database has the specific prefix *msi-cad-vw-database-* followed by either the tenant type's name or, with an enterprise tenant, its specific unique identifier.

Database ID ↑	Mode	Location	Creation time (UTC+1)
msi-cad-vw-database-50018682-a1e2-f7a4-f941-ce90cb13dd8c	Native	europe-west1	17 Jan 2025, 20:36:32
msi-cad-vw-database-authentication	Native	europe-west1	11 Jan 2025, 21:17:42
msi-cad-vw-database-bb6b3924-4f65-f28a-98b8-4b3be516a827	Native	europe-west1	22 Jan 2025, 10:04:17
msi-cad-vw-database-free	Native	europe-west1	17 Jan 2025, 20:36:28
msi-cad-vw-database-standard	Native	europe-west1	17 Jan 2025, 20:36:36

The data inside a database can be viewed from Google's web interface, like depicted in the image below. The database *msi-cad-vw-database-standard* of the Standard tenant type is selected. The left column lists all existing collections. In the middle column are all documents inside the selected collection, and in the right column are the details of a selected document. As can be seen, the identifier of a document is always a UUID/Guid.



The database scales automatically upon usage. The costs generated are calculated by the size of the stored data, the amount of the requests (read, write, delete) and the network bandwidth. The database is located in region *europe-west1* which means that it is a local database and also a bit more cost-efficient.

Access from C# to the database is done with the *Google.Cloud.Firestore* NuGet package by Google. Firstly, a collection is targeted by selecting it in the constructor. This is done in the *IdBaseDataProvider* base class.

```
protected CollectionReference _collection;

public IdBaseDataProvider(DatabaseConnectionProvider databaseConnectionProvider,
string collectionName)
{
      _collection =
databaseConnectionProvider.Database.Collection(collectionName);
}
```

Afterward, the `_collection` variable can be used to read from or write to different documents of the database. A simple example of an asynchronous read operation for a single document with a generic class `TEntity` is given below. If no results for the `ID` are found, a value of `null` will be returned by this method.

```
public async Task<TEntity?> GetAsync(TKey id)
{
```

```
        var snapshot = await _collection.Document(id.ToString()).GetSnapshotAsync();

        return snapshot.Exists
            ? snapshot.ConvertTo<TEntity>()
            : null;
}
```

3.4.2 Object Storage Buckets

Google Cloud Object Storage buckets are used to store binary data. This is necessary for defect management where images can optionally be attached to a defect.

All buckets in Google Cloud scale automatically, both in terms of capacity and performance. They are of the 'standard' type, which guarantees good availability and many read and write authorizations without any noticeable delay. The bucket is also regional to the *europe-west1* region.

The buckets are accessed using C# in the respective microservice's *Data* layer. For implementation, the *Google.Cloud.Storage.V1* NuGet package was used, which provides several methods for the upload of data into a storage bucket. Firstly, this package requires the instantiation of a `StorageClient` which is done at application startup along the other Google Cloud connections. Afterward, an asynchronous method can be used to stream the data into the bucket until it is completed, as depicted below.

```
var result = await _databaseConnectionProvider.FileStorageClient.UploadObjectAsync(
        _options.FileUploadBucket, name, contentType, uploadStream);
return result.Name;
```

The `result` will contain the unique identifier of the uploaded file (in our case, always an image) as the `Name` attribute. This can be set as the `ImageId` of a defect object.

To retrieve the uploaded images, a direct URL to view the image can be generated using a UrlSigner. More information about this can be found in section 3.2.5.

3.5 Cloud Functions

Cloud Functions are used for two purposes. First is the continued update of a parking garage's status in a given interval. The second purpose is the daily creation of operations reports. Nevertheless, the approach to set up a Cloud Function with a periodic execution is always the same.

To begin, a generic function is implemented. It can be configured either via environment variables or even using query parameters in the URL. For us, the common approach was to set rather static values, like the query interval, as environment variables and have more dynamic values in the query parameters. This led to setting up the database ID as a query parameter to easily differentiate between each tenant type. Other values, like the project ID, are set as environment variables, because it is expected that they do not change at all.

As the Cloud Function is deployed straight into Google Cloud, the credentials will always be set by Google. Only a service account must be chosen when setting up the Cloud Function. As per se, the connection to FirestoreDB can be made without much effort. Both types of Cloud Functions that we use

connect directly to the FirestoreDB collections. They do not call any endpoints of existing microservices, because the microservices are wrapped inside the Kubernetes cluster while Cloud Functions are not. When called, the functions execute their database operations and return an HTTP result code as well as some JSON indicating the changes made by the function.

To achieve a periodic execution, Google Cloud Scheduler is used, which is similar to cron jobs on a Linux operating system. A target URL can be configured as well as an execution time via a cron expression. The URL will be called every time the cron expression is fulfilled. By providing the Cloud Scheduler with the URL of a Cloud Function, it will call and execute the function each time.

Each tenant type has its own database. As the Cloud Functions can be provided with a database ID as query parameter, a Cloud Scheduler is set up for each tenant type as well. This way, the same Cloud Function will be called multiple times, but accessing different databases. This way, code can be reused and maintenance of this code is minimized.

4 DevOps

The entire application is managed by terraform and helm in the Google Application, to set up the Kubernetes-cluster, the pods and everything else inside of

4.1. Setup Infrastructure

4.1.1 Configuration

The core of the Infrastructure Setup is the Bucket `msi-cad-vw-backend`. The most important information about the application is stored here. On the one hand the state of Terraform and on the other hand the current configuration in the form of the data `tenants.json`. The latter configures the various tenants, their capacities and resources:

```
[
  {
    "name": "free",
    "replicas": "1",
    "maxReplicas": "2",
    "maxUnavailable": "75%",
    "maxSurge": "25%",
    "averageUtilization": "80",
    "maxCPU": "100m",
    "maxMemory": "150Mi"
  },
  {
    "name": "standard",
    "replicas": "2",
    "maxReplicas": "3",
    "maxUnavailable": "50%",
    "maxSurge": "50%",
    "averageUtilization": "60",
    "maxCPU": "150m",
    "maxMemory": "200Mi"
  },
  {
    "name": "50018682-a1e2-f7a4-f941-ce90cb13dd8c",
    "replicas": "2",
    "maxReplicas": "5",
    "maxUnavailable": "0",
    "maxSurge": "1",
    "averageUtilization": "50%",
    "maxCPU": "150m",
    "maxMemory": "200Mi"
  }
]
```

In this example, there are three different tenants: free, standard and an enterprise user. In addition, various information is defined, such as the maximum CPU and memory Utilization, the number of replicas, the maximum number of replicas and, for the rolling update, the number of maxUnavailable and maxSurge.

This file is used for the Terraform scripts as well as for the Helm configurations. It is therefore an essential part of the application.

4.1.2 Setup Application

An initial environment can be set up with the help of bash scripts and Terraform. A project must first be set up and a billing account created.

The Google APIs are then activated using the bash script.

```
echo Step 4: Enable APIs of $PROJECT_ID
gcloud services enable run.googleapis.com
gcloud services enable artifactregistry.googleapis.com
gcloud services enable firestore.googleapis.com
gcloud services enable compute.googleapis.com
gcloud services enable iam.googleapis.com
gcloud services enable cloudresourcemanager.googleapis.com
gcloud services enable container.googleapis.com
gcloud services enable iamcredentials.googleapis.com
gcloud services enable containerregistry.googleapis.com
gcloud services enable cloudfunctions.googleapis.com
gcloud services enable cloudbuild.googleapis.com
gcloud services enable cloudscheduler.googleapis.com
```

In addition, both the staging and environment environments are configured, the backend bucket and various environment variables for the GitHub pipeline are configured. The configurations for the HTTPS certificates must also be made in this first step. This is done again in a separate bash script.

The different environment variables must be added to all repositories for use in the multi-repo. To automate this, a Python script can be written that connects to the GitHub API and thus allows the configurations to be updated more quickly:

```
def send_request(owner: str, repo: str, encrypted_value: str, key_id: str, token:
str, secret_name: str):
    url =
f"https://api.github.com/repos/{owner}/{repo}/actions/secrets/{secret_name}"
    headers = {"Authorization": f"token {token}", "Content-Type":
"application/json"}
    payload = {
        "encrypted_value": encrypted_value,
        "key_id": key_id
```

```
    }

    response = requests.put(url, headers=headers, json=payload)
```

4.1.3 Terraform

Terraform can be used to configure the environment, which can then be added to and updated automatically.

This means that the entire environment is set up using the configurations defined in the `tenants.json` file described above. The following resource is defined around the information from `tenants.json`:

```
/* Request the static data out of the bucket */
data "google_storage_bucket_object_content" "tenants" {
  name   = "tenants.json"
  bucket = var.backend_bucket

  # depends_on = [google_storage_bucket_object.create_file]
}
locals {
  tenant_config = jsondecode(coalesce(data.google_storage_bucket_object_content.
                      tenants.content, "{}"))
}
```

This configuration can then be used to set up the components that are tenant-specific. This involves iterating over all elements of the configuration and executing the respective Terraform resource for each of these elements. The following example uses buckets:

```
resource "google_storage_bucket" "buckets" {
  for_each = { for tenant in local.tenant_config : tenant["name"] => tenant }

  name     = "${var.default}-storagebucket-${each.value["name"]}"
  location = var.region
  project  = var.project

  force_destroy = false
}
```

The following components will be set up by terraform:

- Kubernetes Cluster with Node-Pool
- Gateway-IP-Address and DNS-Zone
- For each tenant:
    - Database
    - Bucket
    - Service Account
    - Google Cloud Function (2 different)

    - Google Cloud Scheduler (2 different)
- Artifact Registry (Helm and Docker)
- Authentication Database

If there is a new tenant type, the resources will be started automatically.

4.1.4. Helm Charts

Terraform is used to set up the architecture, and Helm can be used to manage and configure Kubernetes itself. There are four different charts that deal with setting up the components. The values.yml defines the default values:

Infrastructure: Setting up the API gateway. Both HTTP and HTTPS are configured here if it is a productive environment, as well as the DNS address (or the associated IP address assigned). In addition, the namespace `infra` is started, in which the gateway will run.

Backend: Set up the different backend services for each tenant. The following configurations are set up here:

- **HTTP-Route**: Automatic link to the gateway and enabling the forwarding of HTTP requests to the corresponding services
- The following things are configured for each of the services (defect management, facility management, parking-garages, parking-management):
    - **Service**: Interface
    - **Deployment**: Pod to be actually set up.
    - **HealthCheck**-Policy: Configuration to get the service healthy
    - **HorizontalPodAutoscaler**: Configuration for scaling

Frontend: Setting up the graphical user interface. Very similar configurations take place here as for the backend, but there is only one frontend for all tenants.

Authentication: Setting up the user-tenant management. Here, too, the configuration is very similar to that for the backend, but there is only one for all tenants.

On the start of a new release with helm, different values need to be set. Here is an example for the startup of a helm chart out of the GitHub pipeline:

```
helm upgrade frontend ${{ env.HELM_CHART_PATH }}/frontend/helm \
    --set googleProject.name=${{ secrets.GC_PROJECT_ID }} \
    --set frontend.version=${{ inputs.frontend-version }} \
    --set productive=${{ inputs.productive }}
```

4.1.5. Example Configuration - Deployment Defect Management

Firstly, the basics must be defined to enable deployment in the first place:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.defectManagement.name }}-deployment
```

```
  namespace: {{ .Values.namespace }}
  labels:
    app: {{ .Values.defectManagement.name }}
```

This is immediately followed by the specification of the deployment. The replicas must be defined here, as well as the strategy for an update. The values for the rolling update are defined via tenants.json.

```
  replicas: {{ .Values.replicaCount }} # Number of pod replicas
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: {{ .Values.rollingUpdate.maxSurge }}
      maxUnavailable: {{ .Values.rollingUpdate.maxUnavailable }}
  selector:
    matchLabels:
      app: {{ .Values.defectManagement.name }}
```

The container specifications are also particularly interesting. Here, for example, the specifications for the Docker image must be transferred along with environment variables.

```
- name: {{ .Values.defectManagement.name }}-container
        image: {{ .Values.googleProject.location }}-docker.pkg.dev/{{
.Values.googleProject.name }}/{{ .Values.googleProject.dockerRepo }}/{{
.Values.defectManagement.image }}:{{ .Values.defectManagement.version }}
        ports:
          - containerPort: {{ .Values.defectManagement.port }}
        env:
        - name: MSICAD_Options__DatabaseId
          value: {{ .Values.resources.database }}-{{ .Values.namespace }}
        - name: MSICAD_Options__ProjectId
          value: {{ .Values.googleProject.name }}
        - name: MSICAD_Options__FileUploadBucket
          value: {{ .Values.resources.bucket }}-{{ .Values.namespace }}
        - name: MSICAD_Options__ServiceBaseRoute
          value: ""
```

There is also a configuration for the mount of a secret that contains the Google credentials. This secret must be mounted to the container on the one hand and added to the deployment as a volume on the other:

```
volumeMounts:
        - name: msi-cad-vw-secret
          mountPath: "/mnt/secret/" # Mount location inside the container
```

```
volumes:
      - name: msi-cad-vw-secret
        secret:
```

```
          secretName: {{ .Values.imagePullSecrets.name }}
          items:
          - key: key.json
            path: msi-cad-vw-secret
```

The configuration of the readiness test is also very interesting. Health checks, timeouts and the like can also be configured here:

```
readinessProbe:
   httpGet:
      path: /{{ .Values.defectManagement.healthCheck }}
         port: {{ .Values.defectManagement.port }}
      initialDelaySeconds:
{{.Values.healtCheck.readinessProbe.initialDelaySeconds}}
      periodSeconds: {{.Values.healtCheck.readinessProbe.periodSeconds}}
      timeoutSeconds: {{.Values.healtCheck.readinessProbe.timeoutSeconds}}
      failureThreshold: {{Values.healtCheck.readinessProbe.failureThreshold}}
      successThreshold: {{.Values.healtCheck.readinessProbe.successThreshold}}
```

The last interesting component is the resources that can be provided: CPU and memory limits and requests are defined here, which is also important for the configuration.

```
resources:
    limits:
        memory: {{ .Values.scaling.memoryLimit }}
        cpu: {{ .Values.scaling.cpuLimit }}
    requests:
        memory: {{ .Values.scaling.memoryRequest }}
        cpu: {{ .Values.scaling.cpuRequest }}
```

4.2 Environments

There are two environments, the staging and the productive staging. The staging environment is used to test new deployments and new configurations to prevent errors on the productive system.

	Staging	**Productive**
Project	msi-cad-vw-parking-staging	msi-cad-vw-parking
Cluster	msi-cad-vw-staging-cluster	msi-cad-vw-cluster
Hosting	HTTP without domain	HTTP and HTTPS with domain https://parkspace.tech
Database	Own databases in project	Separated database
Buckets	Own buckets in project	Separated buckets

Keycloak	One shared keycloak resource (treated as external resource)

The environments are handled in the different pipelines. There are two modes for almost each pipeline to deal with everything on staging and on productive. In each repository of our multi-Repo are two pipelines, where one will deploy new releases on staging and one on productive. More details will be described below.

4.2 Roles and Role Mapping

The different roles, tenants and organizations are handled by Keycloak, like described above. As a result, Keycloak takes over the most important tasks of authentication and authorization between the individual services and roles.

At DevOps level, there are various service accounts. One service account is required to access the Google Cloud and the Kubernetes cluster via terraform and helm. Therefore, one account is set up during the creation process.

Additionally, separated service accounts per tenant were planned, and the basic structure is already set up. In this case, each individual tenant would have received its own service account for authentication with the respective services, which the respective services would have received as a secret. For this purpose, a separate account is created for each tenant in the Terraform setup process.

4.3 Pipelines and Release of new Features

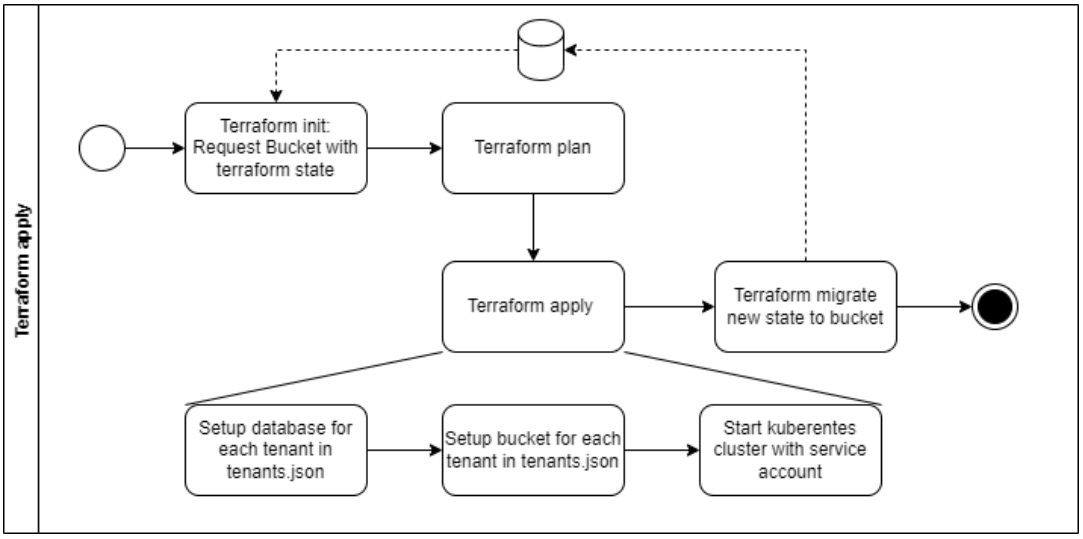
There are a lot of different pipelines for different use cases. In our Multi-Repo exists one repository which contains all pipelines, which will be called multiple times.

4.3.1. Workflows Repository

In this **workflow repository** are a lot of different pipelines for different use cases:

	Idea	**Gets**	**Steps**
Docker Build	Builds and pushs docker images	Google-Secrets, service name and docker paths	1. Execute: Get new version 2. Sign in to google and docker 3. Building 4. Pushing with new version
Docker Set Version	Identifies next needed version for a docker image	Google Secrets, service	1. Get current version 2. Increase version by one
Docker version	Gets all current versions of each docker image	Google Secrets	1. Get all images and their version 2. Return the current version
[Dummy]	Tests if the C#	.NET version and name	1. Get version

.NET Testing	Unit-Tests will run	of module	2. Build project (restore) *3. (dummy) Execute tests*
Helm Build	Builds and pushs helm charts	Google-Secrets, service name and docker paths	1. Sign in to google and docker 2. Build package 3. Push to helm registry
Helm start one tenant	Starts (or stops) a release of the backend charts with a secret for a specific tenant	Google Secrets Versions, cluster, database names, bucket names, startORstop	1. Sign-in google, docker, kubernetes 2. Get tenants.json and select tenant 3. Install backend chart with inputs 4. Install the secret 5. Restart the entire backend 6. OR: uninstall backend
Helm Start / helm start tenants	Starts the existing helm charts: backend for each tenant, frontend, gateway, authentication and the secrets for it	Google Secrets Versions, cluster, database names, bucket names, Certificates	1. Sign-in google, docker, kubernetes 2. Get tenants.json 3. Install authentication and frontend 4. Install release gateway 3. Install backend chart with inputs 4. Install the secret 5. Restart the entire backend
Helm upgrade	Upgrade all existing helm releases	Google Secrets Versions, cluster, database names, bucket names, Certificates	1. Sign-in google, docker, kubernetes 2. Get tenants.json 3. Upgrade authentication, frontend and gateway 3. Install backend chart with inputs
Terraform apply	Plan, apply and migrate state	Google Secrets, workdir	1. Sign-in google, docker, kubernetes 2. Initialize with backend in google 3. Terraform apply
Terraform destroy	Destroy the Kubernetes cluster	Google Secrets	1. Sign-in google, docker, kubernetes 2. Initialize with backend 3. Terraform destroy kubernetes



The following code-snippet is an example for the workflows:

```
on:
  workflow_call:
    outputs:
      version:
        value: ${{jobs.set-version.outputs.version}}
    inputs:
      service:
        required: true
        type: string
    secrets:
      GC_SERVICE_ACCOUNT:
        required: true
      GC_SERVICE_ACCOUNT_SECRET:
        required: true
      GC_PROJECT_ID:
        required: true
```

The workflows in this repository will be triggered by a `workflow_call`. They receive different inputs and get a lot of secrets from the calling repository. Additionally, a few workflows (like shown above) return an output.

4.3.1 Development Pipelines (DevOps)

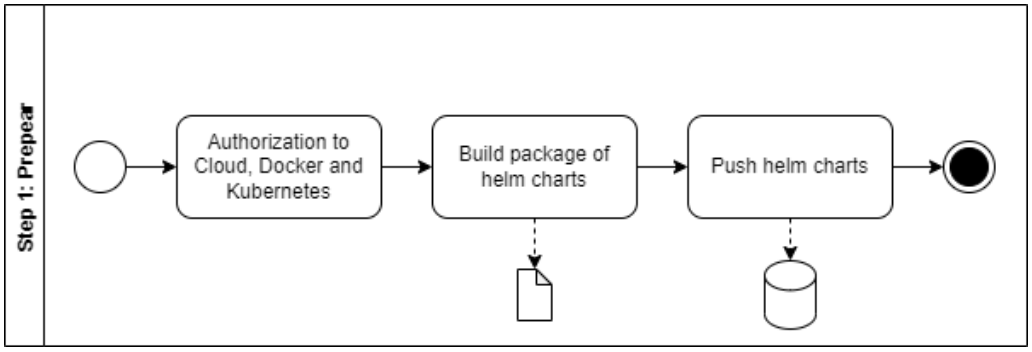
The DevOps repository is used for different workflows that manage the infrastructure of the application. The following workflows are included:

	Description	Uses (workflows)
Uninstall Services (Helm destroy)	Uninstalls all helm releases in the given environment (staging or productive)	Helm-stop

Kubernetes Create	Setups the entire application with terraform and helm. Can also be used to reinstall the application, if there are some changes. It can be configured if productive or staging should be used.	Terraform-apply Docker-Version Helm-start
Kubernetes destroy	Destroy entire Kubernetes Cluster with shutting down the Kubernetes Application. Can be done for productive and staging.	Terraform destroy
Restart Services	Restart all Helm-Services of the defined environment.	Docker-Version Helm-start

In the DevOps repository, the helm-charts are stored in the folder `/helm`. If changes are happening in this repository the following Pipeline will be executed:

Helm push	Pushs new status of helm-charts to the helm-registry in the Google Artifact registry for staging and productive.	Helm build

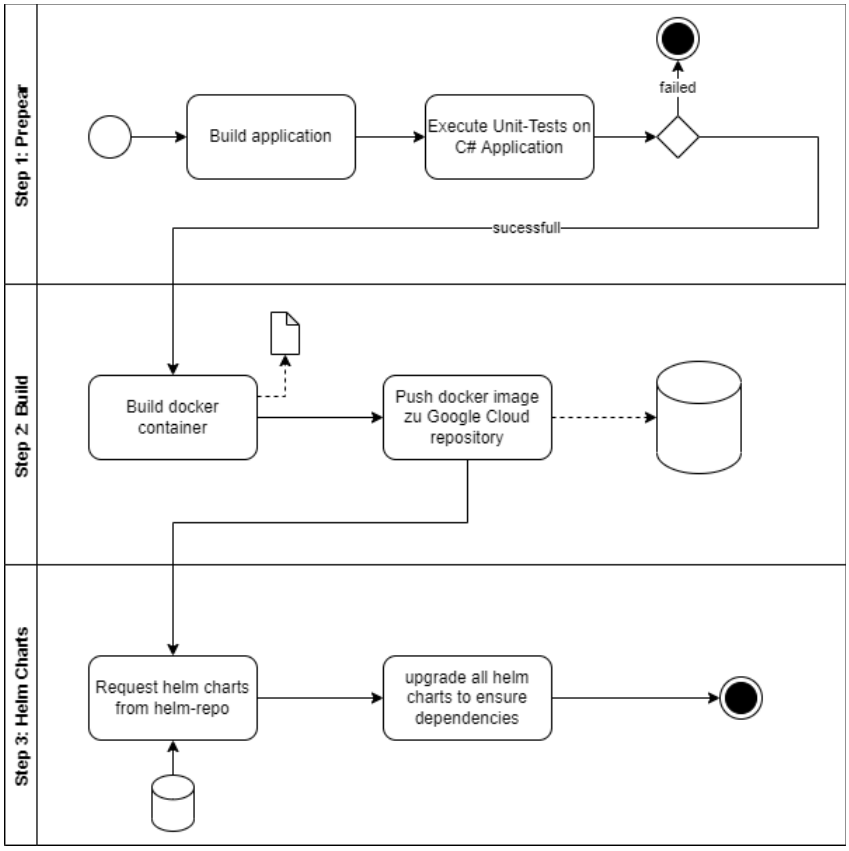


There is a pipeline to create a new tenant, used for adding or deleting a new tenant.

Tenant Management	Create or delete a tenant. Therefore, a new tenant will be added to `tenant.json` and the terraform and helm pipelines will be executed.	Terraform apply Helm one tenant

4.3.2 New Releases

Every repository owns a pipeline with deployment to staging (develop) or productive (main). If a push to develop happens, the staging-pipeline will be executed and for the main branch the productive pipeline.



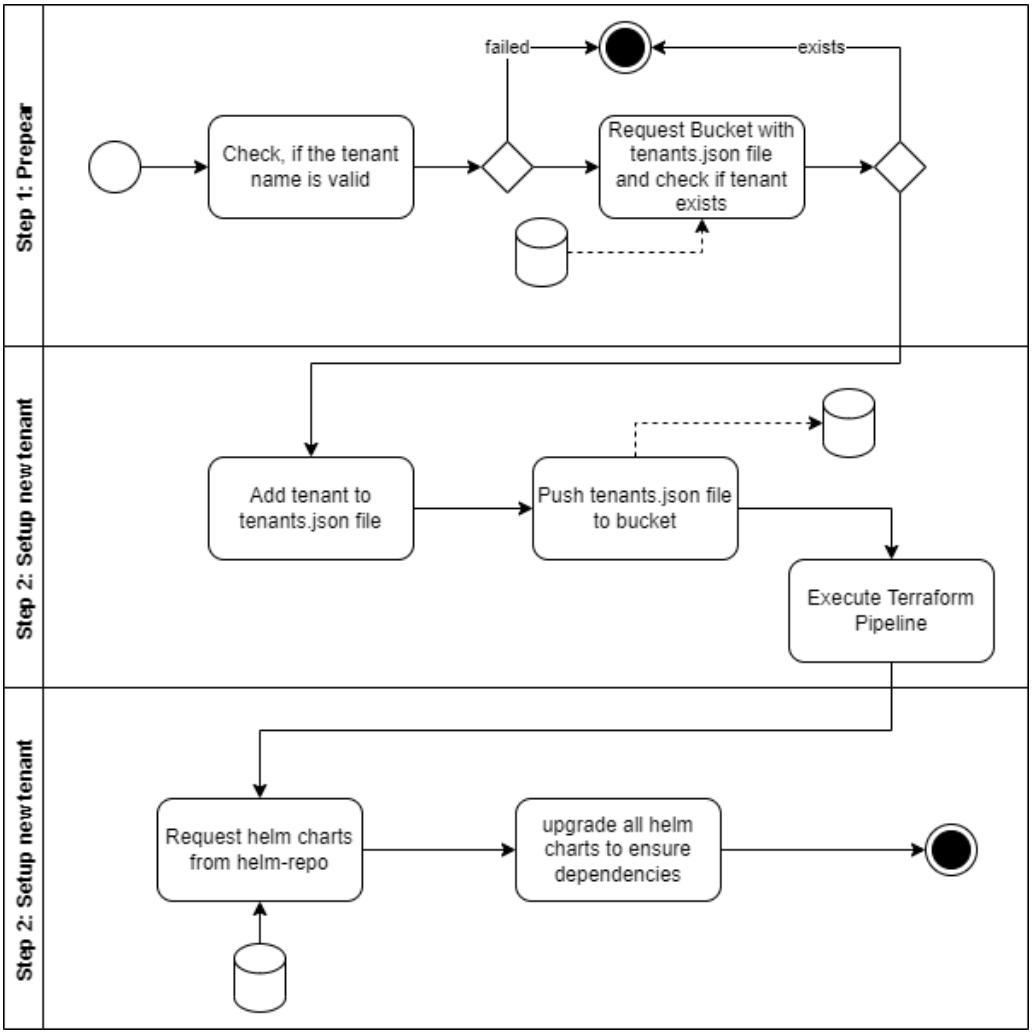
The pipeline will be executed for both staging and productive. The only difference is, that once the staging-project and cluster is used and once the productive system.

4.4 New Tenants

Any user can create a new organization and will be the organization owner thereafter. By creating a new organization, a database entry will be created with the organization details. Furthermore, the user will be linked to the new organization in FirestoreDB as well as Keycloak. In FirestoreDB, the `OrganizationId` is populated, while in Keycloak a custom attribute is used. This custom attribute is also mapped to the access token and, thus, will be included there. All this is done by the *UserTenantManagement* service. To access Keycloak and set the organization ID, the Keycloak REST API is used like described in section 3.2.6.

Any users can request to join an organization. They must be accepted or can be denied by the owner. All these processes are handled by the *UserTenantManagement* service. If a user is accepted to an organization, the `OrganizationId` will be entered in FirestoreDB as well as Keycloak, just like with an organization admin.

If a new enterprise tenant is created, the tenant-management-pipeline must be executed additionally with the tenant-id. And then the following workflow will be executed:



4.5 Monitoring

4.5.1. Monitoring of System Components

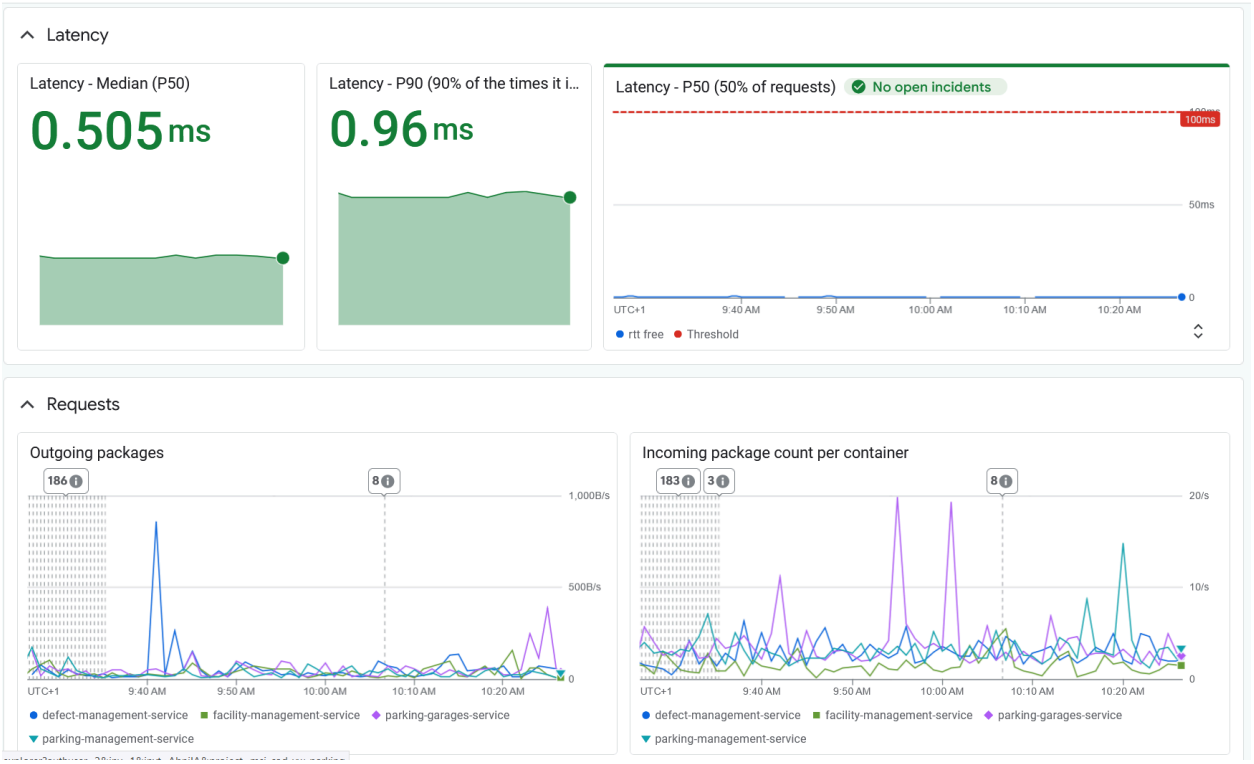
Organizations with a Standard or Enterprise tenant type can access a dedicated monitoring page for their parking garages. This page provides a clear and dynamic visualization of key data metrics, updated daily through Google Functions. Displayed information include:

- Peak Time
- Average duration of stay
- Barrier usage
- Terminal usage
- Total earnings
- Occupancy
- Payment Method Distribution

4.5.2 Monitoring of Runtime Environment

For each tenant type, a dashboard with important information will be created with terraform. In this dashboard, the most important information about the namespaces is displayed:

- Latency of Products
- Requests (Incoming and outgoing)
- Datastorage
    - Firestore (Requests, Latency, …)
    - Bucket (Storage, Requests, …)
- CPU-Usage and Resources



4.6 Load Testing

Three tests were carried out for the load testing:

- “Standard requests” with a normal number of users (10 users simultaneously making requests)
- “Peak usage” on all tenants - high utilization for the cluster (up to 7500 parallel users sending requests to the microservices every second)
- “Tenant test” - increasing the number of tenants to test the maximum number of enterprise tenants

Locust, a Python library that is well suited for load tests, was used for testing. In Locust, a separate user class was created for each tenant type. This made it possible to run the same load tests in parallel for each tenant:

```
class FreeUser(HttpUser):
    tasks = [BackendTasks, FrontendTasks]
    wait_time = between(1, 5)
    def on_start(self):
        self.tenant = "free"
```

The backend tasks contain HTTP get requests, which make requests to all existing services. A token is required for each request. This is initially generated when the user is created. Standard GET requests are sent for the respective services, for example to query and return all garages from the database. In this way, all services and the databases connected to them can be used.

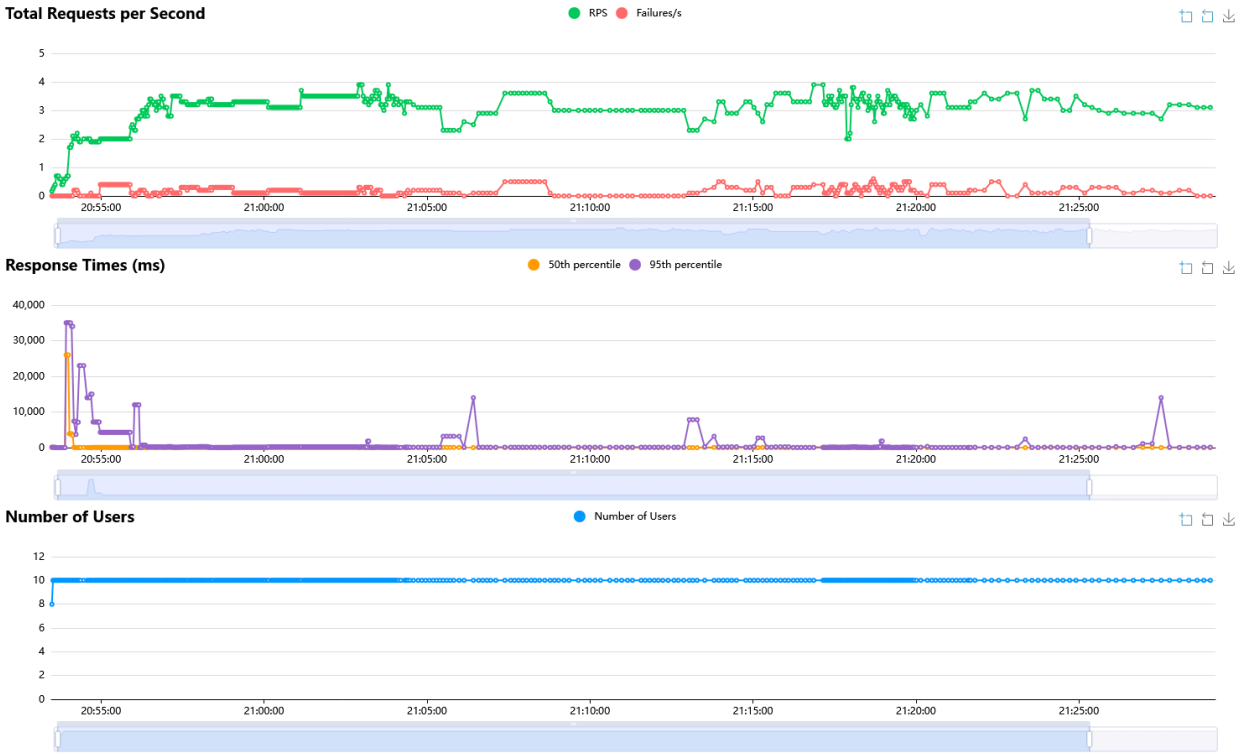
```
@task(2)
def get_parking_garages(self):
    if self.token:
        headers = {"Authorization": f"Bearer {self.token}"}
        response = self.client.get(f"/{self.tenant}/parking-garages/
                   ParkingGarage", headers=headers)
```

The files of the load tests are stored in the devOps repository in the *loadtesting* folder.

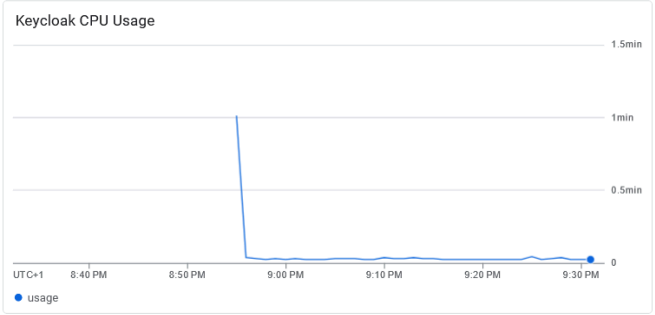
4.6.1. Locust - Average usage behavior

The standard load test, in this case was the use of 10 parallel users. These made requests to all backend paths and to the frontend, and therefore also to all services. There are four different parallel namespaces requested, one for each tenant (free, standard, enterprise-1 (bb6b3924-4f65-f28a-98b8-4b3be516a827) and enterprise-2 (50018682-a1e2-f7a4-f941-ce90cb13dd8c)).

The following figure shows the progression of the requests in the evaluation of Locust:



Various points are striking. One is the very large peak in the response time at the start of the query. This peak is due to Keycloak. The Keycloak container must first wake up in order to process the requests for the tokens for the

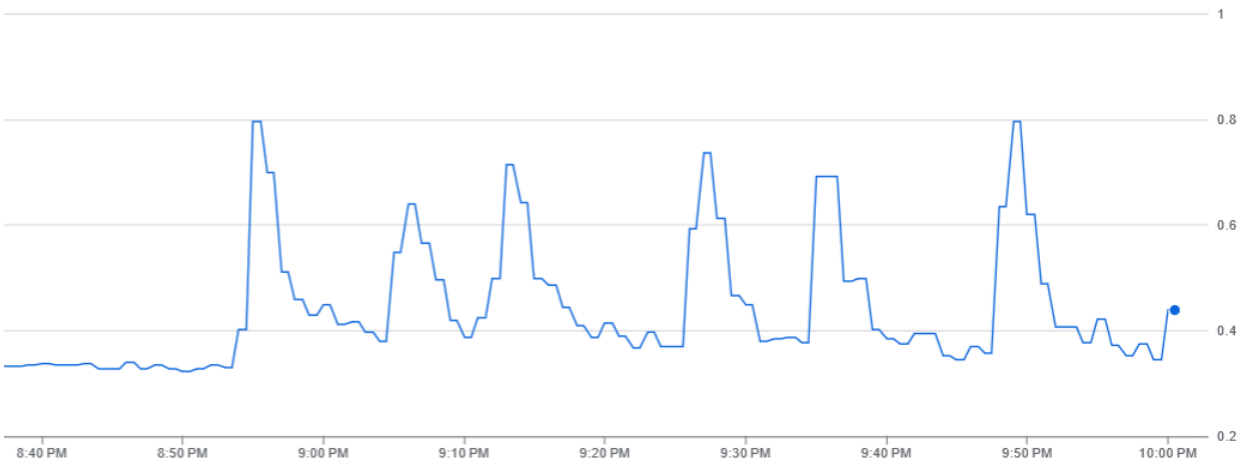


individual users. The CPU utilization of Keycloak can be seen on the right in the figure, and it is clear that the container had a lot to do at the beginning when it was in the startup process. The startup time there were around 30 seconds, which can explain the initial latencies very well.

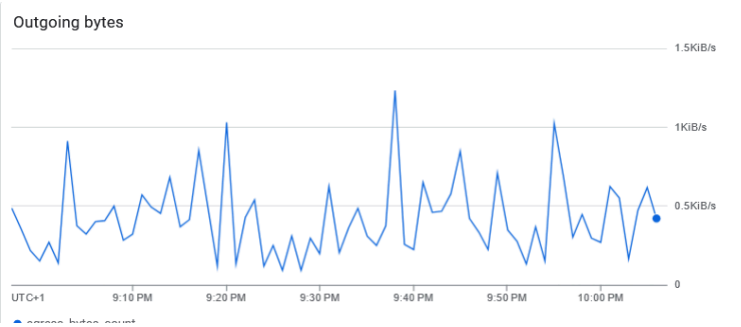
	Name	Occurrences
GET	/free/defect-management/Defect	18
GET	/free/defect-management/HealthCheck	10
GET	/free/facility-management/Barrier	12
GET	/free/facility-management/HealthCheck	16

The strongly fluctuating error rates and the fact that errors occur at all are also striking. The services that are sometimes unavailable are the free-tenant services.These have the least resources available and crash from time to time under low load. However, only a small “off-time” is risked here and the pods are restarted after a short time. Such overloads are also particularly noticeable with the Free-Tenant, as only one replica is started here. This means that accessibility is no longer guaranteed if a pod is stopped due to too much memory or CPU usage (as configured to prevent Noisy Neighbor). This is the case with the other tenants.

It is also interesting to look at the overall utilization of the cluster during the tests. At the start of the test, the load increases significantly. In the meantime, it drops again and again.



This drop and increase can be associated with the number of requests and bytes sent out on the one hand, but also with the starting and stopping of various pods that could be stopped in the meantime due to CPU utilization.

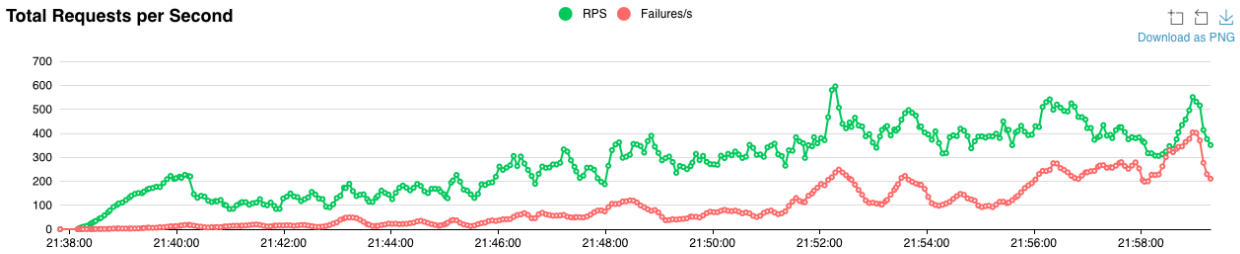


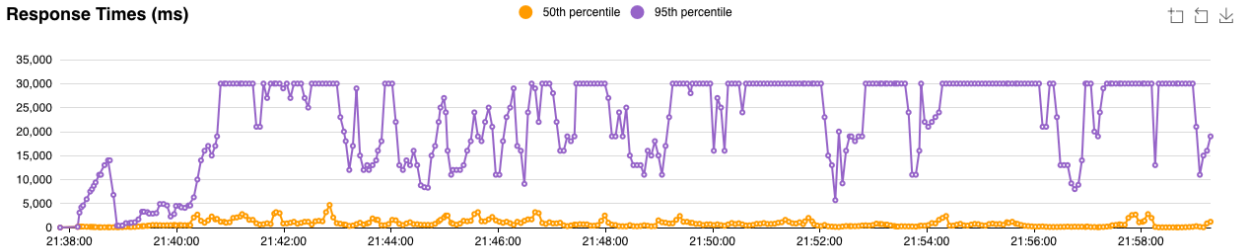
The load test is not noticeable in the latencies of the database responses. These are therefore not the limiting factor of the application.

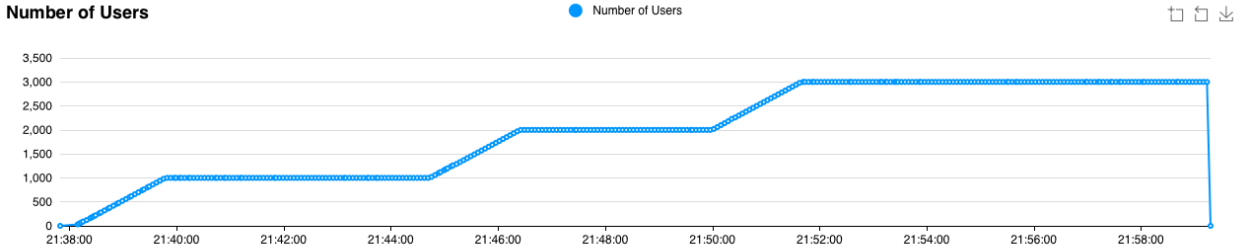
4.6.2. Locust - Peak-Usage

During a load test to put the cluster and the entire application into an exceptional state, two computers were used, each of which executed a Locust script. This meant that twice the number of requests could be executed without jeopardizing the capacity of their own computer.

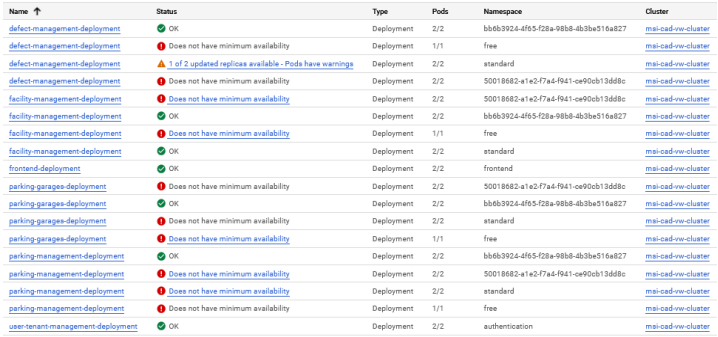
For the test, up to 1500 users were created in parallel on one laptop and 3000 on the other. These users then sent requests every few seconds, simulating the use of many parallel users (e.g. in a once-in-the-lifetime event).







The number of users was gradually scaled up and it was clear to see that the error rate increased as the number of requests increased. These errors were always responses with 503 or 504, i.e. “Service unavailable” or “Gateway timeout”. A check of the services also showed that individual services had been terminated.



Name	Status	Type	Pods	Namespace	Cluster
defect-management-deployment	OK	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
defect-management-deployment	1 of 2 updated replicas available - Pods have warnings	Deployment	2/2	standard	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
facility-management-deployment	OK	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
facility-management-deployment	OK	Deployment	2/2	standard	msi-cad-vw-cluster
frontend-deployment	OK	Deployment	2/2	frontend	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
parking-garages-deployment	OK	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
parking-management-deployment	OK	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
user-tenant-management-deployment	OK	Deployment	2/2	authentication	msi-cad-vw-cluster

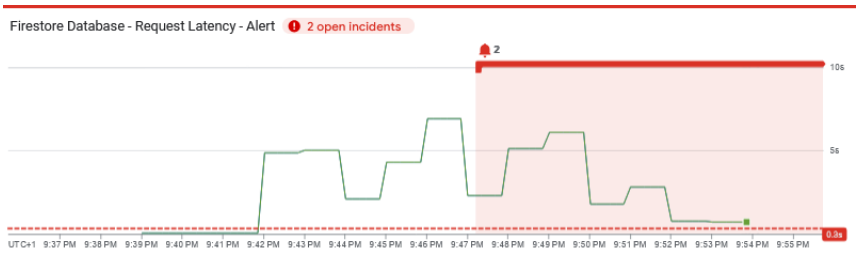
A more detailed analysis of the individual services revealed that the services were terminated due to the maximum capacities permitted for the

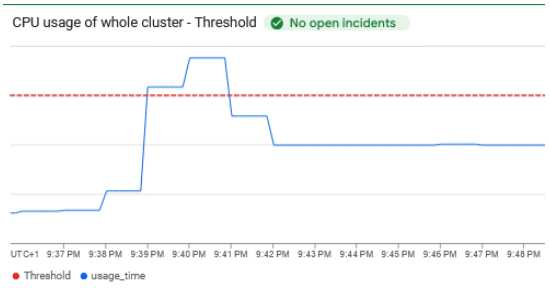
respective service being exceeded. These capacities depend on the respective service and can be adjusted in the tenant configuration. The upper limit chosen here is quite low due to the small overall size of the cluster. In this specific case, the pod has exceeded the permitted upper limit of memory usage and was therefore killed by the node.

```
Events:
  Type  Reason   Age   From   Message
  ----  ------   ----  ----   -------
  Warning     Evicted  14m   kubelet  The node was low on resource: memory. Threshold
quantity: 100Mi, available: 101648Ki. Container facility-management-container was using
97824Ki, request is 0, has larger consumption of memory.
  Normal      Killing  14m   kubelet  Stopping container facility-management-container
```

The other resources are also interesting to look at. On the one hand, it can be seen that Keycloak also receives some requests and has quite a high latency.

Also, very interesting is the increasing latency in the Firestore Database, which rises significantly due to the many requests. The threshold selected in the graph is quite low, but it is very representative of how many requests are actually sent and how much load is on the application.





It is also exciting to evaluate the CPU utilization of the cluster. This is because it has a high utilization at the beginning due to the high number of requests to the services. However, after a short time, parallel to the termination of individual services, the CPU utilization also decreased significantly. This is due to the fact that the services were terminated and can therefore no longer allocate CPU.

After terminating the load test and a short wait, however, all services were able to restart due to a well-configured restart policy and the cluster then worked perfectly again.

A second load test (this time even with three parallel computers) following the first led to a very similar result and progression of the individual requests. During the second test, the only problem that occurred at some point was that no more tokens could be provided by Keycloak.

4.6.3. Increase Tenant Number

Testing the maximum number of parallel tenants. For this purpose, additional tenants are started via the GitHub pipeline. This makes it possible to start many “dummy tenants” and thus test the capacity of the cluster for a maximum number of enterprise tenants.

The number of organizations that log in to the standard and free accounts is only limited by the maximum number of requests and resources that are released for the respective containers. The entire authorization and authentication process is handled by Keycloak and the user-tenant-management-service.

Additional tenants are added to the existing tenants (free, standard and two enterprise). A deterioration in capacity is already noticeable after the creation of an additional tenant.

When a second and third additional tenant is created, it takes quite a long time to start. It should also be noted that individual pods no longer start, or terminate in the meantime due to resource bottlenecks. However, at least one pod is available for each service, thus ensuring the accessibility and functionality of the application.
When a third tenant is created, the capacities are already exceeded and the pods can no longer start properly.

This shows that the current cluster has too few resources available for this number of tenants. This could be regulated by reducing the number of pods to one per tenant, or by increasing the capacities for the cluster.

Name	Status	Type	Pods	Namespace	Cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-1	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-4	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-3	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-2	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
defect-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-4	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-3	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-1	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-2	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
facility-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
frontend-deployment	Does not have minimum availability	Deployment	2/2	frontend	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-1	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-4	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-2	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-3	msi-cad-vw-cluster
parking-garages-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-2	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-4	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	standard	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	bb6b3924-4f65-f28a-98b8-4b3be516a827	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	1/1	free	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	50018682-a1e2-f7a4-f941-ce90cb13dd8c	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-1	msi-cad-vw-cluster
parking-management-deployment	Does not have minimum availability	Deployment	2/2	loadtesting-tenant-3	msi-cad-vw-cluster
user-tenant-management-deployment	Does not have minimum availability	Deployment	2/2	authentication	msi-cad-vw-cluster

4.6.4 Conclusion

The capacities provided for the project are not particularly high for cost reasons. The load limit of the current system is correspondingly low. However, if more tenants were required or the system were to go live, the capacities of the cluster could simply be increased. This would solve the problem of low load capacity

The limitation of the maximum CPU and memory utilization of the individual tenants for the individual microservices is also good in principle, but is currently also scaled very low. However, this can be adjusted by simply adapting the tenant configuration. The positive aspect here, however, is that even if there is a high load for one service, the other services are not affected (no noisy neighbor).

The load testing showed the limits of the system and was able to demonstrate some of the system's behavior, so it can be considered successful.

5 Commercial Model

5.1 Tenant types

5.1.1. Overview

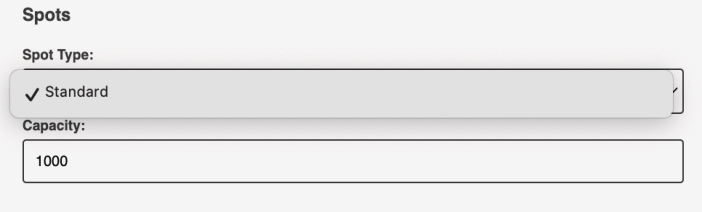
Feature	Free	Standard	Enterprise
Number of parking garages	1	5	unlimited
Number of parking spaces	max. 1000	unlimited	unlimited
Type of parking spaces	no special parking spaces	Standard, Disabled, E-Parking	Standard, Disabled, E-Parking
E-Charging	-	possible	possible
Defect management	-	no status (frontend only)	unlimited
Reports	-	unlimited	unlimited
Roles	no Facility Manager	all	all
Advertising	a lot	none	none
Customization			
Parking prices	one price	standard/day/night	fully customizable
Color scheme	fixed	color selection	fully customizable
Hardware			
FirestoreDB	Shared: one database (per service) for all customers	Shared: one database (per service) for all customers	Dedicated database
Buckets	Shared: one bucket for all customers	Shared: one bucket for all customers	Dedicated bucket
Replica	1 for all	2 for all	2 for each
Resources Limits	CPU: 100m	CPU: 150m	CPU: 150m

	Memory: 150Mi	Memory: 200Mi	Memory: 200Mi
Namespaces	shared	shared	own

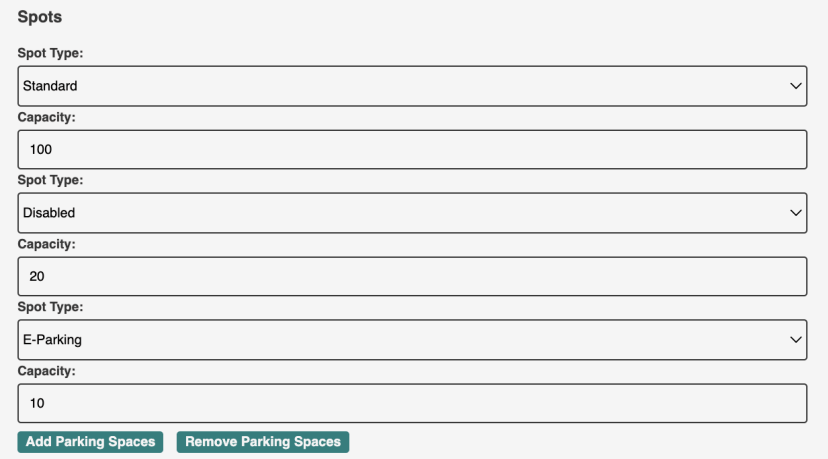
5.1.2 Examples in Application

Differences in ‘Create a Garage’ - Parking Spots:

- ‘Free’ can only create standard spots and only have up to 1000 spots.



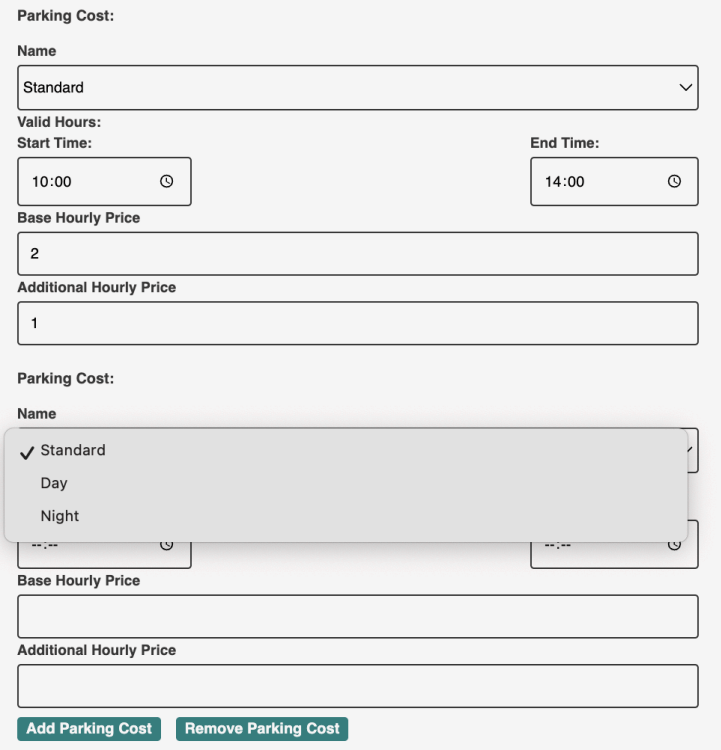
- ‘Standard’ and ‘Enterprise’ can add different spot types and have an unlimited amount of spots.



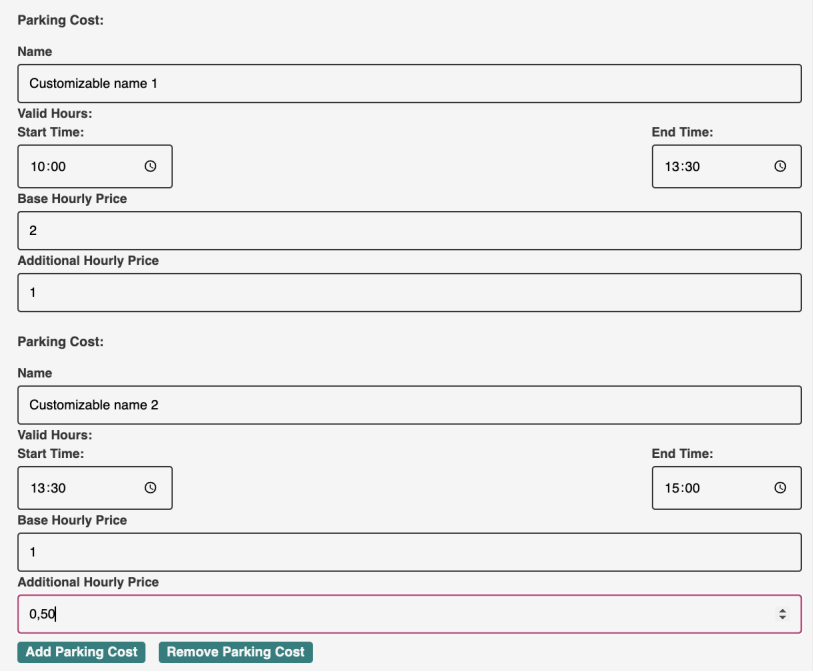
Differences in ‘Create a Garage’ - Parking costs:

- ‘Free’ only has one parking cost.

- ‘Standard’ can add different parking costs and choose between pre-defined names.

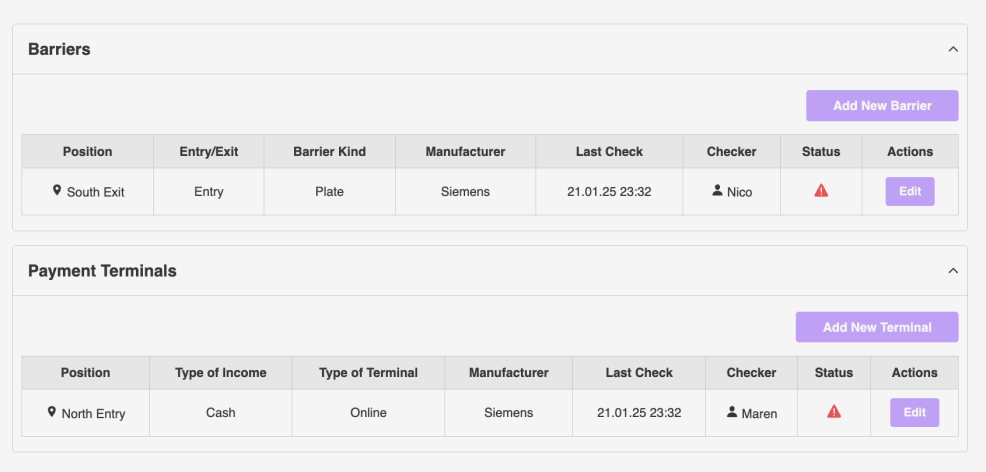


- ‘Enterprise’ can freely choose parking costs and fully customize the parking cost name.

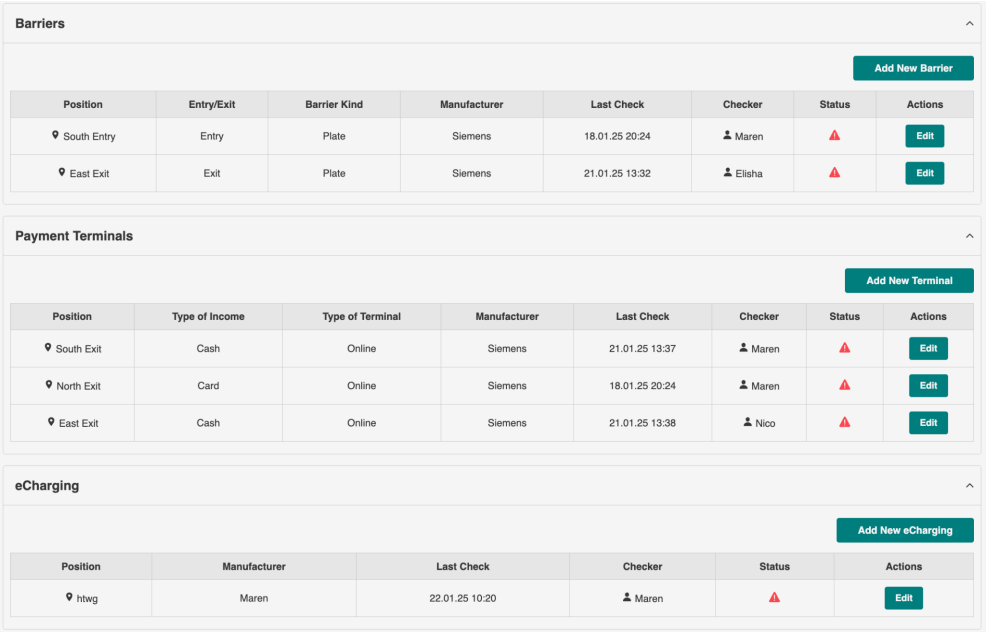


Differences in Facilities:

- ‘Free’ has the option to add Barriers and Payment Terminals.

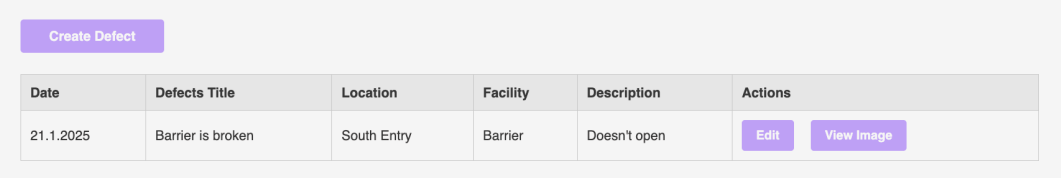


- ‘Standard’ and ‘Enterprise’ can add Barriers, Payment Terminals and E-Charging Terminals.

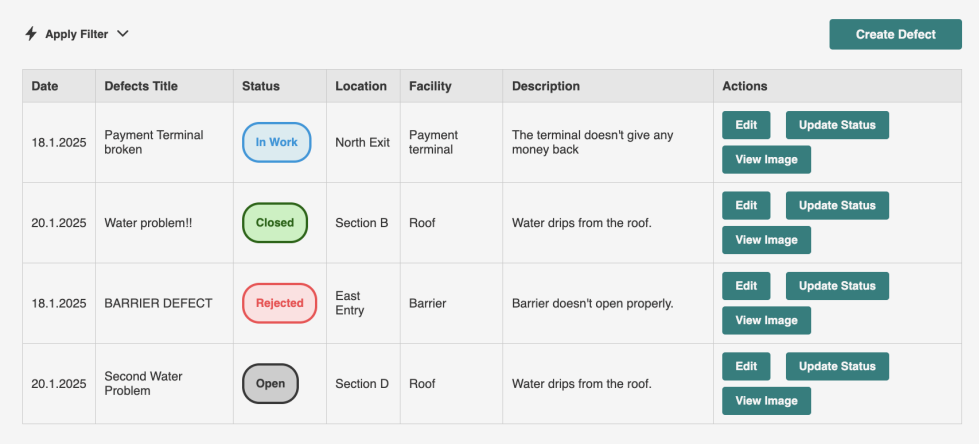


Differences in Defects:

- ‘Free’ does not have ‘Defect Management’.
- ‘Standard’ has ‘Defect Management’.



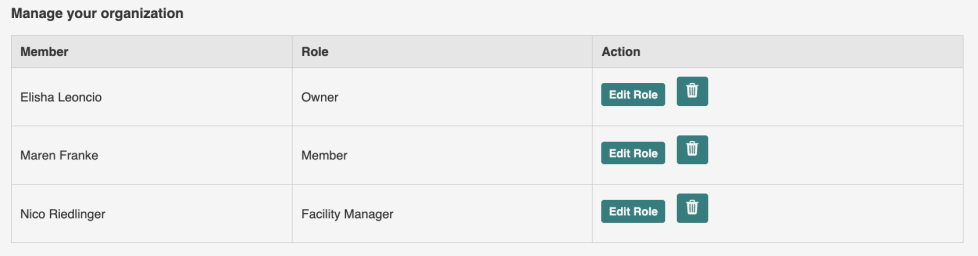
- ‘Enterprise’ has ‘Defect Management’ and this view also displays the status of each Defect.



Differences with Organization roles:

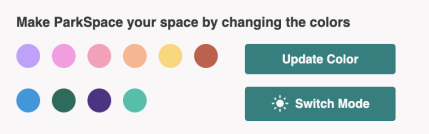
- ‘Free’ only has the member and owner role, and this view cannot edit roles.

- ‘Standard’ and ‘Enterprise’ have all roles and the possibility to edit the roles.

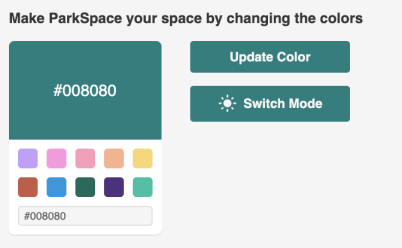


Differences with colors:

- ‘Free’ cannot change the color of the application.
- ‘Standard’ can change the color based on a predefined color palette and switch between modes.



- ‘Enterprise’ can change to whatever color they like and switch between modes.



5.2 Cost and Pricing Model

5.2.1 Costs

For each tenant, the following user statistics are suggested:

- Logged-In Users (Manager, Facility Manager, Member) per day: 5
  - Create new defects / change them: 5 per day
  - Request monitory: 10 per day
  - Create new parking garage / facilities: 2 per day
- Visitors of the parking garage per day: 1000
- Visitors of web-application: 100 per day

To calculate the expected costs for the application the Google Cloud Pricing Calculator[1] was used.

Resource	**Values**	**Costs**
Firestore [per tenant]	- Document-Reads/Writes per day: 100.000 (multiple requests per action) - Document Deletes per day: 10 - Stored Data: 5GiB	€4 per month
Bucket [per tenant]	- Stored Data: 5GiB	€0,1 per month
Functions [2 per tenant]	- Number of requests per month: 1 Mio (minimum value) - Average execution duration: 600ms	€2,78 per month
Cloud run (KeyCloak)	- Average execution duration: 300ms - Number of request: 10Mio per month	€4,15 per month
Kubernetes	Kubernetes cluster with 2 CPUs and 4Gi Memory	€150 per month
Static IP-Address	One static IP-Address	€7,29 per month

The amount for the other components, like the Backend Bucket, the Artifact Registry and so on are too small and will not be considered in this calculation.

The basic usage is summed up and rounded €330 per month:

- 2x Kubernetes Cluster: €300
- Keycloak + Authentication DB: €8,15
- 2x Static IP-Addresses: €15

For each enterprise tenant additional costs, round about €20 per month, need to be calculated:

- 1x FirestoreDB: €4
- 1x Bucket: €0,1
- 2x Function: €6
- More traffic and other additional costs: €10

[1] https://cloud.google.com/products/calculator

If a new enterprise tenant joins, the costs will be increased by around €20. In the other cases, only the traffic will be increased, which will be considered by €10.

It needs to be considered that the overall costs of the system may increase, if a lot of tenants and users will join, because an increase of the Kubernetes cluster would be necessary to deal with the additional traffic. In this case, there should be an additional amount of costs.

5.2.2 Pricing

Therefore, the following costs will be suggested:

- Free: €0
- Standard: €40
- Enterprise: €150

Case	Tenants	Earned Money	Costs	Result
Best case - without increasing cluster	Free: 0 Standard: 10 Enterprise: 4	10* €40 + 4* €150 = €400 + €600 = €1000	10 * €10 + 4 * €20 + €330 = €510	€490
Best case - Increasing cluster	Free: 0 Standard: 50 Enterprise: 10	50* €40 + 10* €150 = €3500 = €1000	Ca. €1400	€2100
Worst case	Free: 100 Standard: 0 Enterprise: 0	€0	100 * €10 + €330	- €1330
Average Case	Free: 5 Standard: 5 Enterprise: 2	5* €40 + 2* €150 = €500	10 * €10 + 2 * €20 + €330 = €470	€30

In the best case, we will have a lot of Enterprise and Standard users. In this case, we will be able to earn money. If we have more than four enterprise tenants, it will be necessary to increase the size of the cluster. In this case, a doubling of the resources would be necessary, which would also mean at least a doubling of the costs.

In the average case, we will more or less sum up to zero.

In the worst case, we will have a lot of free users that generate costs but no enterprise or standard user, which will earn money. Here, we planned to use advertisements to prevent this case.