

Parking Garage

David Albrecht, TEKO Bern

May 2023

Contents

1	Project summary	4
2	Project initialization	4
2.1	Initial situation	4
2.2	Situation analysis	4
2.2.1	Product situation	4
2.2.2	Competitive situation	5
2.3	General conditions	6
2.3.1	Process-related general conditions	6
2.3.2	Product-related general conditions	7
2.4	Delimitations	9
2.5	Stakeholder analysis	10
2.5.1	Stakeholders	10
2.6	Project planning	10
2.6.1	Project Initiation	10
2.6.2	Requirement Engineering	11
2.6.3	Implementation	11
2.6.4	Verification	11
2.6.5	Development Plan	12
2.6.6	Documentation	12
2.6.7	Conclusion	12
3	Risk analysis	13
4	Change management process	13
5	Stakeholder requirements	15
6	System requirements	16
6.1	Functional requirements	16
6.2	Non-Functional requirements	17

7 System architecture and design	18
7.1 Svelte / SvelteKit	18
7.1.1 What is Svelte	18
7.1.2 What is SvelteKit	19
7.1.3 Why Sveltekit instead of other full-stack javascript frameworks	19
7.2 Containers	20
7.2.1 What are Containers	20
7.2.2 OCI Containers	21
7.2.3 Docker	22
7.3 Prisma	23
7.3.1 ORM	23
7.3.2 PostgreSQL	24
7.4 ERD	25
8 Implementation	27
8.1 Documentation workflow	27
8.1.1 Folder Structure	28
8.1.2 Buildscript	29
8.1.3 Index.md	30
8.2 Development environment	30
8.2.1 Setup Development environment	31
8.2.2 Cloud development environments (CDEs)	32
8.3 Project folder structure	33
8.4 Prisma Schema	33
8.4.1 Models	34
8.5 Algorithm	39
8.5.1 Implementation	39
8.6 Customer UI	43
8.6.1 Entry barrier	43
8.6.2 Checkout	47
8.7 Admin UI	50
8.7.1 Garage overview	50
8.7.2 Add garages	52
8.7.3 Edit Garages	55
8.7.4 Garage details	61
8.8 Parking rates	64
8.8.1 Implementation	65
8.9 Accounting	68
8.9.1 Implementation	71
9 Verification and validation	76
10 Configuration manual	77
10.1 Configuration elements	77
10.2 Project environment	77
10.3 Issues	77

11 Hosting	77
11.1 SvelteKit on Railway	78
12 Conclusion	79
13 Declaration of Independence	79
References	80

List of Figures

1 Gantt	10
2 Risk analysis	13
3 Change management process	14
4 Comparing Docker to VM's	20
5 ERD	25
6 local development environment	31
7 Project folder structure	33
8 Entry barrier	43
9 Exit barrier permanent tenants	48
10 Exit barrier	48
11 Garage overview	50
12 Overview with sample data	51
13 New garage form	52
14 Edit garage	56
15 Edit prices	56
16 Garage details	61
17 Level details	62
18 Accounting	68
19 Generate report	69
20 Report	69
21 Yearly report	70
22 Detailed overview of the yearly report	70
23 Railway deployment	78

List of Tables

1 Product-related general conditions	7
2 List of Stakeholders and their abbreviation	10
3 List of Stakeholders and their abbreviation	15
4 Functional requirements	16
5 Non-Functional requirements	17
6 Parking rates weekdays	64
7 Parking rates weekends	64

1 Project summary

The company ParkingTown requires a new development of their in-house parking management software solution because the old system is out of date and has a lot of technical debt. The decision was made to not use any already existing software solution but to develop a new application. The proof of concept prototype will take around three months to develop. The new Application will allow the parking garage account managers from ParkingTown to have a better overview of their garages. The accounting team from ParkingTown will benefit from the new improved report generation system to have a more automated billing and cash flow overview. The parking house customers benefit from an easier-to-use and prettier user experience.

This prototype is an important next step in the digitalization and strategy of ParkingTown.

2 Project initialization

2.1 Initial situation

The company ParkinTown currently manages five multi-story parking garages at different locations on behalf of several customers. The IT system used for this purpose was developed by a former employee who carries out the necessary maintenance work on a contract basis. The system is getting on in years and will have to be replaced in the near future, given the company's planned expansion plans. The managing director of ParkinTown has evaluated several IT solutions for the management of parking garages but considers them all too complicated. He prefers a simple solution tailored to ParkinTown's needs. He is aware of the risks involved in developing it himself and therefore, before the development of the entire application, he wants to have a prototype developed that simulates the operation of the parking garages. Based on the information in this document, a specification for the prototype must first be created. The information in this document must be checked for completeness and extended if necessary. Then a prototype of the IT system must be built based on the specifications.

2.2 Situation analysis

The current software is already old and becoming harder to maintain. To follow ParkinTown's business strategy, new management software is needed.

2.2.1 Product situation

The parking garage manager from ParkingTown is a tool that supports the company's employees in their daily tasks involving the management of parking garages on behalf of their clients.

The new Software should be tailored to the processes of ParkingTown and support the employees in their daily tasks.

ParkinTown is a company that manages multiple parking lots for various clients. Currently, the company uses an IT system that was developed by a former employee and requires maintenance work to keep it functioning. However, the system is outdated and needs to be replaced to accommodate the company's future expansion plans.

The managing director of ParkinTown has evaluated several IT solutions but has found them to be too complicated. Therefore, he desires a simple solution that is tailored to the company's specific needs. However, he is aware of the risks involved in developing an IT system in-house and wants to have a prototype developed to simulate the operation of the parking garages before the development of the entire application.

The prototype must be developed based on a comprehensive specification that needs to be created. The specification must take into account all the requirements and operational needs of the parking garages managed by ParkinTown. It must also consider the different locations and customers of the company.

The prototype should be a functional system that simulates the operation of the parking garages. It should have all the necessary features and functionalities to manage the parking garages effectively. Additionally, it should be user-friendly and easy to use by the company's employees and customers.

The success of the prototype will depend on how well it meets the requirements of the company and its customers. It must be able to handle the volume of parking transactions effectively, and it should also be reliable and secure.

The development of a prototype for the IT system used in managing the parking garages is a critical step toward the success of ParkinTown's expansion plans. It will help the company to identify any gaps in the current system and ensure that the new system meets all the requirements of the company and its customers.

2.2.2 Competitive situation

In the parking garage management industry, there are several IT solutions available that offer similar services to ParkinTown. Some of ParkinTown's competitors include:

1. TIBA Parking Systems - TIBA Parking Systems is a global leader in parking solutions. They offer a comprehensive suite of parking solutions that include hardware, software, and services. TIBA's parking management system is highly customizable and can be tailored to meet the specific needs of each client. ([tiba-parking, 2023](#))
2. Skidata - Skidata is another major player in the parking garage management industry. They offer a range of parking solutions, including software, hardware, and services. Skidata's solutions are designed to be highly efficient and can handle high volumes of parking transactions. ([skidata, 2023](#))
3. Parkmobile - Parkmobile is a mobile parking solution that allows customers to pay for parking using their mobile devices. The company's solution is highly convenient for customers and can be integrated with other parking management systems. ([parkmobile, 2023](#))

4. FlashParking - FlashParking is a cloud-based parking management solution that offers a range of features, including real-time parking availability, revenue management, and customer insights. FlashParking's solution is designed to be highly scalable and can be used by parking garages of all sizes. ([flashparking, 2023](#))

Compared to these competitors, ParkinTown's current IT system is outdated and lacks some of the advanced features offered by these companies. However, ParkinTown's managing director believes that a simple and tailored solution is better suited to the company's needs.

Therefore, the success of ParkinTown's new IT system will depend on how well it meets the specific needs of the company and its customers. ParkinTown may have an advantage over its competitors in terms of the personalized service it can offer to its customers. Additionally, if the new system is user-friendly and easy to use, it may help ParkinTown differentiate itself from its competitors and attract more customers.

2.3 General conditions

ParkinTown requires a new parking garage management tool.

2.3.1 Process-related general conditions

1. Project Management: A project manager should be appointed to oversee the development of the new IT system for ParkinTown. The project manager should be responsible for ensuring that the project is completed on time, within budget, and to the required quality standards.
2. Stakeholder Management: All stakeholders, including the managing director of ParkinTown and the customers should be involved in the development process. Their input should be sought throughout the project to ensure that their needs are met.
3. Requirements Engineering: All requirements for the new IT system should be gathered and documented in detail before the development process begins. This will help ensure that the end product meets the desired functionality.
4. Design: A detailed design specification should be created based on the requirements gathered in the previous step. This design should include all aspects of the system, including user interfaces, database design, and system architecture.
5. Quality Assurance: A quality assurance process should be put in place to ensure that the new IT system is of high quality and meets the required standards. This should include testing and validation of the system to ensure that it works as intended.
6. Documentation: All aspects of the development process, including design decisions, testing results, and system specifications, should be documented. This will allow for future maintenance and updates to be carried out more easily.
7. Training and Support: Training and support should be provided to all users of the new IT system to ensure that they can use it effectively.
8. Data Privacy and Security: The new IT system should be designed with data privacy and security in mind. Measures should be put in place to ensure that

customer data is protected and that the system is secure from external threats. This may include encryption of sensitive data and regular security updates.

2.3.2 Product-related general conditions

Table 1: Product-related general conditions

ID	Trace From	Description
GCR-101	Requirements	The IT system must allow for flexible configuration of the parking garages, including the number of floors and parking spaces per floor. The system should also enable customization of parking rates and tariffs, which may vary depending on the time of day and day of the week.
GCR-102	Requirements	Each parking garage should have one entrance and one exit barrier. The system must be able to generate digital parking tickets for occasional users, which include the date and time of entry as well as the assigned parking space. Permanent tenants should enter a unique personal code at the entrance barrier, and their tickets should be validated before exit. All entry and exit times must be logged for billing and evaluation purposes.
GCR-103	Requirements	The IT system must support two categories of customers, casual users and permanent tenants. Permanent tenants should be assigned a fixed parking space and pay a monthly rent. If the rent is not paid on time, the customer should be blocked until payment is made. The system should also ensure a balanced distribution of parking spaces for occasional users.

ID	Trace From	Description
GCR-104	Requirements	<p>The application should offer a graphical representation of the different floors per parking garage, showing the free and occupied parking spaces as well as those of the permanent tenants. The system should be able to generate reports and evaluations per parking garage and customer category over a defined time period at any time. The turnover of each parking lot should be calculated and displayed for a given month and annually, with individual months shown separately.</p>
GCR-105	Requirements	<p>The amount owed by occasional users should be calculated based on the length of stay and the applicable parking tariff. The system should automatically switch to a daily flat rate for stays longer than 24 hours, and the full amount should be charged for the elapsed days. After payment, a digital exit ticket should be issued to the customer.</p>

2.4 Delimitations

1. Geographic scope: The system will be designed to meet the parking management needs of a single municipality or town, rather than multiple locations or regions.
2. Language and cultural context: The system will be developed in a specific language and cultural context, which may not be applicable or easily adaptable to other contexts or languages.
3. Hardware and software limitations: The system will be designed to operate on specific hardware and software platforms, and may not be compatible with other systems or technologies.
4. Timeframe: The project will focus on the development and implementation of the IT system, rather than the long-term maintenance or evolution of the system over time.
5. Budget constraints: The system will be designed to meet specific budgetary constraints, which may limit the scope or functionality of the system.
6. Parking garage configurations: The system will be designed to support a specific range of parking garage configurations, and may not be easily adaptable to other configurations or designs.
7. User categories: The system will be designed to support two specific categories of users (casual users and permanent tenants), and may not apply to other categories of users.
8. Parking rates and tariffs: The system will be designed to support a specific range of parking rates and tariffs, which may not apply or adaptable to other contexts or markets.
9. Algorithm for parking space allocation: The system will use a specific algorithm to allocate parking spaces for occasional users, which may not be easily adaptable to other algorithms or methods of allocation.
10. Onboarding of new customers to long-term customers and automatically billing/blocking them is not implemented, an API for integration in accounting or IAM tools is provided.

2.5 Stakeholder analysis

2.5.1 Stakeholders

Table 2: List of Stakeholders and their abbreviation

ID	Stakeholder
CUPT	Customers (Casual Users and Permanent Tenants)
PGO	Parking Garage Owners
IDSA	IT Developers and System Administrators
MAR	Municipal Authorities and Regulators
FPSP	Financial Institutions and Payment Service Providers

2.6 Project planning

The Parking Garage project aims to develop a software system that automates the management of parking garages. The project requires a comprehensive approach that involves planning, analysis, design, implementation, verification, and documentation. The Gantt chart outlines the key activities involved in the project.

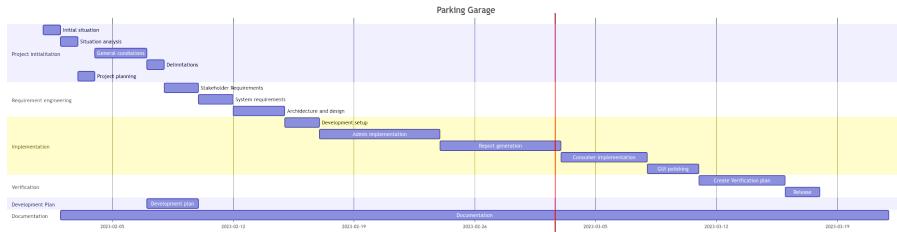


Figure 1: Gantt

2.6.1 Project Initiation

The project initiation phase involves defining the scope and objectives of the project, identifying stakeholders, and creating a project plan. The activities involved in this phase are:

1. Initial Situation (1 day): This activity involves setting up the project, defining the project team, and identifying the project's objectives.
2. Situation Analysis (1 day): This activity involves analyzing the current parking system in place, including identifying the strengths and weaknesses of the current system.
3. General Conditions (3 days): This activity involves defining the constraints, assumptions, and dependencies of the project.
4. Delimitations (1 day): This activity involves setting the boundaries of the project, and defining what is included and excluded.

5. Project Planning (1 day): This activity involves creating a detailed project plan, including timelines, resources, and budgets.

2.6.2 Requirement Engineering

The requirement engineering phase involves gathering, analyzing, and documenting the requirements for the software system. The activities involved in this phase are:

1. Stakeholder Requirements (2 days): This activity involves gathering the requirements of the stakeholders, including the customers, users, and other interested parties.
2. System Requirements (2 days): This activity involves translating the stakeholder requirements into system requirements.
3. Architecture and Design (3 days): This activity involves designing the architecture of the system, including defining the components, interfaces, and data flows.

2.6.3 Implementation

The implementation phase involves developing and testing the software system. The activities involved in this phase are:

1. Development Setup (2 days): This activity involves setting up the development environment, including installing the necessary software and hardware.
2. Admin Implementation (7 days): This activity involves developing the administrative features of the system, including user management and database management.
3. Report Generation (7 days): This activity involves developing the reporting features of the system, including generating reports on occupancy, revenue, and other metrics.
4. Consumer Implementation (5 days): This activity involves developing the consumer-facing features of the system, including user interfaces and payment processing.
5. GUI Polishing (3 days): This activity involves refining the user interface design to improve the user experience.

2.6.4 Verification

The verification phase involves testing the software system to ensure that it meets the requirements and is free of defects. The activities involved in this phase are:

1. Create Verification Plan (5 days): This activity involves creating a plan for testing the system, including defining the test cases, test data, and test procedures.
2. Release (2 days): This activity involves releasing the system to the stakeholders after it has been tested and approved.

2.6.5 Development Plan

The development plan phase involves creating a plan for the long-term development and maintenance of the software system. The activities involved in this phase are:

1. Development Plan (3 days): This activity involves creating a plan for the ongoing development of the system, including identifying future features and improvements.

2.6.6 Documentation

The documentation phase involves technical documentation, and other materials to support the use and maintenance of the software system. The activities involved in this phase are:

1. Documentation (49 days): This activity involves creating user manuals, technical documentation, and other materials to support the use and maintenance of the software system.

2.6.7 Conclusion

The Parking Garage project requires a comprehensive approach that involves planning, analysis, design, implementation, verification, and documentation. The Gantt chart provides a detailed overview of the key activities involved in each phase of the project.

3 Risk analysis

Item	Scenario		Risk	Origin	Comment
	Positive	Negative			
Customer impersonates long term customer	The impersonation is prevented	The impersonator can successfully open the barrier	1 4	x	To mitigate the effect when this risk would occur, a 24/7 support hotline number is displayed that can be called, the hotline have a break/guest account that allows opening the barrier
Customer presses checkin button	Left barrier opens	The east barrier is not opening	1 5	x	
Customer pays the parking fee	Correct parking fee is displayed	Calculation of the parking fee does not work	1 3	x	
Reports generation	The reports get generated correctly	The reports have wrong information	1 2	x	

ETW

1 = very unlikely
2 = unlikely
3 = possible
4 = occasionally
5 = frequent

Effect

1 = inconceivable
2 = inadmissible
3 = disturbing
4 = strongly disturbing
5 = catastrophic

Figure 2: Risk analysis

The risk analysis indicates, that it needed to work with a third-party support company. The parking garage is open 24/7 and it's necessary, that if a customer would be trapped inside that they can call a support hotline for help. Otherwise, the risks are not too high since the application is more used to present data and not as an accounting tool for the customers.

4 Change management process

Change in the software and the business processes is handled according to the change management process

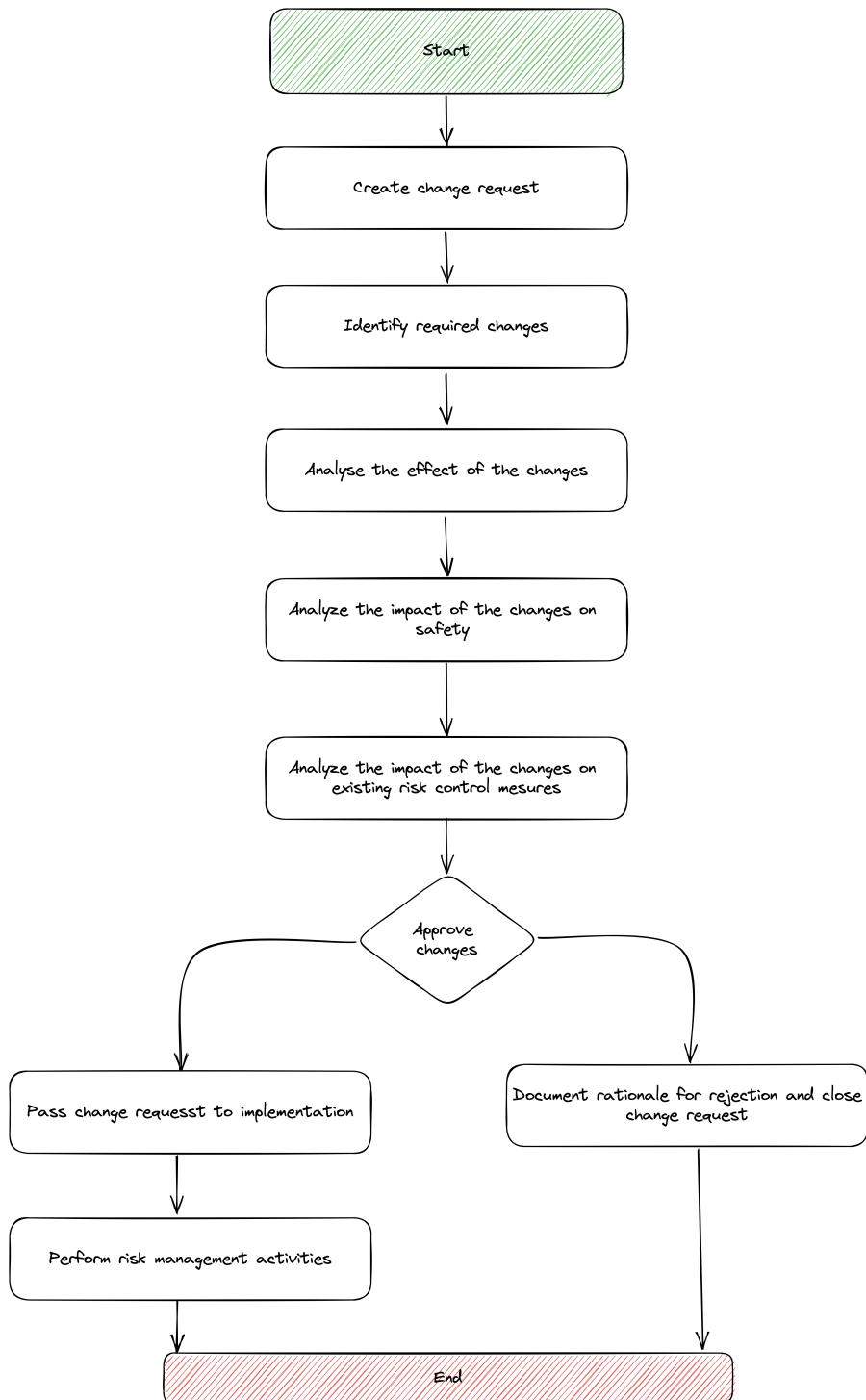


Figure 3: Change management process
14

5 Stakeholder requirements

Table 3: List of Stakeholders and their abbreviation

ID	Stakeholder	Description
CUPT	Customers (Casual Users and Permanent Tenants)	They are the primary stakeholders who will use the parking system and benefit from its features. Casual users will expect a hassle-free parking experience with transparent billing and adequate parking space availability, while permanent tenants will expect a personalized parking solution with timely billing and automated payment options.
PGO	Parking Garage Owners	They are the ones who will own and operate the parking garages. They will expect a reliable IT system that can cater to the diverse needs of their customers and offer timely billing and revenue generation. They may also expect a user-friendly interface to manage parking space allocation and rates.
IDSA	IT Developers and System Administrators	They are responsible for developing and maintaining the parking system. They will expect a clear understanding of the stakeholders' requirements, access to the necessary resources, and a flexible system architecture to accommodate future changes in the parking industry.
MAR	Municipal Authorities and Regulators	They are responsible for ensuring compliance with local regulations, safety standards, and environmental norms. They will expect the parking system to follow the applicable rules and regulations, maintain accurate records of parking transactions, and provide transparent billing information to customers and regulators.
FPSP	Financial Institutions and Payment Service Providers	They are responsible for processing payments and ensuring the security of financial transactions. They will expect the parking system to integrate with their payment processing platforms, offer secure payment options, and maintain accurate records of financial transactions.

6 System requirements

6.1 Functional requirements

Table 4: Functional requirements

ID	Trace from	Description
R101	PGO, IDSA	The IT system must allow for the configuration of parking garages individually, with the number of floors and parking spaces per floor being freely definable.
R102	PGO, IDSA	The number of parking spaces per floor may vary and can be set independently of other floors.
R103	PGO, IDSA	Each parking garage must have exactly one entrance and one exit barrier.
R104	CUPT, PGO	There are two categories of customers for each parking garage - Casual users and permanent tenants.
R105	CUPT, PGO	Permanent tenants are assigned a fixed parking space and pay a monthly rent for it.
R106	CUPT, PGO	If a permanent tenant's rent is not paid on the 15th of each month, their access is blocked until payment is made.
R107	CUPT	Occasional users can generate a digital parking ticket by pressing a button on the entrance barrier, which contains at least the date and time of entry, floor number, and assigned parking space number.
R108	CUPT	Permanent tenants must enter a unique personal code at the entrance barrier.
R109	CUPT, PGO	The date and time of entry are registered for both permanent tenants and occasional users for internal billing purposes.
R110	CUPT	The parking ticket for occasional users is validated before exit.
R111	CUPT	The date and time of exit are recorded.
R112	CUPT	The amount owed is calculated for occasional users based on the length of stay and the parking tariff.
R113	CUPT	When leaving the parking garage, either the unique personal code (permanent tenant) or the ticket number (occasional user) is read or entered at the exit barrier.
R114	CUPT, PGO, MAR	All entry and exit times are logged for both permanent tenants and occasional users.
R115	PGO, IDSA	The system must be able to generate an evaluation per parking garage and user category over a defined time period at any time.
R116	PGO, IDSA	Parking rates must be flexible and vary depending on the time of day.

ID	Trace from	Description
R117	PGO, IDSA	A separate tariff applies for weekends and public holidays.
R118	PGO, IDSA	The parking tariff calculation is made on a quarter-hour basis, with the tariff at the beginning of the respective quarter-hour applying for the entire quarter-hour.
R119	CUPT	Any quarter of an hour that has elapsed will be charged in full.
R120	CUPT	If the parking time is longer than 24 hours, the billing will automatically switch to daily flat rate, charging the full amount for the days that have elapsed.
R121	CUPT	A graphical representation of the different floors per parking garage must be available on the application, showing the free and occupied parking spaces, as well as those of the permanent tenants.
R122	CUPT, IDSA	The allocation of parking spaces for occasional users must be based on an algorithm, which aims at a balanced distribution.
R123	PGO	The application must be able to calculate and display the turnover of each parking lot (divided by customer category) for a given month at any time.
R124	PGO	The application must also be able to calculate and display the annual turnover, showing the individual months separately.

6.2 Non-Functional requirements

Table 5: Non-Functional requirements

ID	Trace from	Description
NFR101	CUPT, PGO, IDSA	The system must be easy to use and navigate for both occasional users and permanent tenants.
NFR102	CUPT, PGO, IDSA	The system must be secure, ensuring that personal data is protected.
NFR103	CUPT, PGO, IDSA	The system must be reliable and available at all times, with minimal downtime for maintenance.
NFR104	CUPT, PGO, IDSA	The application must have a responsive design and work on multiple devices and platforms.
NFR105	CUPT, PGO, IDSA	The system must be scalable and able to accommodate future growth.
NFR106	CUPT, PGO, IDSA, MAR, FPSP	The IT system must be compliant with relevant laws and regulations, including data privacy and protection regulations.

ID	Trace from	Description
NFR107	CUPT, PGO, IDSA	The application must have a user-friendly interface with clear instructions on how to use the system and its features.
NFR108	IDSA	The IT system must be compatible with existing software and hardware systems used by the company.
NFR109	IDSA	The system must have a backup and recovery plan in case of system failure or data loss.

7 System architecture and design

The new parking garage software for ParkingTown is a web-based SvelteKit application. ([svelte, 2023a](#)) The Application will be delivered as a state-of-the-art OCI-compliant container image. ([OCI, 2023](#)) Database access is handled by the ORM ([wikipedia, 2023a](#)) prisma ([prisma, 2023](#))

Prisma allows a wide variety of SQL-based database providers.

7.1 Svelte / SvelteKit

7.1.1 What is Svelte

Svelte is a front-end JavaScript framework that allows developers to build web applications using reactive components. It was created by Rich Harris in 2016 and has since gained popularity due to its unique approach to building web applications. Unlike other front-end frameworks that rely on a virtual DOM, Svelte compiles your code into highly optimized and efficient JavaScript code at build time.

In Svelte, components are defined using a syntax similar to HTML and CSS, making it easy for developers to create reusable and reactive UI elements. Svelte's compiler then generates efficient code that updates the DOM directly, resulting in fast and smooth user experiences.

Svelte provides several built-in features that make it easier for developers to build web applications. These features include:

- Reactivity: Svelte provides a reactive system that allows components to update automatically when data changes.
- Scoped CSS: Svelte allows developers to define CSS styles that are scoped to a specific component, reducing the risk of CSS conflicts.
- Animations: Svelte provides an easy-to-use API for adding animations to your components.
- Event handling: Svelte provides an intuitive syntax for handling events and user interactions.

([wikipedia, 2023c](#))

7.1.2 What is SvelteKit

SvelteKit is a high-performance web application framework built on top of Svelte. It was released in 2021 and provides several built-in features that make it easier to build server-rendered web applications. SvelteKit is designed to create highly optimized web applications with minimal overhead.

SvelteKit provides several key features such as:

- Server-side rendering: SvelteKit allows you to render components on the server, reducing the initial load time of your web application.
- Automatic code splitting: SvelteKit automatically splits your code into smaller chunks, reducing the amount of code that needs to be downloaded by the client.
- File-based routing: SvelteKit allows you to define routes for your web application using a file-based routing system, making it easier to organize your code and routes.
- API routes: SvelteKit provides an easy-to-use API for defining server-side routes and interacting with databases and other services.
- Integration with other tools: SvelteKit integrates with popular tools like TypeScript, GraphQL, and Tailwind CSS, making it easier to build complex web applications.

SvelteKit builds on the strengths of Svelte and provides a powerful and efficient framework for building server-rendered web applications.

([svelte, 2023b](#))

7.1.3 Why Sveltekit instead of other full-stack javascript frameworks

One of the main reasons SvelteKit was chosen for this project is, that I wanted to learn this framework for a long time. This project was a perfect learning opportunity to get hands-on experience with a new up-and-coming javascript framework.

According to the biggest javascript survey, Svelte / Sveltekit is amongst the most loved frameworks. ([StateOfJS, 2023](#))

Some of the reasons why SvelteKit is so loved are:

1. **Performance** One of the primary benefits of using SvelteKit is its performance. SvelteKit is designed to create highly optimized web applications with minimal overhead. It does this by using a unique feature called the “Svelte Compiler” that compiles your code during the build process, resulting in highly optimized and lightweight code.
2. **Easy to learn and use** SvelteKit is relatively easy to learn and use, especially if you are already familiar with modern JavaScript frameworks like React or Vue. SvelteKit provides an intuitive and straightforward syntax that makes it easy to write and understand code.
3. **Built-in features and integrations** SvelteKit comes with several built-in features and integrations, making it easier to build web applications quickly. It includes

features such as server-side rendering, automatic code splitting, file-based routing, and more. Additionally, it integrates with popular tools like Tailwind CSS, TypeScript, and GraphQL.

4. **Flexibility** SvelteKit is highly flexible and customizable, making it an excellent choice for a wide range of projects. Whether you are building a simple blog or a complex e-commerce website, SvelteKit can adapt to your needs and requirements.
5. **Great community support** SvelteKit has a great community that is continually improving and contributing to the framework. The community provides excellent documentation, resources, and support to help you get started and overcome any challenges you may encounter.

7.2 Containers

Containers are a method of software virtualization that allows you to package an application and all its dependencies into a single lightweight package, which can be deployed and run consistently across different computing environments.

7.2.1 What are Containers

To understand containers, it's helpful to first understand traditional virtualization, which involves running multiple operating systems on a single physical machine. With virtualization, each operating system has its own set of resources (like memory and CPU), and the hypervisor (a layer of software that sits between the operating system and the physical hardware) manages the allocation of those resources.

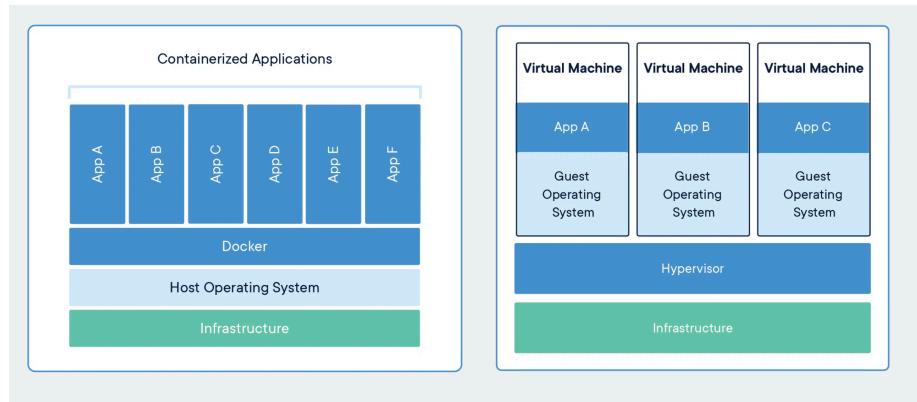


Figure 4: Comparing Docker to VM's

Containers, on the other hand, are a form of operating-system-level virtualization, which means they share the same kernel as the host operating system. This makes them

much more lightweight and portable than traditional virtual machines, since they don't require a separate operating system to be installed and managed for each container.

([docker, 2023b](#))

Instead, containers use a combination of file system isolation and process isolation to provide a separate environment for each application. Each container has its own file system that is isolated from the host file system, and each container runs in its own process space, separate from other containers on the same host.

Containers can be created from a base image (like an operating system or a programming language runtime) and customized with additional software and configurations specific to your application. Once the container is built, it can be deployed to any host that supports the container runtime (like Docker or Kubernetes).

Some of the benefits of using containers include:

- **Portability:** Containers can be run on any host that supports the container runtime, making them a great option for deploying applications across different environments (like development, staging, and production).
- **Consistency:** Since each container contains all the dependencies required to run the application, you can be sure that the application will run consistently across different hosts.
- **Scalability:** Containers can be easily scaled up or down to meet changing demand since they are lightweight and don't require a lot of resources.
- **Isolation:** Since each container runs in its own process space, containers provide a degree of isolation between different applications, reducing the risk of conflicts or security vulnerabilities.

([wikipedia, 2023b](#))

7.2.2 OCI Containers

The OCI is a collaborative project hosted by the Linux Foundation, to create open standards for container formats and runtime. The OCI was founded in 2015 by Docker and other industry leaders, in response to a growing need for interoperability and standardization in the container ecosystem.

The OCI is focused on two main specifications: the Image Format Specification and the Runtime Specification. These specifications define how containers should be packaged, distributed, and run, and are designed to be vendor-neutral and interoperable.

The Image Format Specification defines a standard format for container images, which is used to package an application and its dependencies into a single distributable artifact. This specification includes details like how images should be structured, how layers should be defined and represented, and how metadata should be included.

The Runtime Specification defines a standard interface between the container image and the host operating system, which allows the container to be run in a consistent and portable way across different environments. This specification includes details like

how the container should be started and stopped, how it should interact with the host filesystem and network, and how resources like CPU and memory should be managed.

Both specifications are designed to be flexible and extensible, allowing for innovation and customization while maintaining interoperability. The OCI also includes tools and utilities for working with container images and runtime, like the OCI Image Tool and the OCI Runtime Tool.

One of the main goals of the OCI is to foster collaboration and interoperability in the container ecosystem. By creating open standards that are supported by a broad community of vendors and developers, the OCI aims to reduce fragmentation and increase the adoption of containers as a standard way to package, distribute, and run applications.

([Foundation, 2023](#))

7.2.3 Docker

Docker is a containerization platform that allows you to package an application and its dependencies into a single portable unit called a container. Docker is built on top of the Linux kernel's containerization features, like cgroups and namespaces, and provides a simple and user-friendly interface for working with containers.

Docker containers are OCI compliant and therefore can run on any Container runtime that supports OCI Images like Kubernetes or Openshift.

Docker is the facto standard for building and running containers on a development machine. Docker is available for Linux (native) or as a desktop application called Docker Desktop.

Docker Desktop is a desktop application for Windows and macOS that provides an easy-to-use interface for working with containers. It includes all the tools and services needed to develop, build, and deploy containerized applications on a local machine.

([docker, 2023a](#))

7.2.3.1 When to use Docker Docker at present is the most widely used container runtime/platform. The Docker Platform is easy to use and comes with a feature-rich GUI. Docker is mostly used in development setups. Docker was one of the first implementations of Linux cgroup containers under windows and contributed with this massively to the success of containerization.

With the majority of containers and container orchestration tools like Kubernetes rising in popularity, docker is no longer the industry leader for building, packaging and running production-grade containers.

Some of the reasons are:

- **Kubernetes:** Kubernetes dropped the docker runtime support in Kubernetes version v1.24 ([kubernetes, 2023a](#)) (Docker containers still run on Kubernetes since they are OCI compliant)

- **Develop close to production:** With most containers running in a Kubernetes Cluster developers started using local Kubernetes clusters like minikube ([Kubernetes, 2023b](#)) or rancher desktop ([SUSE, 2023](#)) as development environments. This enables the developers to be as close to the production infrastructure as possible.
- **Docker in Docker:** In order to build docker images from inside docker (for example CI pipeline) the docker demon is required to run as root. This is a potential security risk for the toolchain and it's recommended to use different tools like kaniko ([Google, 2023](#)) to build your production container images.

7.3 Prisma

Prisma is an open-source tool that simplifies database access for developers. It provides a type-safe and scalable ORM (Object-Relational Mapping) layer that enables developers to interact with their database more intuitively and efficiently.

With Prisma, developers can define their data models in a declarative schema language called Prisma Schema. This schema defines the structure of the database and the relationships between tables. Prisma Schema also supports various data types, such as strings, numbers, booleans, and timestamps.

Prisma generates a set of CRUD (Create, Read, Update, Delete) operations based on the defined schema. These operations are generated as TypeScript or JavaScript functions that can be called directly from the application code. This removes the need to write SQL queries manually and also makes the codebase more maintainable and readable.

Prisma is designed to work with multiple database engines such as MySQL, PostgreSQL, MongoDB and other SQL-based databases. With the abstraction provided by Prisma switching between different databases is achievable without having to rewrite their application code.

Prisma also provides advanced features such as data validation, query optimization, and transaction management. These features ensure that the data stored in the database is consistent and secure.

([prisma, 2023](#))

7.3.1 ORM

An ORM (Object-Relational Mapping) is a technique used to map data between an application's object-oriented programming language and a relational database. In other words, an ORM acts as a bridge between the application and the database, providing a layer of abstraction that makes it easier for developers to interact with the database.

The basic idea behind an ORM is to represent database tables as classes in the programming language used by the application. Each row in the table is then represented as an instance of the class, and the columns in the table are represented as properties of the class. This allows developers to interact with the database using familiar object-oriented programming concepts, such as objects, methods, and properties.

One of the primary benefits of using an ORM is that it can simplify the process of interacting with the database. Rather than writing SQL queries directly, developers can use the ORM to perform CRUD (creating, reading, updating, and deleting) database operations. These operations are typically performed using methods provided by the ORM, rather than by writing SQL queries manually.

Another benefit of using an ORM is that it can make the application code more maintainable and readable. By abstracting away the details of how the data is stored in the database, the ORM can make the application code more focused on the business logic of the application, rather than on the details of how the data is stored.

([wikipedia, 2023a](#))

7.3.2 PostgreSQL

PostgreSQL is an open-source object-relational database management system (ORDBMS) that is designed to store, manage, and manipulate large amounts of data.

PostgreSQL is known for its ability to handle complex transactions and data structures, making it ideal for use in applications that require high levels of reliability and concurrency. It uses a technique called multi-version concurrency control (MVCC) to allow multiple users to access the database simultaneously without conflicts. This technique ensures that data consistency is maintained even when multiple users are making changes to the database concurrently. MVCC allows PostgreSQL to handle complex transactions and large amounts of data with ease, making it suitable for a wide range of applications.

One of the main advantages of PostgreSQL is its extensibility. It provides support for user-defined functions, triggers, and custom data types, allowing developers to extend the database to meet their specific needs.

PostgreSQL's support for advanced indexing and querying is another key strength. It provides a variety of indexing options, including B-tree, hash, GiST, and SP-GiST, as well as support for full-text search and geospatial queries. PostgreSQL's indexing and querying capabilities are highly optimized, allowing for efficient retrieval and analysis of large amounts of data. This makes it suitable for use in applications that require complex querying and data analysis, such as business intelligence and data warehousing.

PostgreSQL is also known for its focus on data security and reliability. It provides robust authentication and encryption mechanisms, as well as support for backup and recovery operations. PostgreSQL's focus on data security and reliability makes it a popular choice for applications that require high levels of data security and availability, such as healthcare systems and government agencies.

([Postgres, 2023](#))

7.4 ERD

The ERD represents a database schema for a parking garage management system. It consists of several entities, namely ParkingGarage, Level, ParkingSpace, Customer, ParkingTicket, ExitTicket, and ParkingRate, and the relationships between them.

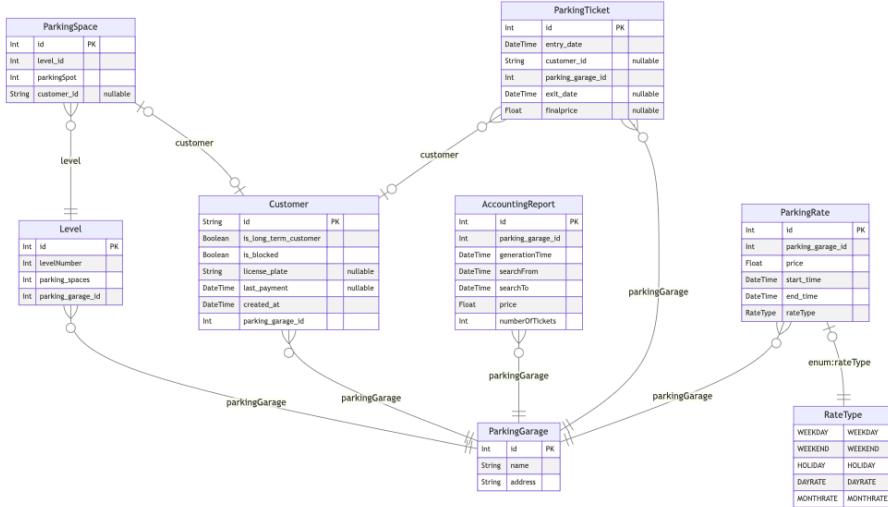


Figure 5: ERD

The **ParkingGarage** entity represents a parking garage, which has a unique identifier (id), a name, and an address. A parking garage can have multiple levels, represented by the **Level** entity.

Each level has a unique identifier (id), a level number, and the number of parking spaces available on that level. Additionally, each level belongs to a specific parking garage, represented by the foreign key **parking_garage_id**, which references the id of the **ParkingGarage** entity.

The **ParkingSpace** entity represents a single parking space within a level. Each parking space has a unique identifier (id), a reference to the level it belongs to (**level_id**), and a parking spot number. Additionally, each parking space can be assigned to a customer, represented by the **customer_id** foreign key, which references the id of the **Customer** entity. The **ParkingSpace** entity only represents occupied spaces.

The **Customer** entity represents a customer who can use the parking garage. Each cus-

tomer has a universally unique identifier (uuid) and can be marked as a long-term customer or blocked from using the parking garage. Additionally, each customer is associated with a specific parking garage, represented by the `parking_garage_id` foreign key, which references the `id` of the `ParkingGarage` entity.

The `ParkingTicket` entity represents a ticket that a customer receives upon entering the parking garage. Each ticket has a unique identifier (`id`), an entry date and time, an exit date and time, a price, a reference to the customer who received the ticket, and a reference to the parking garage in which the ticket was issued. The `customer_id` and `parking_garage_id` foreign keys reference the `id` of the `Customer` and `ParkingGarage` entities, respectively.

The `ParkingRate` entity represents the hourly rate for parking in a specific parking garage during a specific time range. Each rate has a unique identifier (`id`), a reference to the parking garage it applies to, a price per hour, a start time, and an end time. The `parking_garage_id` foreign key references the `id` of the `ParkingGarage` entity. For the prototype, only specific hardcoded time ranges are supported

8 Implementation

For this project, it was important to have the git repository as a single source of truth. The git repository keeps this entire documentation as well. The following sections are often code snippets used but they can all be found in completion on [GitHub](#)

The Application is hosted [here](#)

To keep the code snippets shown in the subsections readable, the HTML part (+page.svelte) is most of the time not included because HTML tends to become large skeletons of boilerplate code. If a specific piece of code is not shown kindly refer to the folder [structure](#) explanation to find the code fast on GitHub.

8.1 Documentation workflow

In the Company, I work we create SaMD (Software as Medical Device). Software that classifies as a medical device needs to be developed with specific regulations in mind and requires following a specific process. At present we use Microsoft Word to write the required documentation but this raises a few challenges.

1. Word is not as easy to generate from within a pipeline (For example openAPI specification)
2. Word is not as easy to Lint inside a pipeline
3. Including Code or Project specific references in good looking format is not possible
4. Word is slow and the formatting is tedious

With this Project, I wanted to try if it's possible to write large project documentation using Markdown and export the written documentation in a professional-looking PDF. To achieve this the following software is used:

1. **Pandoc:** used to convert the markdown into LaTex format and print a PDF ([Pandoc, 2023](#))

```
brew install pandoc
```

2. **LaTeX:**** Pandoc requires LaTeX to be installed to generate the PDF ([LaTeX, 2023](#))

```
brew install --cask mactex
```

8.1.1 Folder Structure

To generate an academic-looking paper the following folder structure is used

```
latex-pdf
|assets
| -citation-style.csl
|-build
| -output_print.pdf
|-content
| -images
| -bibliography.bib
| -index.md
| -disable_float.tex
|-layouts
| -print.tex
|-Makefile
|-build.sh
```

- **assets:** Used for third-party styles, in this case how to style the citations
- **build:** The output directory where the PDF will be generated to
- **content** Holds the full markdown file, all used figures and the bibliography as well as additional LaTeX configuration
- **layouts:** Pandoc can convert the markdown into different output formats, in the layout folder could the different formats specify some individual layouts

8.1.2 Buildscript

The script build.sh is handling the generation of the PDF file.

```
#!/bin/bash
CONTENTDIR="content"
BUILDDIR="build"
FILENAME="index"
ASSETSDIR="assets"

download_csl() {
    mkdir "${ASSETSDIR}" -p
    wget -O "${ASSETSDIR}/citation-style.csl" \
        "https://raw.githubusercontent.com/citation-style-language \
        /styles/master/harvard-anglia-ruskin-university.csl"
}

pdf_print() {
    mkdir "${BUILDDIR}"
    echo "Creating pdf-print output"
    pandoc "${CONTENTDIR}/${FILENAME}.md" \
        -H disable_float.tex \
        --resource-path="${CONTENTDIR}" \
        --citeproc \
        --csl="${ASSETSDIR}/citation-style.csl" \
        --from="markdown+tex_math_single_backslash+tex_math_dollars+raw_tex" \
        --to="latex" \
        --output="${BUILDDIR}/output_print.pdf" \
        --pdf-engine="xelatex" \
        --include-in-header="layouts/print.tex"
}
# Example: `./build.sh pdf_print`
$*
```

The Script defines four Variables:

1. CONTENTDIR: the path to the directory that holds all figures and the markdown file
2. BUILDDIR: the path to the output directory of this script
3. FILENAME: the filename of the markdown file
4. ASSETSDIR: the path to the asset directory that holds the csl

To run the script run the following command from the latex-pdf directory:

```
chmod +x build.sh #allow the script to be executed
./build.sh pdf_print #run the script
```

8.1.3 Index.md

The `index.md` is the markdown file that is specified in the `build.sh` file to convert into PDF.

Pandoc uses a yaml-based syntax in the header to configure the default conversion behavior.

The following configuration was made

```
title: "Parking Garage"
date: "May 2023"
author: "David Albrecht, TEKO Bern"
numbersections: true
toc: true
lof: true
lot: true
mainfont: Times New Roman
bibliography: "bibliography.bib"
link-citations: true
urlcolor: "blue"
```

1. **title:** Set the Document title on the cover sheet
2. **date:** Set the Date on the cover sheet
3. **author:** Sets the author on the cover sheet
4. **numbersections:** Enables the numbering of the headings
5. **toc:** Creates a table of content at the beginning of the document
6. **lof:** Creates a list of figures at the beginning of the document
7. **lot:** Creates a list of tables at the beginning of the document
8. **mainfont:** Sets the font for the entire document
9. **bibliography:** Path to the bibliography (contains the details of external sources)
10. **link-citations** Enables to click a link and follow its reference
11. **urlcolor:** Sets the color of all URL's in the entire document

8.2 Development environment

For a local development environment, the following off-the-shelf software is used:

1. **Visual Studio Code:** VS code is used as the primary code editor
2. **nvm:** nvm is used to manage the specific node versions
3. **Nodejs:** JavaScript runtime
4. **Docker:** For PostgreSQL database

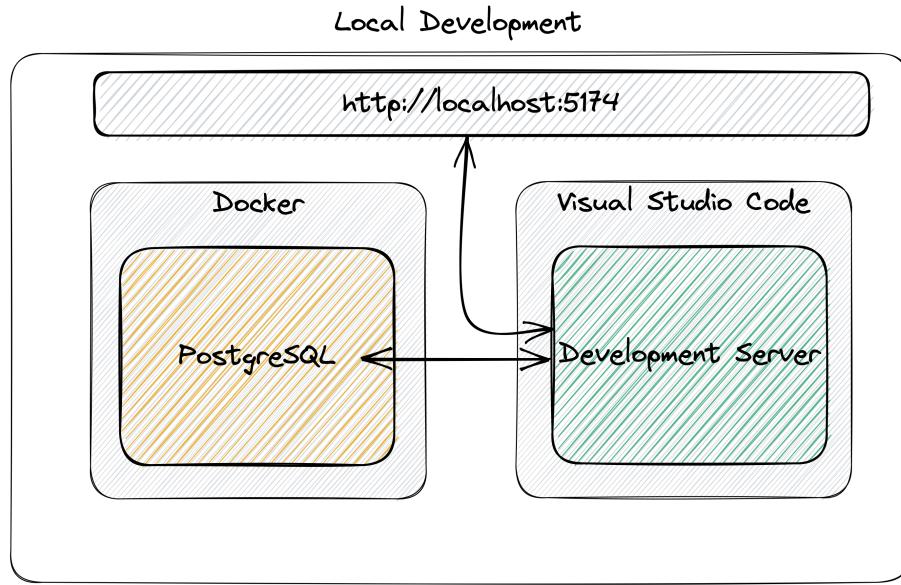


Figure 6: local development environment

8.2.1 Setup Development environment

The project is developed on a MacBook m2 Pro. The installation guide for the development setup is therefore written for apple silicon but all used software components are available for windows and Linux.

1. Install [Visual Studio Code](#) ([Microsoft, 2023](#))
2. Install nvm ([nvm, 2023](#))


```
brew install nvm
```
3. Install Nodejs


```
nvm install --lts
```
4. Install [Docker Desktop](#)
5. Clone GitHub Repository


```
git clone git@github.com:DaAlbrecht/parking-garage.git
cd parking-garage/
```
6. Create .env file for environment variables


```
export USERNAME=<USERNAME>
export PASSWORD=<PASSWORD>
cat > .env << EOF
DATABASE_URL=postgres://$USERNAME:$PASSWORD@localhost:5432/garage
```

7. Start the database container

```
docker-compose up-d
```
8. Install js dependencies

```
npm install
```
9. Create Prisma client and push schema to database

```
npx prisma generate
npx prisma db push
```
10. Start Sveltekit Application

```
npm run dev
```

8.2.2 Cloud development environments (CDEs)

The project is set up to support Cloud development environments. GitHub CodeSpaces ([GitHub, 2023](#)) as well as Gitpod ([Gitpod, 2023](#)) work. For Gitpod some additional configuration is added to the repository to have a better out of the box experience.

Gitpod allows the configuration of the workspace through a `.gitpod.yml` file. The following configuration is added additionally

```
tasks:
  - init: npm install && npm run build
    command: docker-compose up -d && npm run dev
  1. Install all node node modules
  2. start docker containers (PostgreSQL database)
  3. start the svelte application
```

8.3 Project folder structure

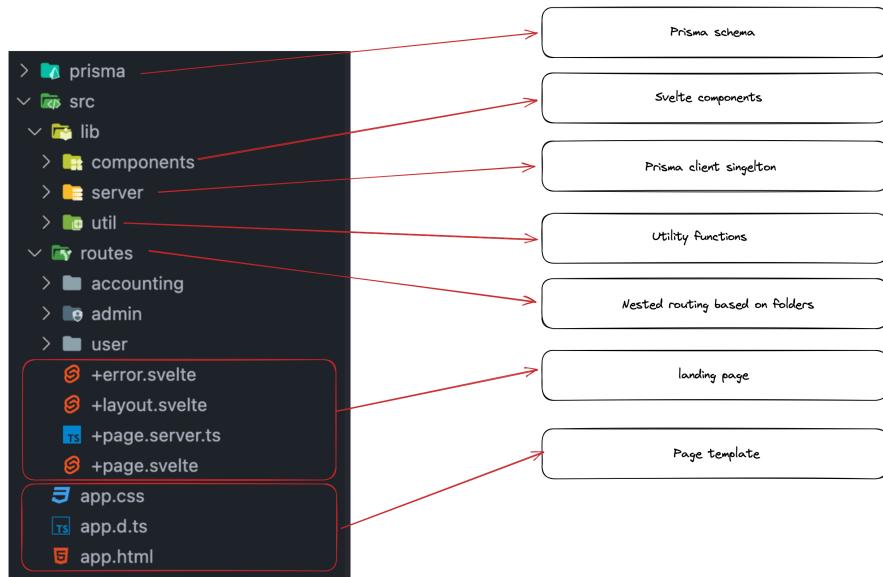


Figure 7: Project folder structure

For a more detailed description check the [SvelteKit documentation](#)

8.4 Prisma Schema

The `schema.prisma` file describes the configuration and model of the database.

The first part of the schema file is responsible for handling the connection to the database. It requires two parameters

1. **provider:** The type of database used
2. **url:** The connection string in the format:

`postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA`

Here's a short explanation of each component from ([Prisma, 2023](#)):

- **USER:** The name of your database user
- **PASSWORD:** The password for your database user
- **HOST:** The name of your hostname (for the local environment, it is localhost)
- **PORT:** The port where your database server is running (typically 5432 for PostgreSQL)
- **DATABASE:** The name of the database
- **SCHEMA:** The name of the schema inside the database

Instead of hardcoding the connection string, the value gets read out from the environment variable

```
datasource db {  
    provider = "postgresql"  
    url      = env("DATABASE_URL")  
}
```

The next section is responsible for generating the Prisma client

```
generator client {  
    provider = "prisma-client-js"  
}
```

Additionally, on schema creation, an erd diagram with the mermaid syntax should be created

```
generator erd {  
    provider          = "prisma-erd-generator"  
    output            = ".../documentation/erd.md"  
    includeRelationFromFields = true  
}
```

8.4.1 Models

After these configuration entries the models according to [ERD](#)

8.4.1.1 ParkingGarage

```
model ParkingGarage {  
    id           Int          @id @default(autoincrement())  
    name         String  
    address      String  
    levels       Level[]  
    parkingRates ParkingRate[]  
    customers    Customer[]  
    accountingReports AccountingReport[]  
    ParkingTicket ParkingTicket[]  
}
```

1. **id:** this is the primary key and is marked to be automatically created/incremented
2. **name:** the name of the garage
3. **levels:** One-to-many (1-n) relation with the record “Level”
4. **parkingRates:** One-to-many (1-n) relation with the record “ParkingRate”
5. **customers:** One-to-many (1-n) relation with the record “Customer”
6. **accountingReports:** One-to-many (1-n) relation with the record “AccountingReport”
7. **ParkingTicket:** One-to-many (1-n) relation with the record “ParkingTicket”

8.4.1.2 Level

```
model Level {
    id              Int          @id @default(autoincrement())
    levelNumber     Int
    parking_spaces  Int
    parkingGarage   ParkingGarage @relation(fields: [parking_garage_id], references: [id],
    parking_garage_id Int
    occupiedSpaces  ParkingSpace[]

    @@unique([parking_garage_id, levelNumber])
}
```

1. **id**: this is the primary key and is marked to be automatically created/incremented
2. **levelNumber**: floor number of the Level
3. **parking_spaces**: the number of parking spaces this level can hold
4. **parkingGarage**: one-to-many (1-n) relation with the record “ParkingGarage”
5. **parking_garage_id**: the foreign key to the ParkingGarage record
6. **occupiedSpaces**: one-to-many (1-n) relation with the record “ParkingSpace”

A parking garage can not have a level twice, to ensure this Prisma uses the following syntax:

```
@@unique([parking_garage_id, levelNumber])
```

This ensures that the combination of the attributes `parking_garage_id` and `levelNumber` are unique

8.4.1.3 ParkingSpace

```
model ParkingSpace {
    id              Int          @id @default(autoincrement())
    level          Level        @relation(fields: [level_id], references: [id], onDelete: Cascade)
    level_id       Int
    parkingSpot    Int
    customer       Customer?  @relation(fields: [customer_id], references: [id], onDelete: SetNull)
    customer_id    String?    @unique

    @@unique([level_id, parkingSpot])
}
```

1. **id**: this is the primary key and is marked to be automatically created/incremented
2. **level**: one-to-one (1-1) relation with the Level record
3. **level_id**: the foreign key to the Level record
4. **parkingSpot**: the number of the spot
5. **customer**: one-to-one (1-1) relation with the Customer record
6. **customer_id**: the foreign key to the Customer record

The model `ParkingSpace` represents an occupied parking space. If there are no parking spots occupied, then this table is empty.

One `ParkingSpace` can not have the same combination of the level and parking spot twice. This prevents them from trying to occupy the same spot twice at the same time.

8.4.1.4 Customer

```
model Customer {  
    id                  String      @id @default(uuid())  
    is_long_term_customer Boolean  
    is_blocked           Boolean  
    license_plate        String?  
    last_payment         DateTime?  
    created_at           DateTime     @default(now())  
    parkingSpace          ParkingSpace?  
    parkingTickets        ParkingTicket[]  
    parkingGarage         ParkingGarage  
    @relation(fields: [parking_garage_id], references: [id], onDelete: Cascade)  
    parking_garage_id     Int  
}
```

1. **id:** this is the primary key and is marked to be automatically created/incremented
2. **is_long_term_customer:** indicates if the user is a casual user or a permanent tenant
3. **is_blocked:** indicates if the user has paid the monthly fee if its a permanent tenant
4. **license_plate:** represents the license plate of an occasional customer
5. **last_payment:** the date, when a permanent tenant paid last.
6. **created_at:** gets set automatically when a new entry is created
7. **parkingSpace:** one-to-many (1-n) relation with the `ParkingSpace` record
8. **parkingTickets:** one-to-many (1-n) relation with the `ParkingTicket` record
9. **exitTickets:** one-to-many (1-n) relation with the `ExitTicket` record
10. **parkingGarages:** one-to-many (1-n) relation with the `ParkingGarage` record
11. **parking_garage_id:** the foreign key to the `ParkingGarage` record

Instead of using:

```
@id @default(autoincrement())
```

for indexing, the primary key, for the customer, a universally unique identifier (UUID) is used. A UUID is a 128-bit label.

When generated according to the standard methods, UUIDs are, for practical purposes, unique. Their uniqueness does not depend on a central registration authority or coordination between the parties generating them, unlike most other numbering schemes. ([wikipedia, 2023d](#))

The “randomness” of the UUID gives additional security and does not let other customers guess other customers’ IDs. This is crucial because the primary key is also used

for permanent tenants to open the entry and exit barrier.

8.4.1.5 AccountingReport

```
model AccountingReport {  
    id             Int          @id @default(autoincrement())  
    parkingGarage  ParkingGarage  
    @relation(fields: [parking_garage_id], references: [id], onDelete: Cascade)  
    parking_garage_id Int  
    generationTime   DateTime  
    searchFrom       DateTime  
    searchTo         DateTime  
    price            Float  
    numberoftickets  Int  
}
```

1. **id:** this is the primary key and is marked to be automatically created/incremented
2. **parkingGarage:** one-to-many (1-n) relation with the ParkingGarage record
3. **parking_garage_id:** the foreign key to the ParkingGarage record
4. **generationTime:** the date when the report is last created
5. **searchFrom:** the start time of the report search period
6. **searchTo:** the end time of the report search period
7. **price:** the final profit over the period
8. **numberoftickets:** the number of tickets found

8.4.1.6 ParkingTicket

```
model ParkingTicket {  
    id             Int          @id @default(autoincrement())  
    entry_date     DateTime  
    customer       Customer?  
    @relation(fields: [customer_id], references: [id], onDelete: SetNull)  
    customer_id    String?  
    parkingGarage  ParkingGarage  
    @relation(fields: [parking_garage_id], references: [id], onDelete: Cascade)  
    parking_garage_id Int  
    exit_date      DateTime?  
    finalprice     Float?  
}
```

1. **id:** this is the primary key and is marked to be automatically created/incremented
2. **entry_date:** the timestamp when a user entered the entry barrier.
3. **customer:** one-to-many (1-n) relation with the ParkingSpace record
4. **customer_id:** the foreign key to the Customer record
5. **parkingGarages:** one-to-many (1-n) relation with the ParkingGarage record
6. **parking_garage_id:** the foreign key to the ParkingGarage record
7. **exit_date:** gets set on checkout

8. **finalprice:** gets set on checkout

8.4.1.7 ParkingRate

```
model ParkingRate {
    id             Int      @id @default(autoincrement())
    parkingGarage  ParkingGarage
    @relation(fields: [parking_garage_id], references: [id], onDelete: Cascade)
    parking_garage_id Int
    price          Float
    start_time     DateTime
    end_time       DateTime
    rateType       RateType
}
```

1. **id:** this is the primary key and is marked to be automatically created/incremented
2. **parkingGarage:** One-to-many (1-n) relation with the ParkingGarage record
3. **parking_garage_id:** the foreign key to the ParkingGarage record
4. **price:** the hourly price
5. **start_time:** start time
6. **end_time:** end time
7. **rateType:** an enum, that represents different types like weekday,weekend etc

8.4.1.8 RateType

```
enum RateType {
    WEEKDAY
    WEEKEND
    HOLIDAY
    DAYRATE
    MONTHRATE
}
```

A simple enum for the different types.

8.5 Algorithm

When a new customer (permanent tenant with no fixed parking spot or an occasional customer) requests a parking spot the method `findEmptyParkingSpace` is called.

The method requires passing the garage, for which a parking space should be found to be passed as a function argument. The function returns a new parking space on the level that has the least percentage of occupied parking spots.

8.5.1 Implementation

```
export async function findEmptyParkingSpace(garage: ParkingGarage) {
  const levels = await prisma.level.findMany({
    where: {
      parking_garage_id: garage.id
    }
  });

  let lastOccupancy = 100;
  let leastOccupiedLevel: Level | null = null;
  for (const level of levels) {
    const current = await getOccupancyForLevel(level);
    if (current < lastOccupancy) {
      lastOccupancy = current;
      leastOccupiedLevel = level;
    }
  }
  if (!leastOccupiedLevel) return null;
  const parkingSpaces = await getAllParkingSpacesForLevel(leastOccupiedLevel);
  if (parkingSpaces.length === 0) return null;

  //find the first parking space that is not occupied
  const freeParkingSpace = parkingSpaces.find((parkingSpace) => !parkingSpace.occupied);

  if (!freeParkingSpace) return null;

  return freeParkingSpace;
}
```

The function `findEmptyParkingSpace` is responsible for handling the parking allocation. Let's break the function down.

First, the function gets all levels for this parking garage then loop over all levels and uses the utility function `getOccupancyForLevel` to get the occupancy for a specific level.

```

export async function getOccupancyForLevel(level: Level): Promise<number> {
  const occupiedParkingSpaces = await getOccupiedParkingSpacesForLevel(level);
  if (occupiedParkingSpaces.length === 0) return 0;
  return occupiedParkingSpaces.length / level.parking_spaces;
}

```

The function first gets all the occupied parking spaces for a specific level with another utility function and then divides the number of occupied parking spaces by the maximal number of parking spaces for this level.

```

async function getOccupiedParkingSpacesForLevel(level: Level): Promise<Array<number>> {
  const parkingSpaces = await prisma.parkingSpace.findMany({
    where: {
      level_id: level.id
    }
  });
  return parkingSpaces.map((parkingSpace) => parkingSpace.parkingSpot);
}

```

To get all occupied parking spaces a database query gets all entries in the `parkingSpace` table [ParkingSpace table](#). Afterward, the array gets filtered to only include the parking spot number.

Then in the `findEmptyParkingSpace` we have all the occupied parking spaces as well as the occupancy for each level. The next step is get all the parking spots for the least occupied level.

```
const parkingSpaces = await getAllParkingSpacesForLevel(leastOccupiedLevel);
```

The utility function getAllParkingSpacesForLevel returns an array containing information about each parking spot

```
export async function getAllParkingSpacesForLevel(level: Level) {
  const occupiedParkingSpaces = await getOccupiedParkingSpacesForLevel(level);
  const parkingSpaces = [];
  for (let i = 0; i < level.parking_spaces; i++) {
    if (occupiedParkingSpaces.includes(i)) {
      const parkingSpace = await prisma.parkingSpace.findFirst({
        where: {
          level_id: level.id,
          parkingSpot: i
        }
      });
      if (!parkingSpace) continue;
      if (parkingSpace.customer_id === null) continue;

      const customer = await prisma.customer.findFirst({
        where: {
          id: parkingSpace.customer_id
        }
      });
      if (!customer) continue;

      parkingSpaces.push({
        parkingSpot: i,
        occupied: true,
        level_id: level.id,
        permanentTenant: customer.is_long_term_customer
      });
    } else {
      parkingSpaces.push({
        parkingSpot: i,
        occupied: false,
        level_id: level.id,
        permanentTenant: false
      });
    }
  }
  return parkingSpaces;
}
```

- this function first gets all the occupied parking spots
- loop from 0 to the maximum number of parking spaces for this level
- push a new element to the array that includes the parking spot, if it's occupied and the corresponding level as well as if it's occupied by a permanent tenant.
- return the array

Afterward, the function `findEmptyParkingSpace` has all the needed information and can just get the next free element from the previously returned array.

```
if (parkingSpaces.length === 0) return null;

//find the first parking space that is not occupied
const freeParkingSpace = parkingSpaces.find((parkingSpace) => !parkingSpace.occupied);

if (!freeParkingSpace) return null;

return freeParkingSpace;
```

8.6 Customer UI

The customer UI is separated into two views

1. Entry barrier (Welcome page)
2. Exit barrier (Checkout)

8.6.1 Entry barrier

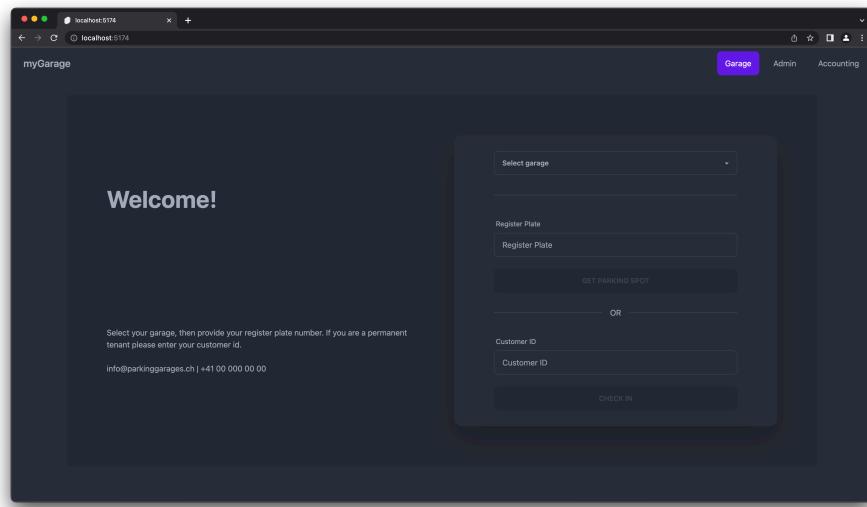


Figure 8: Entry barrier

This view simulates the entry barrier for a parking garage. First, a garage needs to be selected from the dropdown menu, then either the customer id (if the customer is a permanent tenant) or the license plate needs to be entered. If a license plate gets entered that is not currently occupying one of the parking spots, one of the free parking spots will be assigned according to the algorithm [implementation](#)

If a license plate or a customer id that is representing a permanent tenant, the user gets redirected to the [checkout](#)

8.6.1.1 Implementation To preload, the dropdown for the garages with information, the data gets rendered server side.

```
export const load = (async () => {
  const garages = await prisma.parkingGarage.findMany({
    where: {
      levels: {
        some: {}
      }
    }
  });
  return { garages: garages };
}) satisfies PageServerLoad;
```

In SvelteKit, the load function is used to do server-side rendering, on the client, the data can be accessed as following

```
<script lang="ts">
  import type { PageData } from './$types';
  import Login from '$lib/components/Login.svelte';

  export let data: PageData;
</script>
```

8.6.1.1 permanent tenant parking spot When a permanent tenant enters their customer id the following form gets sent

```
<form method="POST" action="/longTermCustomer" use:enhance>
  <input type="hidden" name="garage" value={selectedGarage} />
  <label class="label" for="id">
    <span class="label-text">Customer ID</span>
  </label>
  <input
    type="number"
    name="id"
    bind:value={id}
    placeholder="Customer ID"
    class="input-bordered input w-full"
  />
  <div class="form-control mt-6">
    <button disabled={!canLongTerm} class="btn-info btn">Check in</button>
  </div>
</form>
```

The form sends the form data to the `/longTermCustomer` function. This function is located in the `+page.server.ts` file.

```

longTermCustomer: async ({ request }) => {
  const data = await request.formData();

  const id = data.get('id');
  const garage = data.get('garage');

  if (!id) return fail(422, { error: 'Missing id' });
  if (!garage) return fail(422, { error: 'Missing parkingGarages' });

  const idString = id.toString();
  const garageNumber = Number(garage);

  const customer = await prisma.customer.findFirst({
    where: {
      id: idString,
      parking_garage_id: garageNumber
    }
  });

  if (!customer) return fail(422, { error: 'Customer does not exist' });

  if (customer.last_payment === null) return fail(422, { error: 'Customer has not paid yet' });

  const date = new Date();

  if (date.getDate() > 15 && customer.last_payment.getMonth() !== date.getMonth()) {
    await prisma.customer.update({
      where: {
        id: idString
      },
      data: {
        is_blocked: true
      }
    });
    return fail(422, { error: 'Customer is blocked' });
  }

  if (customer.is_blocked) {
    if (customer.last_payment.getMonth() == date.getMonth()) {
      await prisma.customer.update({
        where: {
          id: idString
        },
        data: {
          is_blocked: false
        }
      });
    }
  }
}

```

```

    });
} else {
    return fail(422, { error: 'Customer is blocked' });
}
}

getPermanentTenantParkingSpot(garageNumber, customer);

throw redirect(303, `/user/${customer.id}`);
}

```

This function checks if the send form data is complete and if a customer for the given id exists and that they are a permanent tenant. If the customer exists and they are not blocked, a utility method gets called to either check the customer to their corresponding parking spot, or if they do not yet have a fixed parking spot the **algorithm** to find new parking spaces is used to find an empty spot.

If the Customer exists, but has not yet paid the monthly fee and its already past the 15. of the month, the customer gets blocked. If the customer was blocked but has payed the fee the customer gets unblocked.

8.6.1.1.2 occasional user parking spot When an occasional user, enters their license plate, the following form gets sent

```
<form method="POST" action="/getParkingSpot" use:enhance>
  <input type="hidden" name="garage" value={selectedGarage} />
  <label class="label" for="id">
    <span class="label-text">Register Plate</span>
  </label>
  <input
    type="number"
    name="id"
    bind:value={registerPlate}
    placeholder="Register Plate"
    class="input-bordered input w-full"
  />
  <div class="form-control mt-6">
    <button disabled={!canGetSpot} class="btn-primary btn">Get parking spot</button>
  </div>
</form>
```

This form sends the form data to the `/getParkingSpot` function. This function is located in the `+page.server.ts` file.

The function occupies a new parking spot according to the `algorithm`

8.6.2 Checkout

The Checkout UI simulates the exit barrier. It's not only the exit barrier where you would put in the parking ticket, but also the machine, where you pay the parking fee.

The Checkout has separate views for permanent tenants and occasional users

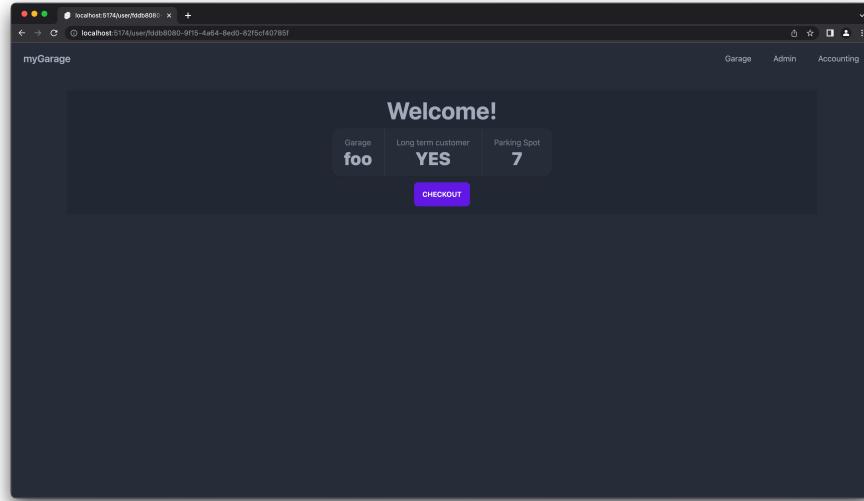


Figure 9: Exit barrier permanent tenants

For Permanent tenants, the view shows the garage, their user status and the parking spot.

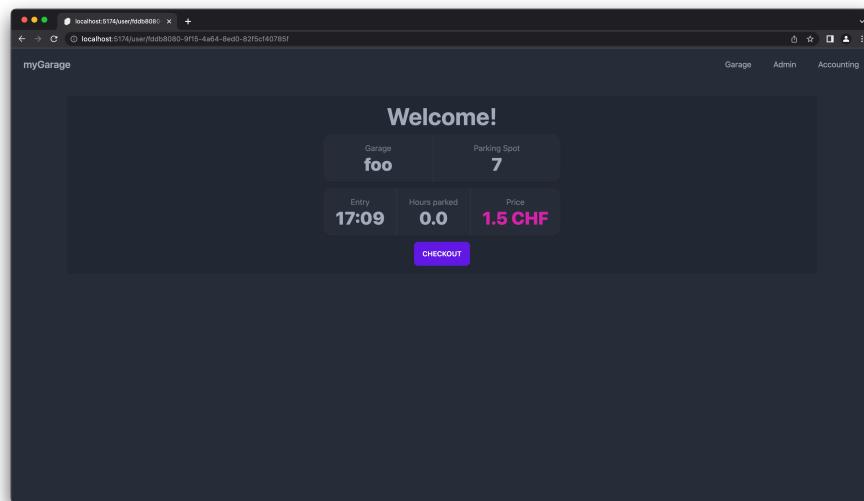


Figure 10: Exit barrier

The checkout view shows the current garage as well as the spot where the parked car is.

Additionally, the entry time and the current price is displayed.

If the checkout button is pressed, the exit barrier opens. For long-term customers nothing else happens, while for occasional users, the user in the database gets deleted and the parking spot entry as well and therefore freeing the parking spot.

8.6.2.1 Implementation

```
import { prisma } from '$lib/server/database';
import type { PageServerLoad } from './$types';
import { calculatePrice } from '$lib/util/accounting';

export const load = (async ({ params }) => {
  const customer = await prisma.customer.findFirst({
    where: {
      id: params.slug
    },
    include: {
      parkingGarage: true,
      parkingSpace: true,
      parkingTickets: true
    }
  });

  const price =
    customer && customer.is_long_term_customer === false
      ? await calculatePrice(customer.parkingTickets[0])
      : undefined;
  return { customer, price };
}) satisfies PageServerLoad;
```

The checkout view requires information from the `parkingGarage`, `parkingSpace` and `parkingTickets`. The information gets read out from the database and send to the client.

```
import { onDestroy } from 'svelte';
import { invalidateAll } from '$app/navigation';
import { enhance } from '$app/forms';

const interval = setInterval(invalidateAll, 1000 * 60 * 60);

onDestroy(() => {
  clearInterval(interval);
});
```

To trigger a rerender the built-it function from sveltekit `invalidateAll` is called periodically that triggers a rerender of the checkout component.

8.7 Admin UI

The admin UI is responsible for creating, updating and deleting garages as well as levels. The Admin UI is split into three different views.

1. **Garages overview:** this view lists all garages
2. **Garage edit:** the edit view allows for editing already created garages as well as adding additional levels and more parking spots
3. **Garage details:** Gives an overview of the garage, lists all parking spots, the occupancy and the estimated revenue for each level

8.7.1 Garage overview

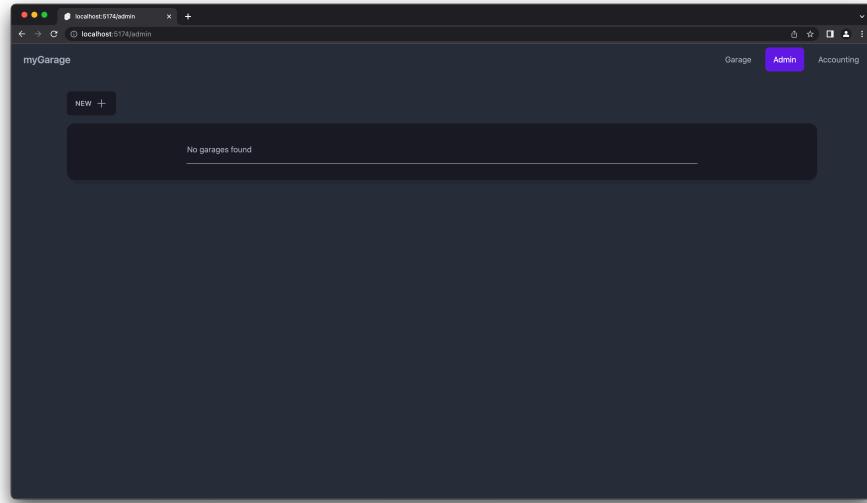


Figure 11: Garage overview

On startup, no garage exists. The overview list is empty.

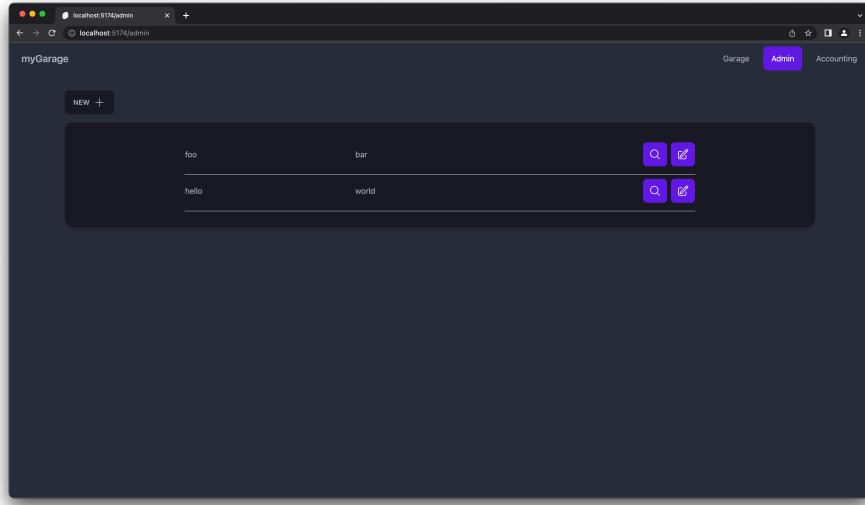


Figure 12: Overview with sample data

When a garage is added, this view shows a list of the garage's name and address. The following options exist:

1. Create new garage
2. Show details for a garage
3. Edit a garage

8.7.1.1 Implementation

```
import { prisma } from '$lib/server/database';
import type { PageServerLoad } from './$types';

export const load = (async () => {
  const garages = await prisma.parkingGarage.findMany();
  return { garages };
}) satisfies PageServerLoad;
```

The overview of the garages does only need to retrieve the information for all garages from the database.

8.7.2 Add garages

To add a new garage, it's required to fill in the name and the address of the garage.

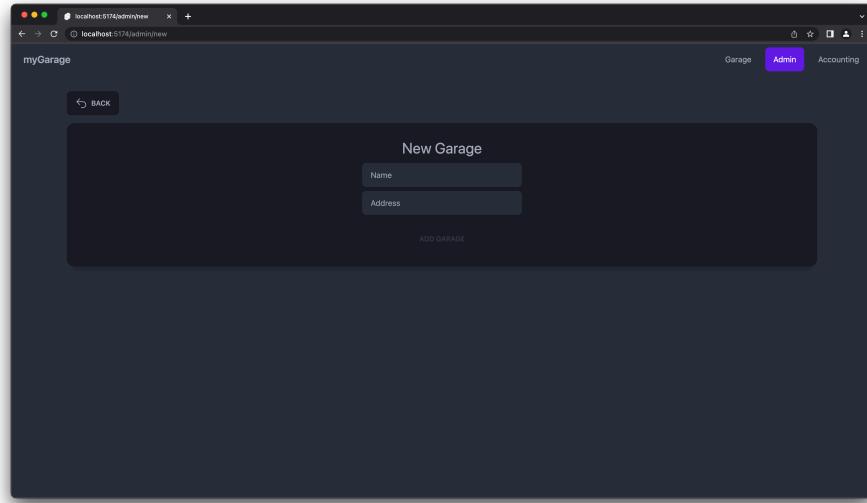


Figure 13: New garage form

After pressing the button, the user will be redirected to the [garage overview](#)

8.7.2.1 Implementation

```
<form method="POST" action="/createGarage" class="flex flex-col items-center gap-2">
  <input
    type="text"
    bind:value={name}
    name="name"
    placeholder="Name"
    class="input w-full max-w-xs"
  />
  <input
    type="text"
    bind:value={address}
    name="address"
    placeholder="Address"
    class="input w-full max-w-xs"
  />
  <button disabled={!name || !address} class="btn-success btn mt-4">add garage</button>
</form>
```

When creating a new garage, the following form data is sent to `/new/createGarage` and executes the following function on the server

```
createGarage: async ({ request }) => {
  const data = await request.formData();
  const name = data.get('name');
  const address = data.get('address');

  if (!name || !address) return fail(422, { error: 'Missing name or address' });

  const nameString = name.toString();
  const addressString = address.toString();

  try {
    const garage = await prisma.parkingGarage.create({
      data: {
        name: nameString,
        address: addressString
      }
    });
    await prisma.parkingRate.createMany({
      data: getInitialParkingRatesForGarage(garage.id)
    });
  } catch (error) {
    return fail(422, { error: 'Garage already exists' });
  }
  throw redirect(303, '/admin');
}
```

This function checks if the form data is complete, if so it creates a new entry in the database. After creating a new entry for a parking garage, a default set of parking rates get created for this parking garage.

Default set of rates

```
import type { ParkingRate } from '@prisma/client';

export const getInitialParkingRatesForGarage: (garageId: number) =>
  Omit<ParkingRate, 'id'>[] = (
    garageId
  ) => [
    {
      price: 1.5,
      parking_garage_id: garageId,
      start_time: new Date('2021-01-04T00:00:00.000Z'),
      end_time: new Date('2021-01-04T05:59:00.000Z'),
      rateType: 'WEEKDAY'
    },
  ]
```

```

{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-04T06:00:00.000Z'),
  end_time: new Date('2021-01-04T08:59:00.000Z'),
  rateType: 'WEEKDAY'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-04T09:00:00.000Z'),
  end_time: new Date('2021-01-04T17:59:00.000Z'),
  rateType: 'WEEKDAY'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-04T18:00:00.000Z'),
  end_time: new Date('2021-01-04T20:59:00.000Z'),
  rateType: 'WEEKDAY'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-04T21:00:00.000Z'),
  end_time: new Date('2021-01-04T23:59:00.000Z'),
  rateType: 'WEEKDAY'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-09T00:00:00.000Z'),
  end_time: new Date('2021-01-09T08:59:00.000Z'),
  rateType: 'WEEKEND'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-09T09:00:00.000Z'),
  end_time: new Date('2021-01-09T17:59:00.000Z'),
  rateType: 'WEEKEND'
},
{
  price: 1.5,
  parking_garage_id: garageId,
  start_time: new Date('2021-01-09T18:00:00.000Z')
}

```

```

    end_time: new Date('2021-01-09T23:59:00.000Z'),
    rateType: 'WEEKEND'
},
{
    price: 3.5,
    parking_garage_id: garageId,
    start_time: new Date('2021-12-24T00:00:00.000Z'),
    end_time: new Date('2021-12-25T23:59:00.000Z'),
    rateType: 'HOLIDAY'
},
{
    price: 40,
    parking_garage_id: garageId,
    start_time: new Date('2021-01-04T00:00:00.000Z'),
    end_time: new Date('2021-01-04T23:59:00.000Z'),
    rateType: 'DAYRATE'
},
{
    price: 400,
    parking_garage_id: garageId,
    start_time: new Date('2021-01-04T00:00:00.000Z'),
    end_time: new Date('2021-01-04T23:59:00.000Z'),
    rateType: 'MONTHRATE'
}
];

```

8.7.3 Edit Garages

The edit view is split vertically into two sections. On the left is the garage meta information like name and address, and on the right is the information for the levels.

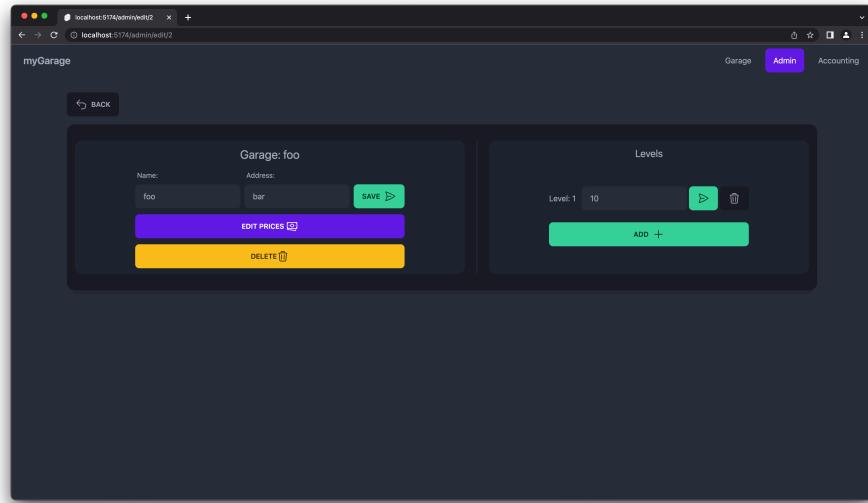


Figure 14: Edit garage

In the level section left, it's possible to add or remove parking spaces or entire levels. by pressing the `edit prices` button, it's possible to edit the prices for a garage in the given predefined timeslots.

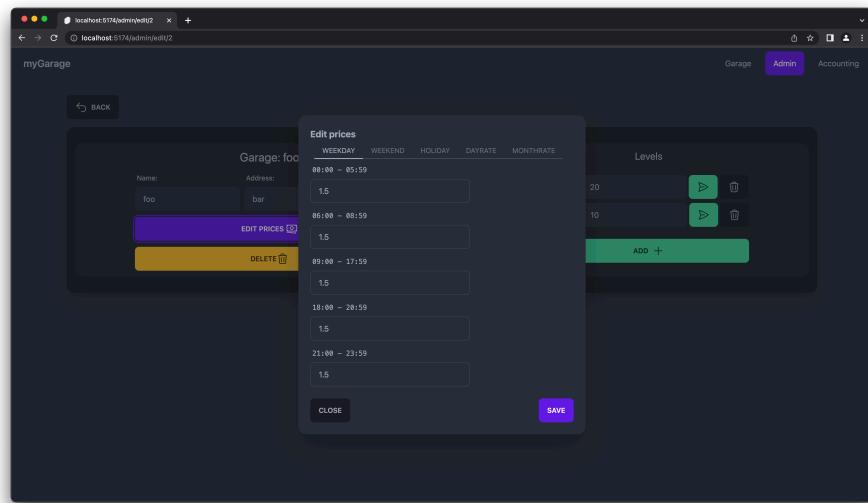


Figure 15: Edit prices

8.7.3.1 Implementation Update garage

```
updateGarage: async ({ request }) => {
  const data = await request.formData();
  const id = data.get('id');
  const name = data.get('name');
  const address = data.get('address');

  if (!id) return fail(422, { error: 'Missing id' });

  const idNumber = Number(id);

  try {
    await prisma.parkingGarage.update({
      where: {
        id: idNumber
      },
      data: {
        name: name?.toString(),
        address: address?.toString()
      }
    });
  } catch (error) {
    return fail(422, { error: 'Garage does not exist' });
  }
  return {
    status: 200
  };
},
```

This function checks if the given form data is complete, and updates the entry with the new information.

Delete Garage

```
deleteGarage: async ({ request }) => {
  const data = await request.formData();
  const id = data.get('id');

  if (!id) return fail(422, { error: 'Missing id' });

  const idNumber = Number(id);

  try {
    await prisma.parkingGarage.delete({
      where: {
        id: idNumber
      }
    });
  } catch (error) {
    return fail(422, { error: 'Garage does not exist' });
  }
  throw redirect(303, '/admin');
}
```

This function checks if the given form data is complete, and deletes the entry from the database.

Delete level

```
deleteLevel: async ({ request }) => {
  const data = await request.formData();
  const id = data.get('id');

  if (!id) return fail(422, { error: 'Missing id' });

  const idNumber = Number(id);

  try {
    await prisma.level.delete({
      where: {
        id: idNumber
      }
    });
  } catch (error) {
    return fail(422, { error: 'Level does not exist' });
  }
  return {
    status: 200
  };
},
```

This function checks if the given form data is complete, and deletes the entry from the database.

Add level

```
addLevel: async ({ request }) => {
  const data = await request.formData();
  const parkingGarageId = data.get('garageId');
  if (!parkingGarageId) {
    return fail(422, { error: 'Missing data' });
  }

  const parkingGarageIdNumber = Number(parkingGarageId);
  const highestLevel = await prisma.level.findFirst({
    where: {
      parking_garage_id: parkingGarageIdNumber
    },
    orderBy: {
      levelNumber: 'desc'
    }
  });
  try {
    await prisma.level.create({
      data: {
        levelNumber: highestLevel ? highestLevel.levelNumber + 1 : 1,
        parking_spaces: 10,
        parking_garage_id: parkingGarageIdNumber
      }
    });
    catch (error) {
      return fail(422, { error: 'Level already exists' });
    }
    return {
      status: 200
    };
  }
}
```

This function first checks if the given form data is complete, then requests the database for all levels for the given garage and orders them descending and returns the first element, this element is representative of the highest level currently in the database. Afterwards a new level gets created that has the level number incremented by one and has a default number of parking spaces of 10 parking spots.

Update the number of parking spaces

```
parkingSpaces: async ({ request }) => {
  const data = await request.formData();
  const id = data.get('id');
  const parkingSpaces = data.get('parking_spaces');

  if (!id || !parkingSpaces) return fail(422, { error: 'Missing data' });

  try {
    await prisma.level.update({
      where: {
        id: Number(id)
      },
      data: {
        parking_spaces: Number(parkingSpaces)
      }
    });
  } catch (error) {
    return fail(422, { error: 'Level does not exist' });
  }
  return {
    status: 200
  };
}
```

This function checks if the form data is complete, then updates the number of parking spaces for the given level.

Update parking rates

```
updatePrices: async ({ request }) => {
  const data = await request.formData();
  const garageId = data.get('garage');

  if (!garageId) return fail(422, { error: 'Missing data' });

  const rates = await prisma.parkingRate.findMany({
    where: {
      parking_garage_id: Number(garageId)
    }
  });

  for (const rate of rates) {
    const price = data.get(rate.id.toString());
    if (price) {
      await prisma.parkingRate.update({
        where: {
```

```

        id: rate.id
    },
    data: {
        price: Number(price)
    }
);
}
return {
    status: 200
};
}

```

This function first checks if the given form data is complete, afterwards all rates for the given garage are requested from the database. Then for each rate, the corresponding value from the form data is read and updated in the database.

8.7.4 Garage details

The detailed view gives an overview of all levels of this garage. Each level is represented as a card and has some basic information displayed like the level number, the estimated revenue from the currently occupied parking spots and how many permanent tenants on this level exist.

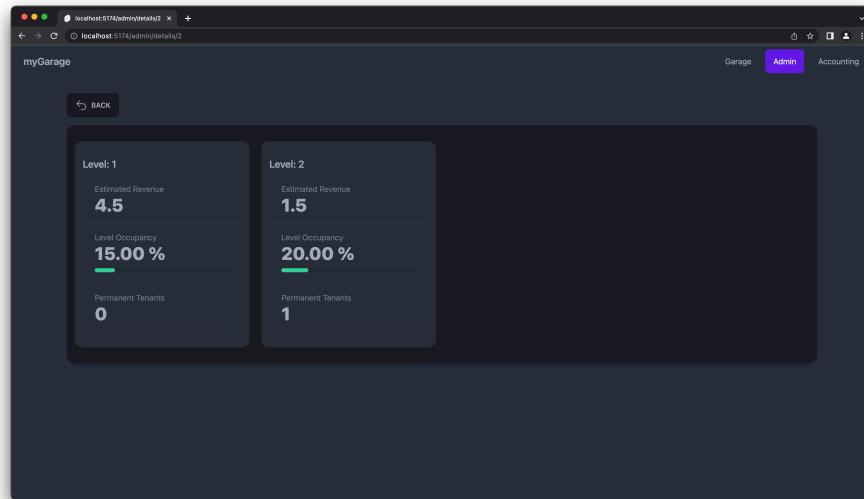


Figure 16: Garage details

A detailed view of a specific level can be shown by clicking on a level.

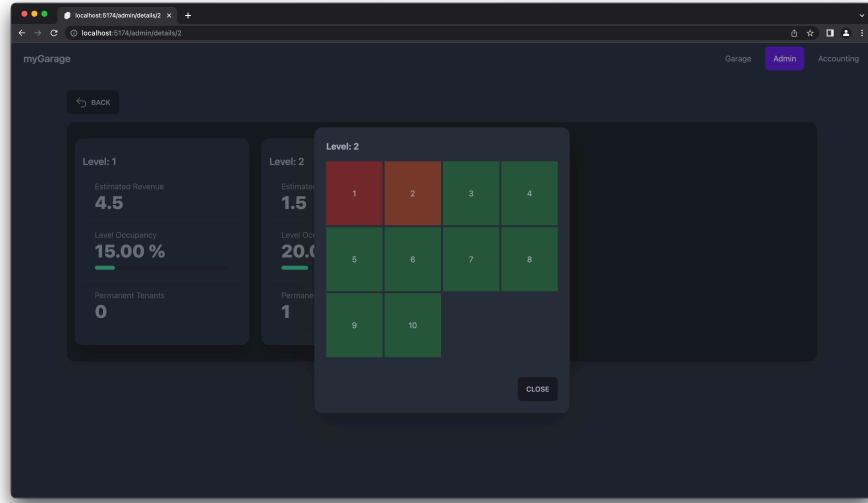


Figure 17: Level details

The detailed level view shows all the parking spaces this level has with the following color code:

- **Green:** This spot is free
- **Yellow:** This spot is occupied by a permanent tenant and will not be free
- **Red:** This spot is currently occupied by an occasional customer but will be free once they leave.

8.7.4.1 Implementation

```
import { prisma } from '$lib/server/database';
import { getAllParkingSpacesForLevel } from '$lib/util/parkingSpaceUtil';
import { getReportForLevel } from '$lib/util/reports';
import type { PageServerLoad } from './$types';

export const load = (async ({ params }) => {
  const levels = await prisma.level.findMany({
    where: {
      parking_garage_id: Number(params.slug)
    },
    orderBy: {
      levelNumber: 'asc'
    }
});
```

```

const levelInfo = await Promise.all(
  levels.map(async (level) => {
    const report = await getReportForLevel(level);
    const parkingSpaces = await getAllParkingSpacesForLevel(level);
    return {
      level: level,
      parking_spaces: parkingSpaces.map((parkingSpace) => ({
        occupied: parkingSpace.occupied,
        permanentTenant: parkingSpace.permanentTenant
      })),
      report: report
    };
  })
);

return { levelInfo };
}) satisfies PageServerLoad;

```

The detail view first loads all information needed for both views. There are no actions so there only exists a load function. The information is retrieved using the same utility method as used to create **reports**

The reactive element (clicking on a level) is made using the dispatch function

```

<script lang="ts">
  import { createEventDispatcher } from 'svelte';
  import type { LevelInfo } from '../../../../../routes/admin/details/[slug]/+page.svelte';
  import ParkingSpace from './ParkingSpace.svelte';

  export let level: LevelInfo | undefined = undefined;
  let inner: HTMLElement | undefined = undefined;
  const dispatch = createEventDispatcher();

  function close(e: KeyboardEvent | MouseEvent) {
    if (e.target === inner) return;
    dispatch('close');
  }
</script>

<div class="modal" class:modal-open={level}
  on:click={(e) => close(e)} on:keydown={(e) => close(e)}>
  <div class="modal-box" bind:this={inner}>
    {#if level}
      <h3 class="text-lg font-bold">Level: {level.level.levelNumber}</h3>
      <div class="grid grid-cols-4 gap-1 py-4">
        {#each level.parking_spaces as parking_space, i}
          <ParkingSpace

```

```

        occupied={parking_space.occupied}
        permanant={parking_space.permanentTenant}
        label={i + 1}
    />
{/each}
</div>
<div class="modal-action">
    <button class="btn" on:click={() => dispatch('close')}>Close</button>
</div>
{/if}
</div>
</div>

```

8.8 Parking rates

In the requirements from ParkingTown AG, it's specified that the prototype should support changes in the parking rates. It's not specified how these rates should be implemented. To keep the prototype simple the rates have the following time slots that can not be modified. What can be modified is the Price per hour.

A default value is implemented following the tables listed here:

Weekdays

Table 6: Parking rates weekdays

time	Price per hour
00:00-05:59	CHF 3.00 / hr
06:00-08:59	CHF 3.40 / hr
09:00-17:59	CHF 4.20 / hr
18:00-20:59	CHF 3.40 / hr
21:00-23:59	CHF 3.00 / hr

Weekend

Table 7: Parking rates weekends

time	Price per hour
00:00-08:59	CHF 3.00 / hr
09:00-17:59	CHF 3.80 / hr
18:00-23:59	CHF 3.00 / hr

Additionally to these timeslots, there is a daily rate = 40 and a monthly rate = 400

8.8.1 Implementation

The price is calculated using a utility method called `calculatePrice`. The logic that handles the finding of the correct time slot is written without using any library. Modern JavaScript time libraries like Luxon ([moment, 2023](#)) could simplify the logic immensely.

```
export async function calculatePrice(parkingTicket: ParkingTicket) {...}
```

The function first checks if the `parkingTicket` is for a permanent tenant or an occasional user.

```
if (parkingTicket.customer_id !== null) {
  const customer = await prisma.customer.findFirst({
    where: {
      id: parkingTicket.customer_id
    }
  });

  if (!customer) {
    throw new Error('No customer found for this parking ticket');
  }

  if (customer.is_long_term_customer) {
    return 0;
  }
}
```

If the user is a permanent tenant, no cost for their visit needs to be added, therefore the function returns early.

If the customer is an occasional user a few constants get defined.

```
const rates = await prisma.parkingRate.findMany({
  where: {
    parking_garage_id: parkingTicket.parking_garage_id
  }
});

let isHoliday: boolean | null = null;
const timeNow = new Date();

let timeDifferenceInHours = Math.ceil(
  (Date.now() - parkingTicket.entry_date.getTime()) / 1000 / 60 / 60
);

const days = Math.floor(timeDifferenceInHours / 24);

if (days > 0) {
```

```

        timeDifferenceInHours = timeDifferenceInHours - days * 24;
    }

    if (!rates) {
        throw new Error('No rates found for this parking garage');
    }

    const dayRate = rates.find((rate) => rate.rateType === RateType.DAYRATE);

    if (!dayRate) {
        throw new Error('No day rate found for this parking garage');
    }

```

This code first gets all the rates for this parking garage. Then a new Date object gets created that represents the current day. Afterward, the difference between the entry date and the current date is calculated and rounded up in hours. The next step is to get the daily rate for this parking garage, check if it exists and if not throw an error.

```

const matchingRatesHourSlots = rates.filter((element) => {
    const startTime = element.start_time.getHours();
    let endTime = element.end_time.getHours();

    if (endTime === 0) {
        endTime = 24;
    }

    return startTime <= timeNow.getHours() && endTime >= timeNow.getHours();
});

```

Then on the retrieved rates, a filter gets applied that filters the array and returns a new array with the elements that fulfill the condition. The filter first gets the start time of the rate and the rate slot and converts the date object to a number corresponding to the hours. The same is done for the end time, with the special case that JavaScript `getHours()` returns 0 when its midnight. To handle this special case the hours get manually set to 24 to allow an easy comparison of timeslots. After the filter is run, the `matchingRatesHourSlots` contains all slots that match the given time frame but it's not yet filtered based on the `rate.type` (weekday, weekends etc)

```

//https://date.nager.at/swagger/index.html
const request = await fetch(
    'https://date.nager.at/api/v3/IsTodayPublicHoliday/CH?offset=0',
    {
        method: 'GET',
        headers: {
            accept: '*/*'
        }
    }
);

isHoliday = request.status === 200 ? true : request.status === 204 ? false : null;

```

In order to check if today is a holiday in Switzerland a free API is used. This could be hard-coded but is tedious and would need to be changed every year. The API returns 200 if today is a holiday and 204 if it's not.

```
if (isHoliday) {
  const rate = matchingRatesHourSlots.filter((rate) => rate.rateType === 'HOLIDAY');
  return timeDifferenceInHours * rate[0].price + days * dayRate.price;
}

//weekdays
if (timeNow.getDay() < 6 && timeNow.getDay() > 0) {
  const rate = matchingRatesHourSlots.filter((rate) => rate.rateType === 'WEEKDAY');
  return timeDifferenceInHours * rate[0].price + days * dayRate.price;
}
//weekends
if (timeNow.getDay() == 6 || timeNow.getDay() == 0) {
  const rate = matchingRatesHourSlots.filter((rate) => rate.rateType === 'WEEKEND');
  return timeDifferenceInHours * rate[0].price + days * dayRate.price;
}
```

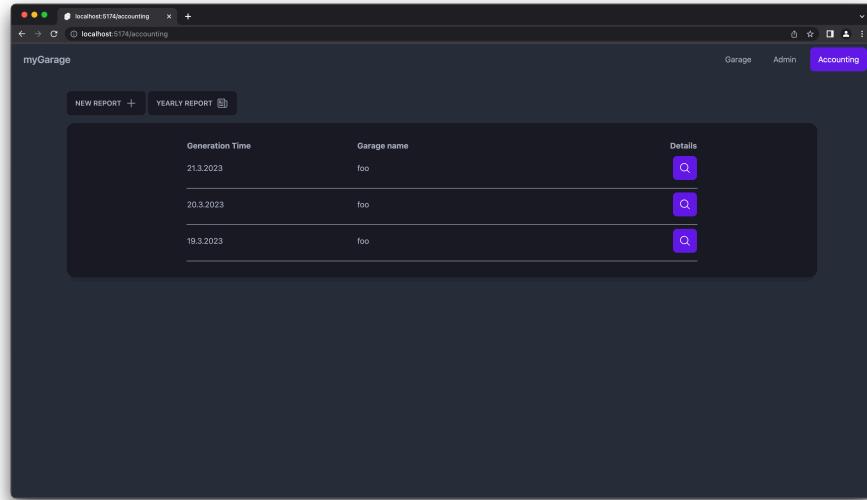
If it's not a holiday, `getDay()` returns the current weekday as a (0-6) based on that the `matchingRatesHourSlots` array gets filtered a second time to only return the matching time and type.

```
timeDifferenceInHours * rate[0].price + days * dayRate.price;
```

The calculation multiplies the time difference with the hourly rate and adds the daily rate for each day spent extra.

8.9 Accounting

Accounting is responsible for the generation of reports. There are two different kinds of reports. The first one is a report that allows searching in a custom period. The other report is a yearly report, that shows the revenue listed for each month.



A screenshot of a web application window titled "localhost:8174/accounting". The window has a dark theme with purple accents. At the top right, there are tabs for "Garage", "Admin", and "Accounting", with "Accounting" being the active tab. Below the tabs, there are two buttons: "NEW REPORT" with a plus sign and "YEARLY REPORT" with a document icon. The main content area displays a table with three columns: "Generation Time", "Garage name", and "Details". The table contains three rows of data:

Generation Time	Garage name	Details
21.3.2023	foo	[Search icon]
20.3.2023	foo	[Search icon]
19.3.2023	foo	[Search icon]

Figure 18: Accounting

When generating a report, a new form needs to be filled out. The form allows defining what type of customer the search should be for (permanent tenants or occasional customers) and which garage.

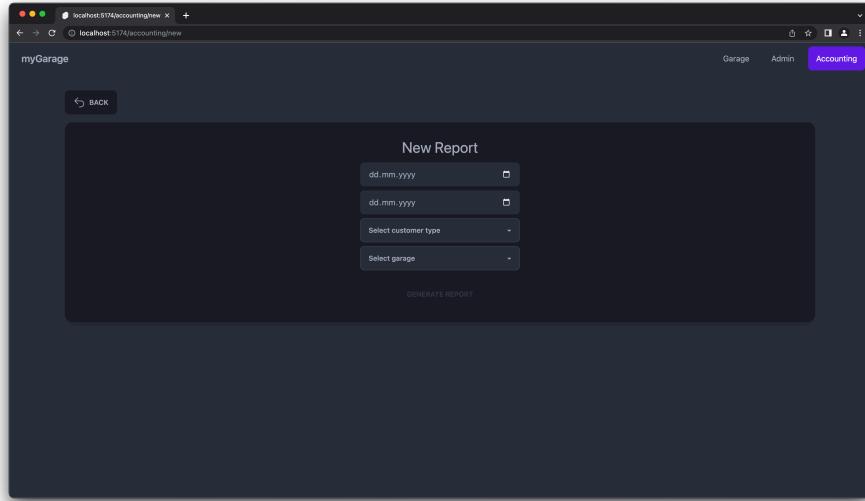


Figure 19: Generate report

The generated report shows the report id, generation date, how many parking tickets got issued during the period and the revenue from the parking tickets. The report takes currently parked occasional users into the calculation and calculates an estimated revenue.

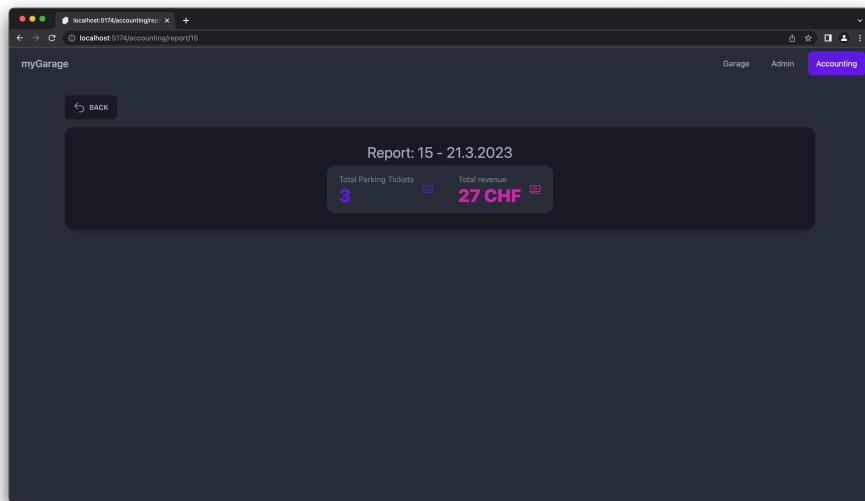


Figure 20: Report

The yearly report shows first an overview of all garages for the current year.

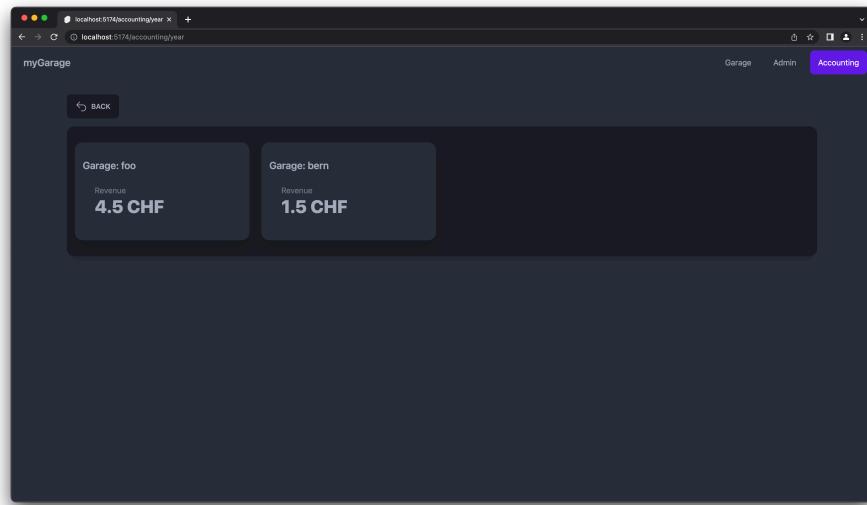


Figure 21: Yearly report

When clicking on a garage, a detailed view listing the months and their revenue shows

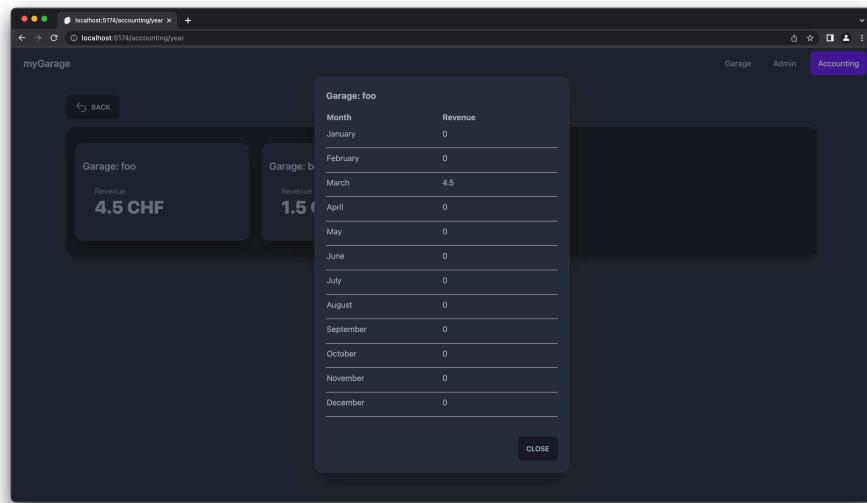


Figure 22: Detailed overview of the yearly report

8.9.1 Implementation

For the views, HTML from the level implementation was reused.

8.9.1.1 New Report

```
export const actions = {
  generateReport: async ({ request }) => {
    const data = await request.formData();
    const id = data.get('id');
    const from = data.get('from');
    const to = data.get('to');
    const isPermanentTenant = data.get('customerType');

    if (!id || !from || !to) {
      throw error(500, 'Missing parameters');
    }

    const startTime = new Date(from.toString());
    const endTime = new Date(to.toString());

    const garage = await prisma.parkingGarage.findUnique({
      where: {
        id: Number(id)
      }
    });

    if (!garage) {
      throw error(500, 'Missing garage');
    }
    let combinedRevenue = 0;
    let numberoftickets = 0;

    if (isPermanentTenant === 'true') {
      const info = await reportPermanentTenants(garage, startTime, endTime);
      combinedRevenue = info.revenue;
      numberoftickets = info.numberoftickets;
    } else {
      const info = await infoFromPeriod(garage, startTime, endTime);
      const estimatedRevenue = await getReportForGarage(garage.id);
      combinedRevenue = info.revenue + estimatedRevenue.estimatedRevenue;
      numberoftickets = info.numberoftickets;
    }

    const report = await prisma.accountingReport.create({
      data: {
```

```

    parking_garage_id: garage.id,
    generationTime: new Date(),
    searchFrom: startTime,
    searchTo: endTime,
    price: combinedRevenue,
    numberOfTickets: numberOfTickets
}
});

throw redirect(303, `/accounting/report/${report.id}`);
}
;

```

When creating a new report, first the sent form gets validated and checked if all required input data is sent. Afterward, the start and end times from the period get created using the built-in Date implementation from JavaScript. Then the garage, the report should be created for, gets read from the database. Then it either generates a report for permanent tenants or occasional users. The reason why the creation of the two reports differs is that for the occasional customers, it adds up all ticket prices and then gets the currently parked customers and calculates their fees up to the time of creation. Permanent users, on the other hand, don't get charged for leaving the parking garage and therefore have no price on their parking ticket.

Calculating occasional customers

```
async function infoFromPeriod(garage: ParkingGarage, startTime: Date, endTime: Date) {
  const tickets = await prisma.parkingTicket.findMany({
    where: {
      parking_garage_id: garage.id,
      exit_date: {
        gte: startTime,
        lte: endTime
      }
    },
    include: {
      customer: true
    }
  });

  if (!tickets) return { revenue: 0, numberoftickets: 0 };

  const numberoftickets = tickets.filter(
    (ticket) => ticket.customer?.is_long_term_customer !== true
  ).length;

  const revenue = tickets.reduce((acc, ticket) => acc + ticket.finalprice!, 0);

  return { revenue: revenue, numberoftickets: numberoftickets };
}
```

For occasional users, first, get all parking tickets that are in the given period, and count the length of the array filtering for only parking tickets that don't belong to a permanent tenant. Then the price of the parking tickets gets summed up, and there it doesn't matter that it's not fileted since permanent customers' exit tickets have a price of 0.

Calculating permanent customers

```
async function reportPermanentTenants
(garage: ParkingGarage, startTime: Date, endTime: Date) {
  const customers = await prisma.customer.findMany({
    where: {
      parking_garage_id: garage.id,
      is_long_term_customer: true,
      created_at: {
        lte: endTime
      },
      last_payment: {
        gte: startTime
      }
    },
    include: {
      parkingTickets: true
    }
  });

  const monthRate = await prisma.parkingRate.findFirst({
    where: {
      parking_garage_id: garage.id,
      rateType: RateType.MONTHRATE
    }
  });

  if (!monthRate) {
    throw new Error('No month rate found for this parking garage');
  }

  let revenue = 0;
  let numberOfTickets = 0;
  for (const customer of customers) {
    numberOfTickets += customer.parkingTickets.length;
    let start = DateTime.fromJSDate(customer.created_at!);
    let end = DateTime.fromJSDate(customer.last_payment!);
    if (DateTime.fromJSDate(startTime) > DateTime.fromJSDate(customer.created_at!)) {
      start = DateTime.fromJSDate(startTime);
    }
    if (DateTime.fromJSDate(endTime) < DateTime.fromJSDate(customer.last_payment!)) {
      end = DateTime.fromJSDate(endTime);
    }

    let current = start;
    while (current <= end) {
      revenue += monthRate.value * (end - current).as('months');
      current = current.add(1, 'month');
    }
  }
}
```

```

        revenue += monthRate.price!;
        current = current.plus({ months: 1 });
    }
}
return { revenue: revenue, numberOfTickets: numberOfTickets };
}

```

Calculating the revenue of permanent tenants is more complicated. Since their price is not on the parking ticket it's not as easy as summing up the price. First, all customers created before the given time period that have at least paid their last monthly fee once during the period get requested from the database. Then the monthly rate for that garage gets saved. Afterward, the function iterates over each customer. The function then iterates over the period and checks if the customer was paying each month, and only adds the monthly fee to the total revenue if the customer paid.

9 Verification and validation

ID	Trace from	Test Case	Should be	Is
V101	R101	Go to the admin page and create a new garage	The configuration of parking garages allows the number of levels and parking spaces per floor being freely definable	OK
V102	R102	Add a new floor to the garage created in V101 and add a different number of parking spaces	The number of parking spaces per level can be different and changed per level basis	OK
V103	R107	On the garaage view, slect the garage created in V101 and enter a random number as license plate, press the entrance button	The view redirected to the checkot view, a parking ticket is created	OK
V104	R108	In the Database management tool, enable the customer created in V103 to be a permanant tenant, past his id in the garage view	he view redirected to the checkot view, a parking ticket is created	OK
V105	R109 R114	checkout as the customer from V104	exit ticket gets created	OK
V106	R110 R111 R112 R113 R114	Enter as a new occational user, checkout again	parking ticket gets created	OK
V107	R115	Create a new report for both customer types in the accounting tab	Report get generated, saved in the history and cost is caluclated	OK
V108	R116 R117 R118 R119 R120 R123 R124	In the edit view of the garage created in V101, change the default rates	The rates are changed and saved	OK
V109	R121 R122	In the detail view of the garage created in V101, click on a level	A view of the levels and their occupied parking spots is visible	OK

10 Configuration manual

This section describes the configuration management process used in the project. It covers the creation of the working environment, the project structure, the tools and frameworks used, and what to consider when modifying project files.

10.1 Configuration elements

Element	Description
Technical Documentation (TD)	Project documents which are created within the framework of software development and maintenance according to the process specifications
sveltekit project code	The entire non-third-party svelte code
Docker	container runtime 20.10.23
Prisma	ORM 4.10
Railway	Hosting provider
PostgreSQL	Database engine

10.2 Project environment

The entire source code can be found on [GitHub](#)

Branch	Description
main	Main branch. Protected. A commit is not possible. Changes must be realized via pull requests
IssueId-issue-name	Development branches generated from tickets

10.3 Issues

Issues are used to track changes to the source code. An issue is created on GitHub. The issue should describe what should be changed or implemented. Issues can be used for all changes (e.g. add a feature, adjust configuration, reminder, etc.).

11 Hosting

The source code repository is publicly available therefore I want to also host the application. I already used different free hosting providers like Vercel ([Vercel, 2023](#)) or Netlify ([Netlify, 2023](#)). Since the application is using a database it's sadly not possible to use Vercel's free plan since that only allows edge deployment.

I already used Railway ([Railway, 2023](#)) for a personal project once and it was pretty simple to host an application for free with decent uptime.

11.1 SvelteKit on Railway

The following configuration needs to be changed to run SvelteKit on Railway

`package.json`

1. add the following line to scripts

```
"start": "node build",
```

2. add node adapter

```
npm i -D @sveltejs/adapter-node
```

3. change the adapter in `svelte.config.js`

```
import adapter from '@sveltejs/adapter-node';
```

4. change vite config to resolve prisma enums

```
resolve: {  
    alias: {  
        '.prisma/client/index-browser': './node_modules/.prisma/client/index-browser.js'  
    }  
}
```

Then go to [Railway](#) select deploy from GitHub and choose the repository.

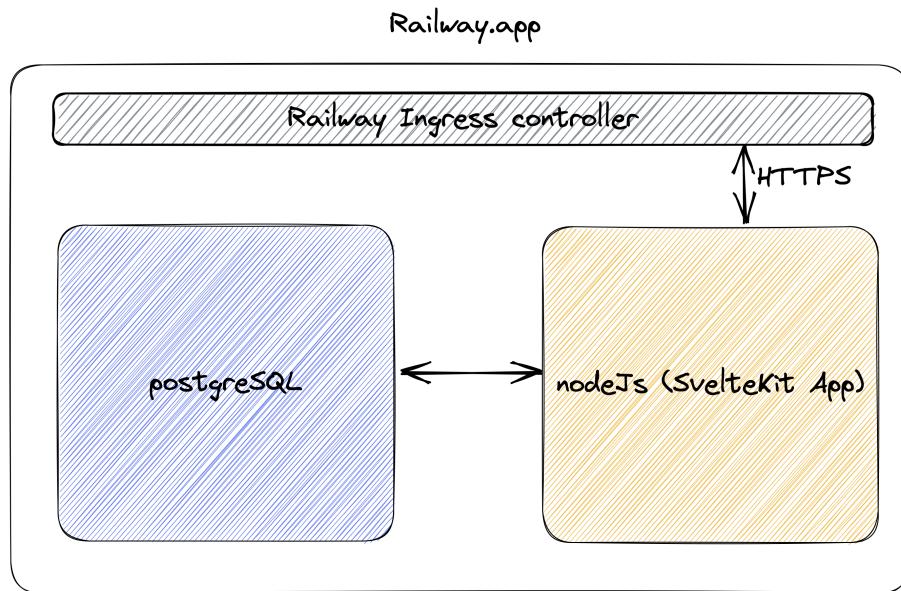


Figure 23: Railway deployment

After choosing the repository a new environment gets created, Railway tries to create and deploy the application but a database needs to be added first. From the service catalog, a PostgreSQL database can be deployed, after successfully deploying the database add the variables to the app with the new [database reference variables](#) from Railway. After this, the application will be built and deployed.

12 Conclusion

I chose to use a framework and language I'm not familiar with to learn something new. Additionally, I also wanted to test if writing large documentation and academic papers in markdown is suited or if it becomes slow and unmanageable when the documentation grows.

I learned a lot about SvelteKit and typescript and would say I'm now more comfortable with using both. I enjoyed writing the documentation in markdown and using pandoc to convert it to a pdf. I noticed that pandoc has its limitation and that if you want to highly customize the built pdf it's maybe easier to just use LaTeX from the start. I will for sure use this setup in a future project again, but will spend more time on fine-tuning the pandoc output to my needs to get a cleaned and more polished look.

13 Declaration of Independence

Work that is demonstrably taken over in full or in the essential parts unchanged or without correct reference to the source is considered prefabricated and will not be evaluated.

I confirm that I have written this thesis independently and have marked all sources used. This thesis has not been submitted to an examination board before, neither in the same nor in a similar form.

Name / First name:

Place / Date / Signature:

References

- docker, 2023a. *Docker*. [online] Available at: <<https://www.docker.com>> [Accessed 2 March 2023].
- docker, 2023b. *What are containers*. [online] Available at: <<https://www.docker.com/resources/what-container/>> [Accessed 4 March 2023].
- flashparking, 2023. *Why flashparking*. [online] Available at: <<https://www.flashparking.com/why-flashparking/>> [Accessed 2 March 2023].
- Foundation, T.L., 2023. *Open container initiative*. [online] Available at: <<https://opencontainers.org>> [Accessed 4 March 2023].
- GitHub, 2023. *Github codespaces*. [online] Available at: <<https://github.com/features/codespaces>> [Accessed 10 March 2023].
- Gitpod, 2023. *Gitpod*. [online] Available at: <<https://www.gitpod.io>> [Accessed 10 March 2023].
- Google, 2023. *Kaniko*. [online] Available at: <<https://github.com/GoogleContainerTools/kaniko>> [Accessed 4 March 2023].
- kubernetes, 2023a. *Docker runtime*. [online] Available at: <<https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/#so-why-the-confusion-and-what-is-everyone-freaking-out-about>> [Accessed 4 March 2023].
- kubernetes, 2023b. *Minikube*. [online] Available at: <<https://minikube.sigs.k8s.io/docs/>> [Accessed 4 March 2023].
- LaTeX, 2023. *Latex*. [online] Available at: <<https://www.latex-project.org>> [Accessed 4 March 2023].
- Microsoft, 2023. *Vscode*. [online] Available at: <<https://code.visualstudio.com>> [Accessed 4 March 2023].
- moment, 2023. *Luxon*. [online] Available at: <<https://moment.github.io/luxon/#/>> [Accessed 10 March 2023].
- Netlify, 2023. *Netlify*. [online] Available at: <<https://www.netlify.com>> [Accessed 10 March 2023].
- nvm, 2023. *Nvm*. [online] Available at: <<https://github.com/nvm-sh/nvm>> [Accessed 5 March 2023].
- OCI, 2023. *Open container initiative*. [online] Available at: <<https://github.com/opencontainers/image-spec>> [Accessed 2 March 2023].
- Pandoc, 2023. *Pandoc*. [online] Available at: <<https://pandoc.org>> [Accessed 4 March 2023].
- parkmobile, 2023. *Parkmobile parking solutions*. [online] Available at: <<https://parkmobile.io/parking-solutions/>> [Accessed 2 March 2023].
- Postgres, 2023. *Postgres*. [online] Available at: <<https://www.postgresql.org>> [Accessed 4 March 2023].
- Prisma, 2023. *Connect prisma*. [online] Available at: <<https://www.prisma.io/docs/getting-started/setup-prisma/start-from-scratch/relational-databases/connect-your-database-typescript-postgres>> [Accessed 5 March 2023].
- prisma, 2023. *Prisma*. [online] Available at: <<https://www.prisma.io>> [Accessed 2 March 2023].
- Railway, 2023. *Railway*. [online] Available at: <<https://railway.app>> [Accessed 10 March 2023].

- skidata, 2023. *Skidata parking solution*. [online] Available at: <<https://www.skidata.com/de-ch/loesungen/parken-mobilitaet/parkleitsystem-digitale-beschilderung>> [Accessed 2 March 2023].
- StateOfJS, 2023. *State of js*. [online] Available at: <https://2022.stateofjs.com/en-US/libraries/#tier_list> [Accessed 4 March 2023].
- SUSE, 2023. *Rancher desktop*. [online] Available at: <<https://rancherdesktop.io/>> [Accessed 4 March 2023].
- svelte, 2023a. *Svelte*. [online] Available at: <<https://svelte.dev/>> [Accessed 2 March 2023].
- svelte, 2023b. *Svelte kit*. [online] Available at: <<https://kit.svelte.dev/docs/introduction>> [Accessed 4 March 2023].
- tibaparking, 2023. *About tibaparking*. [online] Available at: <<https://tibaparking.com>> [Accessed 2 March 2023].
- Vercel, 2023. *Vercel*. [online] Available at: <<https://vercel.com/>> [Accessed 10 March 2023].
- wikipedia, 2023a. *Object relational mapping*. [online] Available at: <https://en.wikipedia.org/wiki/Object-relational_mapping> [Accessed 2 March 2023].
- wikipedia, 2023b. *Operating system level virtualization*. [online] Available at: <https://en.wikipedia.org/wiki/OS-level_virtualization> [Accessed 4 March 2023].
- wikipedia, 2023c. *Svelte*. [online] Available at: <<https://en.wikipedia.org/wiki/Svelte>> [Accessed 4 March 2023].
- wikipedia, 2023d. *Uuid*. [online] Available at: <https://en.wikipedia.org/wiki/Universally_unique_identifier> [Accessed 10 March 2023].