

# Parking Garage

David Albrecht, TEK0 Bern

May 2023

## Contents

<b>1</b>	<b>Project summery</b>	<b>3</b>
<b>2</b>	<b>Project initialization</b>	<b>4</b>
2.1	Initial situation . . . . .	4
2.2	Situation analysis . . . . .	4
2.2.1	Product situation . . . . .	4
2.2.2	Competitive situation . . . . .	5
2.2.3	Sales situation . . . . .	6
2.3	General conditions . . . . .	6
2.3.1	Process-related general conditions . . . . .	6
2.3.2	Product-related general conditions . . . . .	6
2.4	Delimitations . . . . .	8
2.5	Stakeholder analysis . . . . .	9
2.5.1	Stakeholders . . . . .	9
2.6	Project planning . . . . .	9
2.6.1	Project Initiation . . . . .	10
2.6.2	Requirement Engineering . . . . .	10
2.6.3	Implementation . . . . .	10
2.6.4	Verification . . . . .	11
2.6.5	Development Plan . . . . .	11
2.6.6	Documentation . . . . .	11
2.6.7	Conclusion . . . . .	11
<b>3</b>	<b>Risk analysis</b>	<b>12</b>
<b>4</b>	<b>Development plan</b>	<b>12</b>
<b>5</b>	<b>Stakeholder requirements</b>	<b>12</b>
<b>6</b>	<b>System requirements</b>	<b>14</b>
6.1	Functional requirements . . . . .	14
6.2	Non-Functional requirements . . . . .	15

<b>7</b>	<b>System architecture and design</b>	<b>16</b>
7.1	Svelte / Sveltekit . . . . .	16
7.1.1	What is Svelte . . . . .	16
7.1.2	What is SvelteKit . . . . .	17
7.1.3	Why Sveltekit instead of other fullstack javascript frameworks . . . . .	17
7.2	Containers . . . . .	18
7.2.1	What are Containers . . . . .	18
7.2.2	OCI Containers . . . . .	20
7.2.3	Docker . . . . .	20
7.3	Prisma . . . . .	21
7.3.1	ORM . . . . .	22
7.3.2	PostgreSQL . . . . .	22
7.4	Context diagram . . . . .	23
7.5	Sequence diagram . . . . .	23
7.6	Class diagram . . . . .	23
7.7	ERD . . . . .	23
<b>8</b>	<b>Implementation</b>	<b>25</b>
8.1	Documentation workflow . . . . .	25
8.1.1	Folder Structure . . . . .	26
8.1.2	Buildscript . . . . .	26
8.1.3	Index.md . . . . .	27
8.2	Development environment . . . . .	28
8.2.1	Setup Development environment . . . . .	28
8.2.2	Cloud development environments (CDEs) . . . . .	29
8.3	Project folder structure . . . . .	29
8.4	Prisma Schema . . . . .	30
8.4.1	Models . . . . .	30
8.5	Algorithm . . . . .	34
8.5.1	Imports . . . . .	34
8.5.2	Implementation . . . . .	35
8.6	Customer UI . . . . .	37
8.6.1	Entry barrier . . . . .	37
8.7	Checkout . . . . .	38
8.8	Admin UI . . . . .	39
8.8.1	Garage overview . . . . .	40
8.8.2	Add garages . . . . .	41
8.9	Edit Garages . . . . .	41
8.10	Parking rates . . . . .	44
<b>9</b>	<b>Verification and validation</b>	<b>45</b>
<b>10</b>	<b>Hosting</b>	<b>45</b>
<b>11</b>	<b>Conclusion</b>	<b>45</b>

<b>12 Glossary</b>	<b>45</b>
<b>13 References</b>	<b>45</b>
13.1 notes . . . . .	45

## List of Figures

1	Gantt . . . . .	9
2	risk analysis . . . . .	12
3	Comparing Docker to VM's . . . . .	19
4	ERD . . . . .	24
5	Entry barrier . . . . .	37
6	exit barrier permanent tenants . . . . .	38
7	exit barrier . . . . .	39
8	garage overview . . . . .	40
9	overview with sample data . . . . .	40
10	new garage form . . . . .	41
11	edit garage . . . . .	42
12	admin landing page . . . . .	43

## List of Tables

1	Product-related general conditions . . . . .	7
2	List of Stakeholders and their abbreviation . . . . .	9
3	List of Stakeholders and their abbreviation . . . . .	13
4	Functional requirements . . . . .	14
5	Non-Functional requirements . . . . .	15
6	Parking rates weekdays . . . . .	45
7	Parking rates weekends . . . . .	45

## 1 Project summery

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 lots 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 in view of 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 placing the order for 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 supplemented if necessary. Then a prototype of the IT system must be built on the basis of the specifications.

### **2.2 Situation analysis**

The current software is already old and becoming harder to maintain. ParkingTown does not have the necessary development capabilities and is therefore searching for an external company that handles the development, maintenance and hosting of the new Software.

#### **2.2.1 Product situation**

The parking garage manager from ParkingTown is a tool that supports the company's employee 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 placing an order for 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.

Overall, 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.

### **2.2.3 Sales situation**

## **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. This should include user manuals and training sessions.
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	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.
GCR-105	Requirements	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.

## 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.

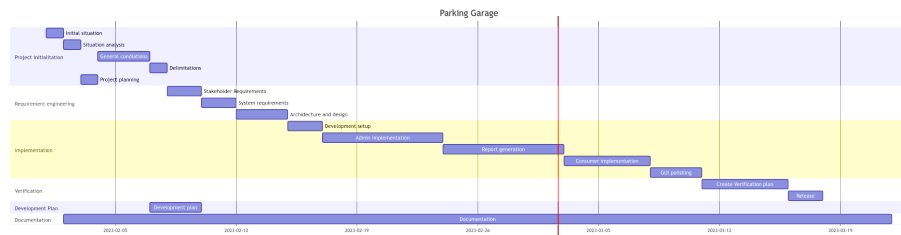


Figure 1: Gantt

### **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	ETW (1-5)	Effect (1-5)	Technical	organisation	
Customer impersonates long term customer	The impersonation is prevented	The impersonator can successfully open the barrier	1	4	x	x	
Customer presses checkout button	Exit barrier opens	The exit barrier is not opening	1	5	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 breakglass account that allows opening the barrier
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 = hardly noticeable 2 = noticeable 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 Development plan

### 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 quickly. 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 that make it a compelling choice for web application development:

- **Server-side rendering:** SvelteKit allows developers 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 fullstack javascript frameworks

One of the main reasons sveltekit was chosen for this project is, that I wanted to learn this technology 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.

In conclusion, SvelteKit is an excellent choice for web development projects, providing high performance, ease of use, built-in features and integrations, flexibility, and great community support.

## 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.

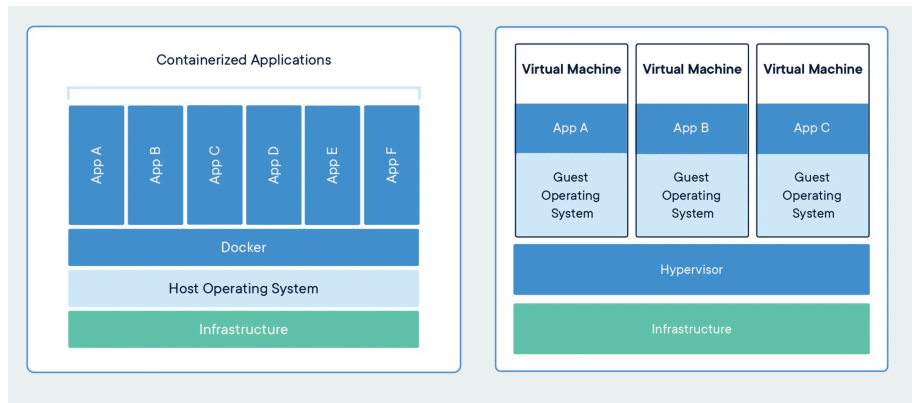


Figure 3: Comparing Docker to VM's

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.

In addition to its core features, Prisma also has a growing ecosystem of plugins and integrations that make it even more powerful. For example, Prisma Migrate enables developers to manage database schema changes in a version-controlled manner, while Prisma Client allows for real-time data synchronization between the client and server.

(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.

These are some of the reasons why PostgreSQL is used as a database, running inside a container.

(Postgres, 2023)

#### **7.4 Context diagram**

#### **7.5 Sequence diagram**

#### **7.6 Class diagram**

#### **7.7 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.

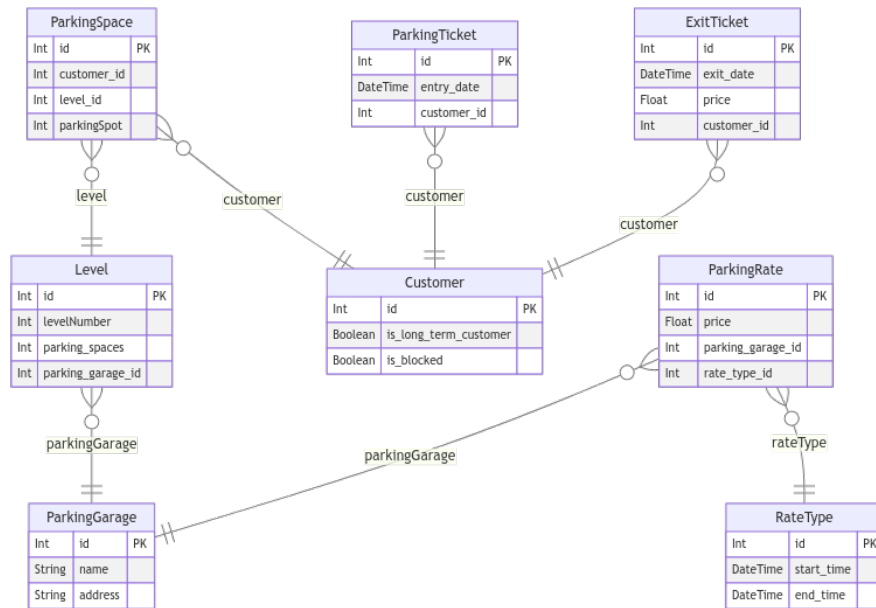


Figure 4: ERD

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 customer 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, 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 ExitTicket entity represents a ticket that a customer receives upon exiting the parking garage. Each ticket has a unique identifier (`id`), an exit date and time, a price for the parking duration, 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](#)

### 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
| -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
- **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}/${FILENAME}.md" \
        --resource-path="${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, TEK0 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

### 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

tree view generated with the command:

```
tree | pbcopy
+-- README.md
+-- docker-compose.yaml
+-- package-lock.json
+-- package.json
+-- postcss.config.cjs
+-- prisma
|   \-- schema.prisma
+-- src
|   +-- lib
|   |   +-- server
|   |   \-- util
|   \-- routes
+-- static
|   \-- favicon.png
+-- svelte.config.js
+-- tailwind.config.cjs
+-- tsconfig.json
\-- vite.config.ts
```

TODO: describe structure

## 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 parameter

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[]
  parkingTickets ParkingTicket[]
  exitTickets ExitTicket[]
  customers   Customer[]
}

```

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. **parkingTickets:** One-to-many (1-n) relation with the record “ParkingTicket”
6. **exitTickets:** One-to-many (1-n) relation with the record “ExitTicket”
7. **customers:** One-to-many (1-n) relation with the record “Customer”

#### 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())
  customer    Customer    @relation(fields: [customer_id], references: [id])
  customer_id String
  level       Level       @relation(fields: [level_id], references: [id])
  level_id    Int
  parkingSpot Int

  @@unique([level_id, parkingSpot])
}

```

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

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 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
  parkingSpace ParkingSpace[]
  parkingTickets ParkingTicket[]
  exitTickets  ExitTicket[]
  parkingGarages ParkingGarage @relation(fields: [parking_garage_id],
                                          references: [id])
  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. **parkingSpace:** one-to-many (1-n) relation with the ParkingSpace record
5. **parkingTickets:** one-to-many (1-n) relation with the ParkingTicket record
6. **exitTickets:** one-to-many (1-n) relation with the ExitTicket record
7. **parkingGarages:** one-to-many (1-n) relation with the ParkingGarage record



8. **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 ParkingTicket

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

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

#### 8.4.1.6 ExitTicket

```
model ExitTicket {  
  id          Int          @id @default(autoincrement())  
  exit_date   DateTime  
  price       Float  
  customer    Customer     @relation(fields: [customer_id], references: [id])  
  customer_id String  
  parkingGarage ParkingGarage @relation(fields: [parking_garage_id], references: [id])  
  parking_garage_id Int  
}
```

1. **id**: this is the primary key and is marked to be automatically created/incremented
2. **exit\_date**: the timestamp when a user left the garage

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

#### 8.4.1.7 ParkingRate

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

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

## 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 Imports

```
import { prisma } from '$lib/server/database';
import type { ParkingGarage, ParkingSpace, Level, Customer } from '.prisma/client';
import {
  getAllParkingSpacesForLevel,
  getOccupancyForLevel,
  type LevelParkingSpace
} from './parkingSpaceUtil';
```

The is using the ES6 module syntax to import the following modules:

1. **prisma:** object from the database module inside the `$lib/server` directory. This likely refers to an instance of the Prisma Client, which is a type-safe database client for TypeScript and Node.js applications. The database module is proba-

bly a custom module that exports the Prisma object after initializing it with the necessary configuration options.

2. **ParkingGarage, ParkingSpace, Level, Customer:** type from the `.prisma/client` module. This module is generated by Prisma based on your database schema and contains TypeScript types that correspond to your database tables and columns. In other words, it provides a strongly typed interface to interact with your database using Prisma Client.
3. **getAllParkingSpacesForLevel, getOccupancyForLevel:** utility functions and types

### 8.5.2 Implementation

1. Function definition

```
export async function findEmptyParkingSpace(  
  garage: ParkingGarage  
) : Promise<LevelParkingSpace | null>
```

This defines an asynchronous function that requires a parameter of type `ParkingGarage` **ParkingGarage Prisma** and returns a Promise that is either of type `LevelParkingSpace` or `null`.

`LevelParkingSpace` is a custom interface defined in the `parkingSpaceUtil.ts` file.

1. The interface has the following properties:

```
export interface LevelParkingSpace {  
  parkingSpot: number;  
  occupied: boolean;  
  level_id: number;  
}
```

- **parkingSpot:** the number of the parking spot
- **occupied:** boolean that represents if the parking spot is free or occupied
- **level\_id:** reference on what level this parking spot is

3. Get all levels for this parking garage

```
const levels = await prisma.level.findMany({  
  where: {  
    parking_garage_id: garage.id  
  }  
});
```

gets all entries from the `level` table that correspond with the current garage

4. Check which level is the least occupied

```
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;

```

loop over all levels and use 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 with 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.

1. Get all parking spots for this level

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

The utility function `getAllParkingSpacesForLevel` returns an array of `LevelParkingSpace`

```

export async function getAllParkingSpacesForLevel(level: Level):
Promise<LevelParkingSpace[]> {
  const occupiedParkingSpaces = await getOccupiedParkingSpacesForLevel(level);
  const parkingSpaces = new Array<LevelParkingSpace>();
  for (let i = 0; i < level.parking_spaces; i++) {
    parkingSpaces.push({
      parkingSpot: i,
      occupied: occupiedParkingSpaces.includes(i),
    });
  }
}

```

```

        level_id: level.id
    });
}
return parkingSpaces;
}

```

- this function first gets all the occupied parking spots
- creates a new array of the type LevelParkingSpace gets created
- 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
- return the array

6. Find a parking space in the array that is not occupied

```

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 vies

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

### 8.6.1 Entry barrier

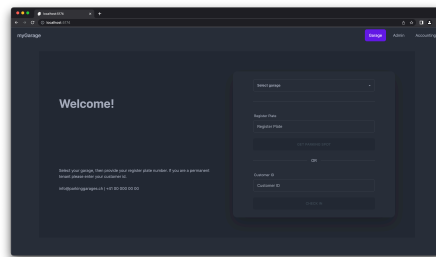


Figure 5: Entry barrier

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**

## 8.7 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

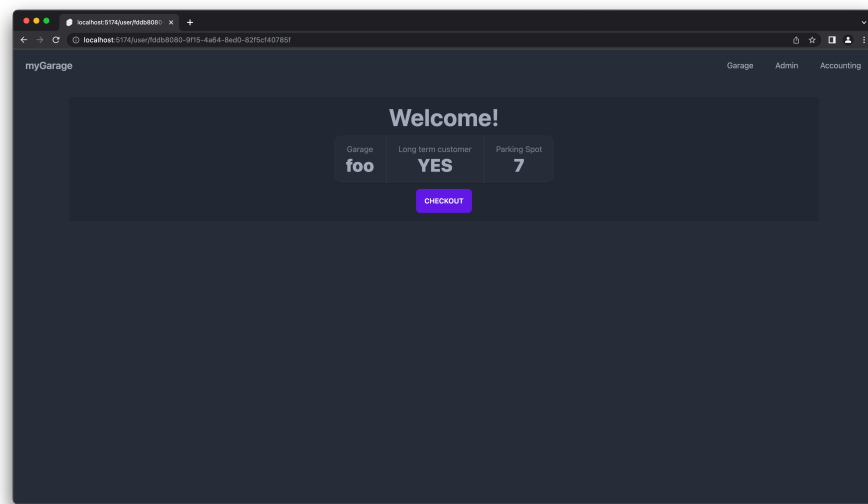


Figure 6: exit barrier permanent tenants

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

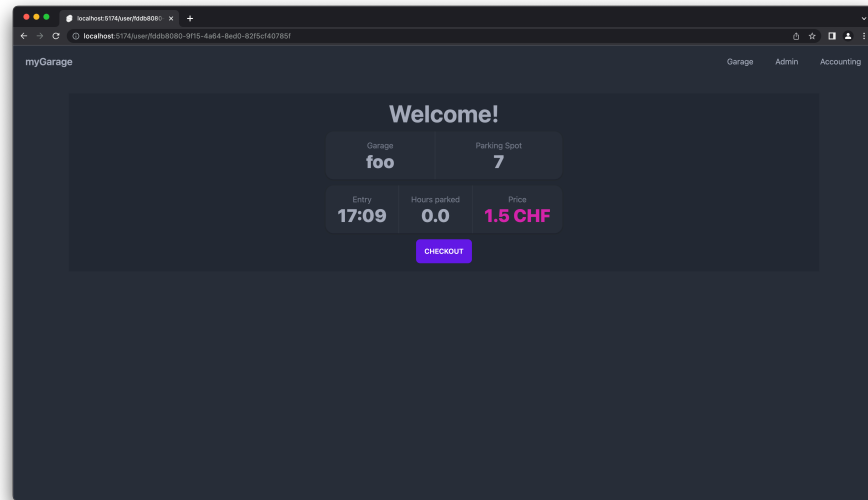


Figure 7: exit barrier

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.

## 8.8 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

### 8.8.1 Garage overview

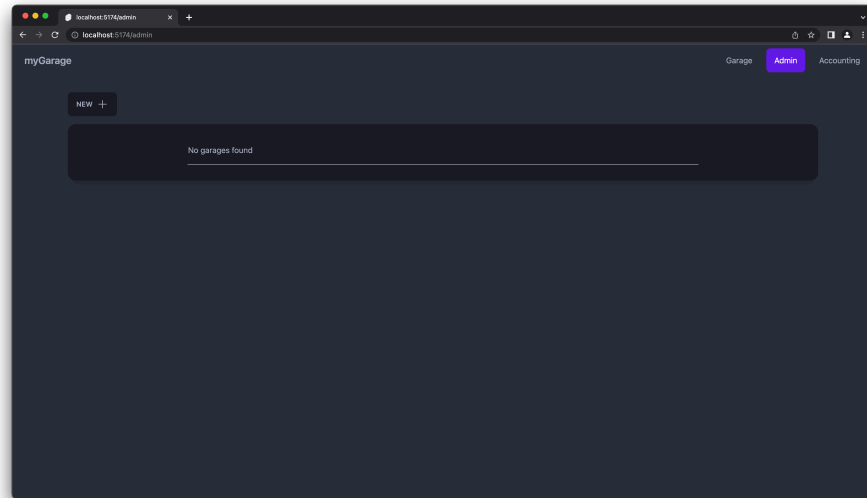


Figure 8: garage overview

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

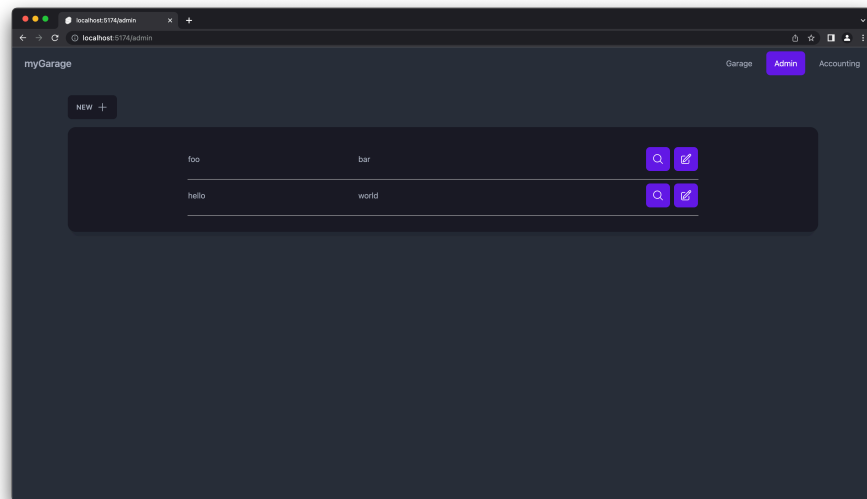


Figure 9: overview with sample data



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

### 8.8.2 Add garages

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

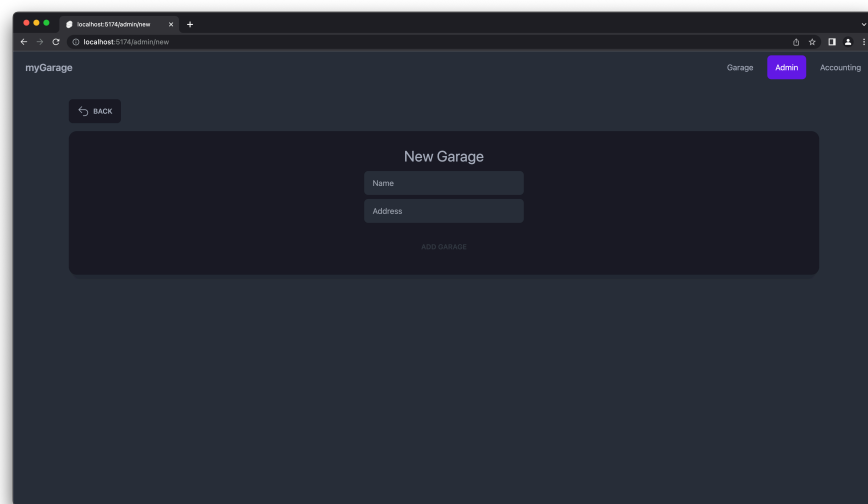
A screenshot of a web browser displaying a 'New Garage' form. The browser's address bar shows 'localhost:5174/admin/new'. The page has a dark theme. At the top left is the text 'myGarage'. At the top right are links for 'Garage', 'Admin' (highlighted in purple), and 'Accounting'. Below the links is a 'BACK' button with a left arrow. The main content area is a dark box titled 'New Garage' containing two input fields: 'Name' and 'Address'. Below these fields is a button labeled 'ADD GARAGE'.

Figure 10: new garage form

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

## 8.9 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.

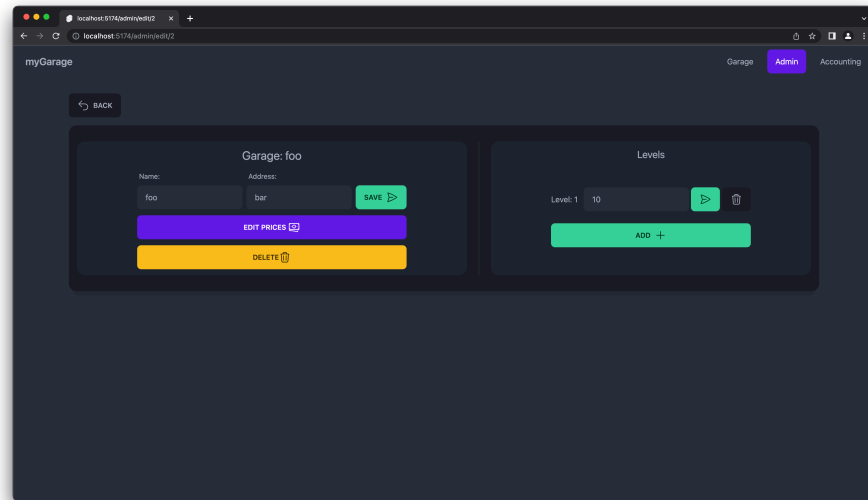


Figure 11: edit garage

In the level section left, it's possible to add or remove parking spaces or entire levels.

Svelte has default layout called `+layout.svelte`. This layout describes the default structure of all routes.

```
<script>
  import '../app.css';
  import Header from '../Header.svelte';
  //subscribe to the page store. This allows to read page data like the active rout for high
  import { page } from '$app/stores';
</script>

<div class={`flex h-full flex-col`} >
  <Header route={$page.route.id} />
  <div class="container mx-auto px-4 pt-8">
    <slot />
  </div>
</div>
```

This specifies the default view as follows:

1. Render the Header (Navbar)
2. Embed all components

Svelte does not allow the creation of named slots on the default layout. To do this it's possible to create scoped layouts for each route.

To get a seamless experience in the admin UI an additional layout file is created

AdminLayout.svelte

```
<div class="flex flex-col">
  <div class="mb-4"><slot name="navbar" /></div>
  <div
    class="rounded-box mx-2 w-72 place-items-center gap-4 bg-neutral p-4 py-8 shadow-xl xl:m
  >
    <slot />
  </div>
</div>
```

This layout can be imported into any svelte component and allows styling the components fast.

The landing page for the admin UI is /admin



Figure 12: admin landing page

The svelte component /src/routes/admin/+page.svelte implements the skeleton for this view

```
<script lang="ts">
  import type { PageData } from './$types';
  export let data: PageData;
  import AdminLayout from './AdminLayout.svelte';
</script>

<AdminLayout>
  <div slot="navbar">
    <a href="/admin/new" class="btn gap-2">
      {#each data.garages as garage}
        <tr>
          <td>{garage.name}</td>
          <td>{garage.address}</td>
        </tr>
      {/each}
    </tbody>
  </table>
  </div>
</div>
</AdminLayout>
```

```

<style>
  .garagetable td {
    /* vertical-align: top; */
    padding-bottom: 1rem;
    padding-top: 0.5rem;
  }
  .garagetable tr {
    border-bottom: 1px solid #c4c5c7;
  }
</style>

```

Trimmed down version of the component with the most important parts, for a complete overview check [GitHub admin component](#)

```

import type { PageData } from './$types';
export let data: PageData;
import AdminLayout from './AdminLayout.svelte';

```

1. **import PageData:** this allows to pass the server-side rendered data to the component
2. **let data:** creates the variable, on which the server-side rendered data will be accessible
3. **AdminLayout** imports the custom layout

```

<AdminLayout>
  <div slot="navbar">
    <a href="/admin/new" class="btn gap-2">
  </div>
  <table class="garagetable mx-auto w-full max-w-5xl table-fixed">
    <tbody>
      {#each data.garages as garage}
        <tr>
          <td>{garage.name}</td>
          <td>{garage.address}</td>
        </tr>
      {/each}
    </tbody>
  </table>
</AdminLayout>

```

Here the AdminLayout component gets filled. The navigation button to create a new garage is set in the slot called navbar. This slot comes from the AdminLayout

## 8.10 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 imple-

mented. 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` and a `monthly rate`

## 9 Verification and validation

## 10 Hosting

## 11 Conclusion

## 12 Glossary

## 13 References

### 13.1 notes

not clear

- how should report look
- customer onboarding

- customers (can you be customer of multiple garages)

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].

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].

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](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].

wikipedia, 2023a. *Object relational mapping*. [online] Available at: <[https://en.wikipedia.org/wiki/Object-relational\\_mapping](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](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](https://en.wikipedia.org/wiki/Universally_unique_identifier)> [Accessed 10 March 2023].