KU LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Problem Solving and Engineering Design, Part 3

CW1B2

Ruben Mariën (r0883561)
Robin Martens (r0885874)
Neel Van Den Brande (r0876234)
Rik Vanhees (r0885864)
Rune Verachtert (r0884615)
Tuur Vernieuwe (r0886802)

An intelligent parking garage

FINAL REPORT

Co-titular
prof. dr. ir. Bart De Decker

Coaches
Shuaibu Musa Adam
Shirin Kalantari
Hamdi Trimech

ACADEMIC YEAR 2022-2023

Declaration of originality

We hereby declare that this submitted draft is entirely our own, subject to feedback and support given us by the didactic team, and subject to lawful cooperation which was agreed with the same didactic team. Regarding this draft, we also declare that:

1. *Note has been taken of the text on academic integrity (*`https://eng.kuleuven.be/studeren/masterproef-en-papers/documenten/20161221-academischeintegriteit-okt2016.pdf`*).*
2. *No plagiarism has been committed as described on* `https://eng.kuleuven.be/studeren/masterproef-en-papers/plagiaat`*.*
3. *All experiments, tests, measurements, . . . , have been performed as described in this draft, and no data or measurement results have been manipulated.*
4. *All sources employed in this draft – including internet sources – have been correctly referenced.*

This we solemnly declare, in our own free will and on our own word of honor.

Abstract

Nowadays, time has become more of the essence, so people value efficiency on all fronts highly. This project was aimed at developing an efficient and intelligent parking garage with an automatic payment and reservation tool. Sadly, often this efficiency comes at the expense of the user's privacy. That is why this paper also looks into the question: "How can a garage be as secure as possible, without reducing its efficiency?"

A scale model of a real life parking garage was built alongside a functional application, to deliver a proof of concept in this report. Via usage of automatic number plate recognition and internet of things devices, the garage is able to detect entering and leaving cars and calculate the amount each car owner has to pay. This payment can happen automatically, but only if allowed by the user. Via a self built application the user can make reservations for a garage, see the available parking lots at any time and activate or pay the garage bills. The entire system is built with privacy and security as its primary goal.

Contents

List of Figures

List of Tables

List of Abbreviations

2FA	Two Factor Authentication
ACL	Access Control Level
ACM	Access Control Misconfiguration
ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
APM	Application Performance Monitoring
CDN	Content Delivery Network
CPU	Central Processing Unit
CSRF	Cross Site Request Forgery
CVC	Card Validation Code
DDOS	Distributed Denial of Server
DOS	Denial of Service
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transport Protocol Secure
IDOR	Indirect Object Reference
IOT	Internet of Things
IPS	Intelligent Parking System
JS	JavaScript
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MDF	Medium Density Fibreboard
MITM	Man In The Middle
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
QR	Quick Response
RAM	Random Access Memory
RCE	Remote Code Execution
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
SSRF	Server-Side Request Forgery
TLS	Transport Layer Security
TOTP	Timed One Time Password
TPA	Third-Party Authenticator
UDMS	Ultrasonic Distance Measuring Sensor
URL	Uniform Resource Locator
WiFi	Wireless Fidelity
WSGI	Web Server Gateway Interface
XSS	Cross Site Scripting

1 Introduction

People these days value efficiency in life on all fronts higher than ever. Time has become more and more valuable. Almost every family owns a car with most of them even owning two or three [STATBEL, 2022], and travelling by car has become increasingly popular. Therefore, parking efficiency is an element that affects everyone on a near daily basis. This requires engineers to invent and examine new solutions to improve this parking experience. The dissatisfaction with the classic ticket system in most parking garages, causes an evolution into more digital and automated systems. Often accomplished through licence plate recognition in combination with mobile apps and automated payment options [4411, 2022]. Despite the convenience of these systems, it poses a risk of a privacy breach [Verheyden, 2022]. Therefore, sufficient attention should be given to the security of these systems. The privacy of the user is the number one priority.

The main goal of the project is to make an automatic parking garage using Internet of Things (IoT) devices. This means that a client will be able to drive into the garage and park his/her car there for a certain duration of time. Then, drive away without having to pay with the use of a ticket. This is accomplished by cameras and Automatic Number Plate Recognition (ANPR) software, together with a mobile application.

This report describes both the mechanical and software aspects of the design. Section 2 describes the working of the entire system in an abstract way. Section 3 gives the concrete implementation of the system. The reasons why the components of the system were chosen can be found in section 4. Then, section 5 describes a sample user experience in the Intelligent Parking System (IPS). Next, section 6 gives a critical analysis of the final system. Section 7 gives a conclusion of the project and finally, Section 8 gives an overview of the course integration with the different courses in the first and second year of the Bachelor in Engineering Science.

1.1 Problem description

The official problem description is very broad: "design a fully functional intelligent parking garage" with the following requirements [Decker and Hughes, 2022]:

1. the parking garage detects the amount of available parking lots;
2. the amount of available parking lots is displayed across multiple screens;
3. drivers can reserve a parking lot;
4. the parking garage detects entering and exiting vehicles, which eliminates the need of parking tickets.

Therefore, the following infrastructure has to be designed, provided and built:

1. sensors to detect the occupancy of a parking lot;
2. a central server which stores and provides all the necessary data;
3. a frontend application which the clients can use.

The main research question this project tries to answer is "*How can we realise a safe* IoT *-infrastructure which makes parking easier, faster and foremost, safer?*" [Decker and Hughes, 2022].

2 General design description

This section describes the entire working of the system in an abstract way. The details of the concrete implementation can be found in Section 3.

The IPS consists of three main parts: the on site sensors and gateway, the frontend application and the backend.

The backend is the central entity in the system, as both the local system and the frontend application connect to it in order to query and post data. This is done via a Representational State Transfer (REST) Application Programming Interface (API), which transfers data in a JavaScript Object Notation (JSON)-format; the standard for REST APIs [Gupta, 2022]. The backend consists out of the combination of the servers which handle in the incoming requests and the database, which stores all the information about the parking garage (e.g.

occupancy of parking lots, users inside the parking garage, etc.). The backend also hosts the web variant of the frontend application and thus needs proper redirecting in order to redirect requests to the right destinations (in casu the frontend application or the database). The backend resides in a separate location (e.g. a data centre), separated from the actual garage.

The local setup requires a gateway (IoT-device) and sensors installed in the physical garage. The sensors and the gateway are preferably connected through a Local Area Network (LAN), but cable wiring is also possible, although less practical.

The gateway is a micro-controller or IoT-device which is the central processing unit for the local system. When the local system exists out of multiple IoT-devices, an extra micro-controller can serve as a gateway for all these devices. This micro-controller can post and request data from the backend and controls the interaction between the different sensors. It performs only simple logical tasks, but delegates complex calculations to the backend.

Two cameras detect entering and exiting cars, via ANPR. They are linked via the gateway with two servo motors, which control the barriers for entering and exiting the garage. Inside the garage, sensors detect if a parking lot is occupied or not. A Light Emitting Diode (LED) visually indicates to the drivers if the parking lot is occupied/reserved. Furthermore, a display at the entrance of the garage indicates the amount of free spots left.

The frontend application serves as a Graphical User Interface (GUI) for the users to interact with the backend database. The application comes in two formats: a mobile application which is installable on both iOS and Android and a web application in the form of a website. Both variants have the same functionality. Figure 1 gives a very schematic overview of the different components of the IPS.[1]

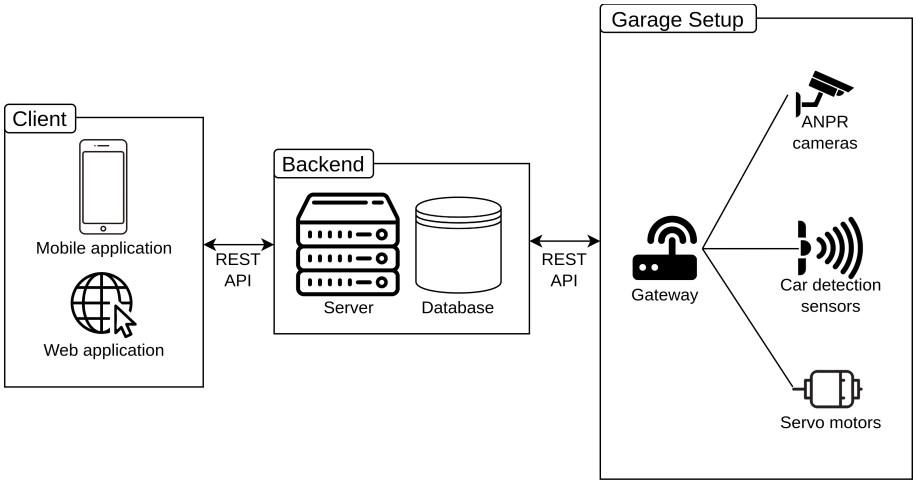

Figure 1: Abstract overview of the intelligent parking system. The backend can be deployed on one psychical machine, which can run the proxy server and the database server in parallel.

2.1 Core functionalities

The three main spearheads of the IPS described above are user privacy, security and ease of use. Table 1 gives a summary of this paragraph, listing the three core functionalities, together with the steps taken to achieve them. Ease of use is split up in three sub-objectives.

Firstly, it is not necessary for users of the parking garage to install the mobile application or even have a pre-existing account. It is possible for users to drive to the parking garage and park without any need for a prior setup, but receive an almost equal user experience.

Secondly, if the user creates an account, he/she is able to reserve parking lots in a specific garage between specific times, in order to guarantee an available parking lot for the user at his/her arrival.

Thirdly, the infrastructure supports automatic and non-automatic payments from within the frontend application, the former for users which created an account, the latter as an alternative for users without an account.

[1] All figures and tables without an explicit credit are proprietary.

Besides the ease of use for the person parking in the garage, the frontend application also supports admin features for a garage owner. A garage owner can add or delete his/her garages from the system, as well as configure their parking lots.

Maximum user privacy is achieved by, firstly, not storing any information that is not needed for the operation of the IPS and secondly, hashing all user-identifiable and/or sensitive information (e.g. passwords, licence plates, email addresses) before it is stored inside the database. This way, even if an attacker obtains access to the database, he/she will not be able to retrieve any useful information from it.

The two ways to park in the garage (i.e. with or without a pre-existing account) provide two levels of user privacy. If the user decides to park without an account, no user-identifiable information is stored in the database, except for the licence plate, which is deleted upon exit of the garage. This way, it is not possible to retrieve any personal information, nor information about the user's whereabouts from the system. From the standpoint of the system, that user ceases to exist upon exit of the garage. In the other case, licence plate information is coupled with the user's email address and a real first and last name, but history of the user's whereabouts are not stored (they are deleted upon exit of the garage).

Making an automatic payment requires payment information from the user. This information is hashed before it is stored inside the database, but the user can opt for a manual payment, in which case no payment details are stored in the database.

Furthermore, a garage owner is not able to query the users or licence plates in its garage, only the amount free parking lots left.

Security is the last core functionality. This mainly focuses on securing the backend, because it stores all important user information, but also includes securing the mobile application and the local garage setup, such that the sensors, nor the ANPR-cameras can be hijacked.

Firstly, the connections between the frontend and the backend and between the backend and the local garage gateway are encrypted. Secondly, the server only accepts traffic coming from the local garage gateway or the mobile application, which prevents Cross Site Request Forgery (CSRF)-attacks. Thirdly, users can only query information regarding their own account (i.e. their personal information and bookings), which prevents Indirect Object Reference (IDOR)-attacks and fourthly, the API can only be queried by authenticated users.

Table 1: An overview of the core functionalities of the entire system and how this is achieved on an abstract level.

Core functionality	**Achieved by**
Ease of use	• No obligatory account • Reserving parking lots • Automatic and non-automatic payments • One application for both user and garage owners
User privacy	• Least to know principle • Hashing sensitive user information
Security	• Encrypted connections • Request origin validation • Authenticated API

3 Implementation

This section gives a detailed explication of the implementation of the core functionalities as described in Section 2.1. It is divided into four parts: Section 3.1 explains the physical model of the parking garage, Section 3.2 discusses the concrete implementation of the local garage system, Section 3.3 describes the different user flows within the frontend application and lastly Section 3.4 outlines the backend system.

3.1 Parking garage

For the purpose of the demonstration, a working model of the IPS was made. Since it is not in the scope of the project to create a full scale working parking garage, a scaled down model of a parking garage was created to show the functionalities of the IPS. The scale model meets the following set of requirements: 1) it has a sufficient amount of parking spaces to show the functionalities as described in Section 2.1; 2) it has working barriers to let cars enter and exit the parking garage and 3) it has signalling LEDs to show if a parking spot is occupied/reserved or available.

3.1.1 Scale model

The model has six parking lots that each have a cutout for a Ultrasonic Distance Measuring Sensor (UDMS) which detect if a car is occupying a parking lot, as well as cutouts for the indication lights that will be green if the parking space is available and red if it is either occupied or reserved. There is a small entrance/exit road equipped with two barriers controlled by servo motors to stop cars from entering if there are no available spaces left, or to stop cars from exiting if they have not paid yet. Above the barrier is a horizontal beam that is used to mount the camera for taking images of the licence plates of the entering and exiting cars. On the entrance side of the road is a Liquid Crystal Display (LCD) display mounted that shows the number of available spaces left in the garage. Underneath the parking spaces is an empty space that can be used to run the wiring and house the Raspberry Pi devices. Table 6 in Appendix D shows an overview of all the used components in the local garage system.

The scale model was laser cut out of Medium Density Fibreboard (MDF)-plates. All parts are secured together using press fit connections and further reinforced with hot glue. Figure 2 shows this scale model of the parking garage.

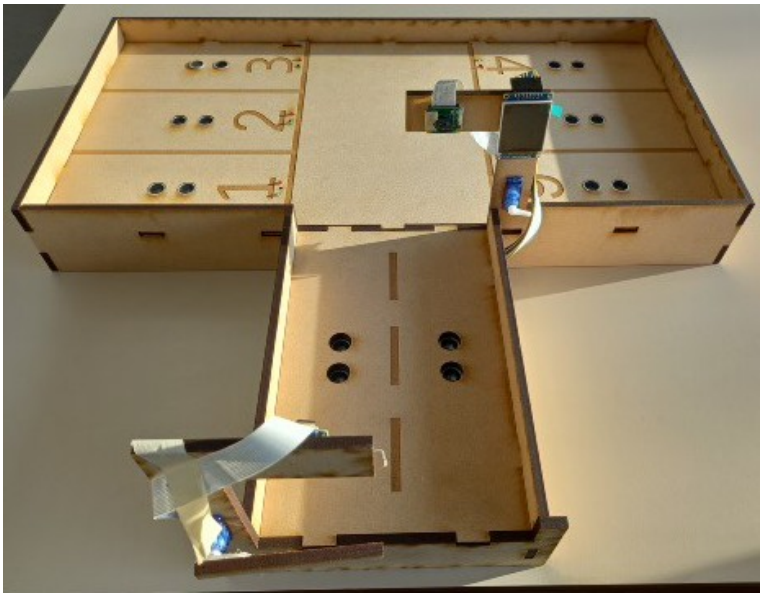

Figure 2: Scale model of parking garage.

3.2 Local garage system

There are four main functionalities that the local system fulfils: 1) identifying cars by their licence plate, 2) detecting cars on parking spots and controlling the signalling LEDs, 3) operating the servo motors of the barriers and 4) controlling the LCD-display. The backend server acts as a synchronisation for the multiple

IoT-devices in the local garage system. In the current implementation, the local garages system uses two Raspberry Pi devices. Figure 3 shows the deployment diagram of the local garage system, with the different modules which run on the Raspberry Pi devices.[2]

3.2.1 Raspberry Pi

The heart of the local system are two Raspberry Pi devices model B V1.2. They run a 64-bit Operating System (OS)[3]. This allows the usage of 64-bit Python packages, like `torch`[4]. The four main functionalities are separated into three Python packages[5] which run in parallel. This provides a segmented approach to installing all the dependencies of the different packages. Furthermore, this makes it possible for the Raspberry Pi to run the multiple services together, making the overall system faster. Figure 7 in Appendix A shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The first Raspberry Pi will also power and control the LCD-screen. Moreover that, both Raspberry Pi devices take charge of one half of the parking garage (i.e. one camera, six LEDs, four UDMSs and one servo motor). The functionalities are subdivided into two main systems (and programs on the Raspberry Pi), namely the `entrance_system`, which handles the entrance and exits of cards and the `parking_lot_system`, which handles the detection of cars. The paragraphs below explain these systems more in depth.

The Raspberry Pi communicates with the backend database with specifically designed API endpoints (see Table 9 in Appendix E for more details about the specific API endpoints) in order to update the different tables regarding the garages and parking lots. It also needs to be able to query the database, to — for example — receive information whether the user has paid or not.

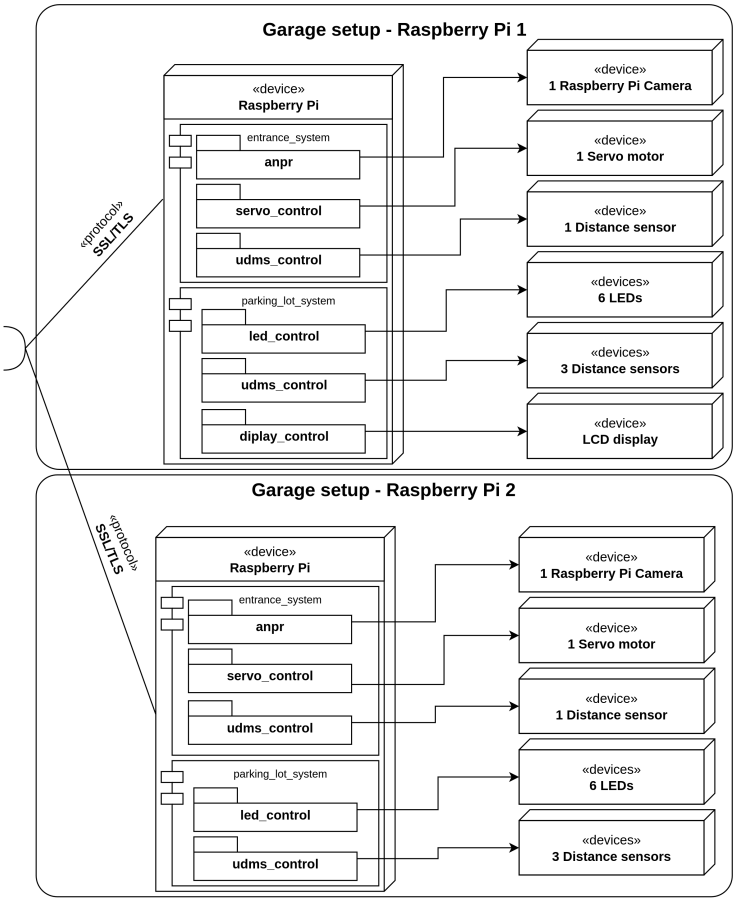

Figure 3: Deployment diagram of the garage setup.

[2]In this and the following paragraphs 'Raspberry Pi' can be replaced by a more general IoT-device. The core principles remain the same.

[3]The amount of bits of an operation system is a characteristic of the processor and determines how many memory addresses the Central Processing Unit (CPU) can access. CPUs with 32-bit can access at most 2^{32} bytes (= 4 GB) of Random Access Memory (RAM) [Computer Hope, 2020].

[4]`https://pytorch.org/`

[5]The code can be found in this GitHub repository: `https://github.com/orgs/2022P03/repositories`.

3.2.1.1 Entrance system

The local garage system contains two cameras, both powered by a Raspberry Pi. Due to performance issues, the Raspberry Pi will only take a picture of the car and send the picture to the backend, which in turn executes the ANPR (see Section 3.4.2.4 fore more details).

The taking of the image is triggered by an UDMS. The `udms_control` remembers the previous state of the UDMS. If the current state differs from the previous state, a picture is taken with a Bash script, to enhance speed. The image itself is sent to the backend. If the Raspberry Pi receive a positive status code (i.e `200 OK`), `servo_control` will open the barrier and close it after the car has passed. Automatically, the amount of free spots will drop by one in the backend due the successful posting of the licence plate image.

The same procedure applies to the exit camera, but here the backend provides an additional check to confirm that the licence plate has paid. If not, the barrier will not open and the user will be notified in the frontend application that the payment has to be fulfilled before he/she can exit the garage. Automatically, the amount of free spots will increase by one in the backend due to the successful posting of the licence plate image.

From the moment that the parking garage is completely full, either by physical cars or by reservations, the entrance system will refuse to enter new cars, except the ones with a valid reservation.

Figure 19 in Appendix F visualises the entrance procedure in detail with a sequence diagram.

3.2.1.2 Parking lot system

To know the amount of available parking places and to detect cars at the entrance or exit, the local system needs to be able to sense the presence of a car at a specific location in the garage. This can be done by several sensors. The most useful ones are UDMS or light sensors. The latter is more expensive and does not offer any extra advantages over the UDMS. For this purpose the demo garage uses the UDMS (HC-SR04).

The UDMS can measure distance by sending ultrasonic sound waves. After the pulse bounces of an object, it gets detected by the same sensor. Using the time between sending and receiving the pulse, the distance can be calculated because the speed of sound is known. To reduce the risks of false positives, the Raspberry Pi remembers the last two states of every parking lot. From the moment that it detects a car (i.e. the distance measured is smaller than 5 cm) two consecutive times, a request is sent to the backend indicating that the parking lot is occupied and the red LED is turned on. The same procedure happens when a car leaves.

One Raspberry Pi has a second component to the `parking_lot_system`, namely `display_control`, which powers and controls the LCD-display. This package will make a request to the backend every 5 seconds for the free parking lots left in the garage and display this amount (see Section 6 for a discussion about this method).

3.3 Frontend

This section describes the implementation of the core functionalities of the frontend as described in Section 2.1. The frontend application is written in Dart, with the Flutter[6] framework of Google. Flutter is used, because of its platform independence (it can run both on Android, iOS and web), and its provided type safety and null safety. Furthermore, Flutter supports a hot reload feature, which makes it easy to develop the application. The next section gives a rough idea of how the app will work when it is launched. The application is subdivided into different pages. Each page is enumerated with a combination of letters and number; a new number indicating a new flow and a new letter indicating a new subflow. An overview and a short description of all app pages can be found in Appendix B, Table 5.

3.3.1 Application design

The following paragraphs discuss the many functionalities of the frontend application. These functionalities are achieved using a multitude of screens, which are bundled into several flows in the application. Table 2 gives an overview of all the major application flows and Figure 8 in Appendix B gives a schematic overview of the relations between those screens. Of course, the routes defined in Figure 8 are the ideal routes, given that the user enters valid input, which is not always the case. The frontend application implements rigorous validation, such that all data entered will match the type and format asked by the backend. If the user enters non-valid information, he/she will not be able to continue and a dialog is displayed to the user containing the exact description of the errors. Figure 17 in Appendix B shows examples of this. Besides signalling errors, pop-ups are also frequently used to display information about the application or about the request the user has made. See Figure 18 in Appendix C for examples of this.

[6] `https://flutter.dev/`

In general, if a button in the application is pressed which loads a new screen, a request is sent to the backend to get the information about the user, the garage or the reservation required to construct the page. When information has to be changed in the database, different requests are sent to the backend to post, change or delete information in the database.

The first page of the application is determined dynamically, based on the authentication state of user. If the user is both authenticated and verified (i.e. the user has logged in *and* has provided a Two Factor Authentication (2FA)-code), the application will open the home page directly and the authentication flow will be skipped. If the user is unauthenticated, the login page is opened and the user has to follow the authentication flow. The third case exists when a user is authenticated, but not yet verified (and has 2FA enabled for his/her account). In that case, page 0c is opened for the user to submit their 2FA-code.

The following paragraphs explain the flows more in depth. All referenced figures can be found in Appendix B.

3.3.1.1 Authentication flow

Depending on if the user is logged in or not, the login page is the first page the user lands on when opening the application. Here they will have the option to login with either their existing account or to register a new one. If they choose to make a new account, then the register page pops up. On this page users have to fill in their first name, last name, email address and password (a licence plate is later added, see Section 3.3.1.6). These credentials are used to create a secure account. The user is required to confirm their password by entering the same password in another text field. This is required for lowering the chances of accidentally typing the wrong password. Once the necessary text fields are submitted, the user will be present in the database, but will not yet be active. An email is then sent to the given email address to verify that it is a real address. This email allows users to activate their account and subsequently log in.

If the account is verified, the user can login by hitting the 'Sign in'-button. When authenticated, the homepage will be loaded and the app will make a request to the backend server to load all the possible garages. Whilst the app is connecting with the backend, there will be a progress indicator on the screen and the user will still be able to access other features like the navigation bar (Pages 0, 0a, 0b and 0c; Figure 9).

3.3.1.2 Home flow

The home flow describes the options available to the user in the home screen itself. This includes opening the navigation bar, which provides the entry point for all major flows inside the application and checking the notifications (Pages 1, 1a and 1b; Figure 10).

3.3.1.3 Reservation flow

The most important flow in the application is the reservation flow. After all the garages have been loaded into the app, the user can select one of the garages to make a reservation. Next the information screen will pop up of the selected garage. On this screen the user can observe the location, the amount spots left in the garage, the price and the opening hours. If the user is satisfied with this garage, he/she can book a reservation. There will be the option to choose a licence plate linked to the account (only enabled licence plates are able to make a reservation, see Section 3.4.4.2), the time and day of the reservation and an available spot in the garage. The user can choose between selecting a spot and getting a spot assigned at random. In the former, the app requests all available parking lots from the backend, out of which the user can choose one, in the latter, the backend returns a random free parking lot. Hereafter, an overview of the reservation is shown and the user is asked to confirm (Pages 2, 2a, 2b, 2c, 2d; Figure 13).

3.3.1.4 User settings flow

The user settings flow is reached via the navigation bar in the home screen. The user can add a device in order to execute the 2FA and add credit card details for automatic payment. In addition, the 2FA and the automatic payment can be enabled or disabled as soon as a device or payment card is added. (Pages 3, 3a, 3b; Figure 12).

3.3.1.5 User reservations flow

This short flow enables users to view, alter and delete their made reservations. The current active reservation is highlighted in green; past reservations are greyed out. Note that it is possible for user with multiple validated

licence plates to have multiple reservations at the same time. Multiple reservations on the same moment with the same licence plate are not allowed. See page 4 on Figure 13 for more details.

3.3.1.6 Profile flow

The profile flow is one of the more elaborate flows. When the user selects the "Profile" tab in the navigation bar, two options are available: user information and licence plates. On the screen containing the user's info, he/she can change his/her first name, last name, email address and favourite garage name. To change their province or password another screen is shown and in order to delete your account a popup is shown. The favourite garage and the province can optionally be given by the user to enhance their experience as it alters the filtering of garage on the home screen (e.g. garages close to the user's location are displayed first) (Pages 5, 5d, 5d1, 5d2 and 5d3; Figure 14).

When the user selects the "licence plates" button instead of the "user information", there are several options. A licence plate can be added, which is shown on a popup on page 5a. If this plate already exists in the database and not under the current user's id – meaning that it is possible someone else registered this licence plate with their account –, there is an option to report this. This to make sure that no one is able to follow the whereabouts of a person based on their licence plate. Furthermore, a licence plate can be enabled, as this is required in order to make a reservation for this vehicle. In order to do this, the vehicle registration document has to be uploaded, which is implemented in the application in order to further improve the user privacy. As this not a regular request to make, page 5b1 adds an explication why it is important to validate the licence plate before it is used (Pages 5a, 5a1, 5a2, 5b, 5b1 and 5c; Figure 14).

3.3.1.7 Garage settings flow

The garage settings flow is only accessible to users that have the role `GARAGE_OWNER`. These users can see their garages in the navigation bar (Page 1a; Figure 10). They have the option to either configure their owned garages or they can add a new garage into the database. If the user presses on one of those garages, he/she will be redirected to a new page, where they can configure all the settings of the garage (Pages 6, 6a and 6b; Figure 15).

3.3.1.8 Payment flow

Payment is the last major functionality flow of the application. While actively being parked in a garage, a widget is present on the home page showing the licence plate of the vehicle parked and how long it has been parked for. When pressing pay on this widget, the user is able to see a receipt. When continuing, the user can pay for his park via Stripe on a browser tab inside of the app. Then the payment is either successful or unsuccessful. In either way the user can return to the home screen. (Pages 7, 7a, 7a1 and 7a2; Figure 16; see Section 3.4.2.5)

For users who are not familiar using the app or a mobile app in general, there would be a guide in the navigation bar. However, this was not implemented in the final design due to a lack of time. Lastly, there is also an option to sign out the user's account. A detailed schematic overview of the entire app can be found in Appendix B in Figure 8.

3.3.2 Deployment

The frontend deployment should support two use cases: users who want to download the mobile application and users who want to access the website.

Due to Flutter's nature, it can run natively on all major platforms and operating systems. To make the mobile application accessible to the general public, it should be uploaded to the Google Play Store, the Apple App Store and the Microsoft Store. For the purpose of this project, the application will be installed on the devices of the team members.

The web application should be hosted on a web server, for the users to be able to access the site. The backend already incorporates a web server, namely Nginx[7]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HyperText Markup Language (HTML) and JavaScript (JS)). Figure 4 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client connection to the backend API.

[7] `https://nginx.org/`

Table 2: Overview of the major app flows.

Flow no.	Flow name	Short description
0	Authentication flow	Flow for registering, logging in the user and sending the 2FA-code (Figure 9).
1	Home flow	Home page and banner which provides a step-stone for all major flows in the application (Figure 10).
2	Reservation flow	For making a reservation by selecting a licence plate, a spot and a date and time (Figure 11).
3	User settings flow	Flow for altering user settings like 2FA and enabling automatic payments (Figure 12).
4	User reservations flow	Screen for getting an overview of all user reservations (Figure 13).
5	Profile flow	Flow for altering user information, e.g. name or password (Figure 14).
6	Garage settings flow	Flow for garage owners which enables altering and adding of garages (Figure 15).
7	Payment flow	Flow for making payments for parking (Figure 16).

3.3.3 Backend communication

The frontend needs communication with the backend in order to show the user the right information. This is achieved by using a REST API, which exposes different Uniform Resource Locators (URLs) to the client. Tables 7, 8 and 10 in Appendix E provide a detailed overview of all the different API endpoints exposed by the backend. When the client sends a request to one of these URLs an action happens in the backend based on the request. There are four different types of requests implemented in the backend: `GET`, `POST`, `PUT` and `DELETE`. These requests respectively receive, post, change and delete information.

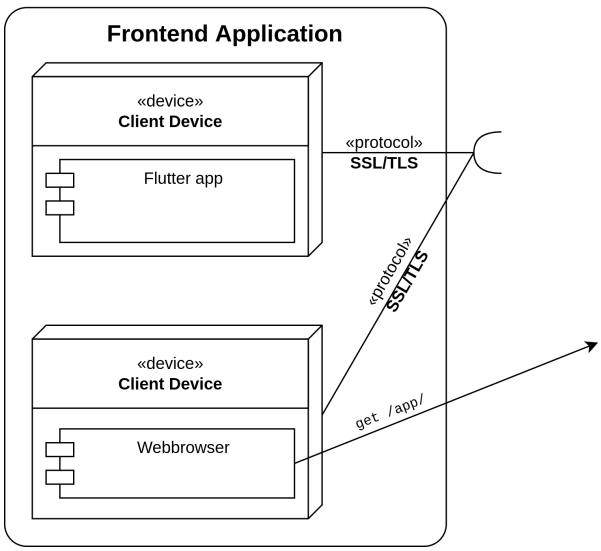

Figure 4: Deployment diagram of the frontend application.

To further explain this communication, the creation of a reservation is a good example. As soon as a user selects a garage, an overview of this garage is given. In order to show this page, a `GET` request is first sent to the URL of the garage, specified using the index of the garage (`/api/garage/<int:pk>`). This request receives all the information of the garage needed to construct the page shown. Later on, when the user has selected a time and date, in order to select a spot another request is sent to the backend. This time also a `GET` request, sent to the URL concerning the parking lots of a certain garage (`/api/parking-lots/<int:pk>`). A list of parking lots is returned, with their respective state and information. Based on this list, the frontend can build the page for spot selection. After the overview of the reservation, when the user confirms, a `POST` request is sent. This request arrives at the URL concerning the reservations of the user (`api/reservations`) and a

reservation is added to the database. Figure 20 in Appendix F gives a schematic overview of the different requests made with a sequence diagram.

The above requests are just examples of the many requests sent to the backend in the usage of the frontend. Since the latter is not functional without these requests, the communication between front- and backend is very important for the system.

3.4 Backend

The main functionalities of the backend are described in Section 2.1. This section describes the implementation of those core functionalities. Section 4 outlines the augments for the different software used in the backend.

The main parts of the backend described below can run on a single physical machine. The deployment of those services can happen both on a local server or on a cloud server. For the purpose of this project, a local home server is preferred, but a real-world system requires scalability which makes a cloud server indispensable.

3.4.1 Server

The server is the main entry point to the outside world of the database and the REST API and thus needs proper security measures. The encryption of traffic happens on the server, origin validation to prevent CSRF-attacks is included in the backend application (see Section 3.4.2).

Nginx is used as the main HyperText Transport Protocol (HTTP)-server in the backend. It serves as an industry standard for a fast and lightweight server. The most important feature for the backend is that it can handle Secure Socket Layer (SSL)/Transport Layer Security (TLS) and redirect HTTP-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022]. Furthermore, the server has to redirect incoming requests to the right application, namely, the web variant of the frontend application or the backend application. This is achieved via a *proxy-pass*, which can redirect traffic from one server to another, based on certain conditions in the request (e.g. all URLs which starts with `api/` are redirected to the Gunicorn Python server (see below).

Besides a web server, the backend application needs a way to communicate between the web server and the actual application (in casu the Django application). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[8] as its WSGI. The main purpose of the WSGI is making the deployment more stable and faster. The former is achieved by running multiple instances of the Django application, which improves the overall availability of the system [Chakon, 2017]. Besides improving the deploy stability, the WSGI makes it possible to use a Nginx server as reverse proxy for redirecting HTTP to HTTPS-traffic. This is not possible without a WSGI.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote machine. The backend is therefore deployed with Docker containers, which bundles *Docker images* with an OS in a so-called isolated *container*. In this way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup on the server machine. In total, there are three containers, one for the Nginx server, one for the Gunicorn gateway which runs the Django application and one for the MySQL database. The containers run as a single service with Docker Compose, which makes communication between the different containers effortless [Docker, 2022]. Figure 5 shows the deployment diagram of the entire backend.

3.4.2 Backend application

The backend server-side application is written in the Django framework. The Django framework serves as an Object Relational Mapper (ORM) for the database, which makes it much easier to query the database. The main purposes of the backend application is to retrieve the right information from the database and more importantly, validate if the user is authenticated and authorised to query that information. The database itself is a MySQL, relational database.[9]

3.4.2.1 User authentication

User authentication is an essential part of the IPS in general, due to the fact that this is the primary defence against unauthorised querying of the API. The authentication system in the backend consists out of three *views*: registering a new user, logging in a user and logging out an existing user.

[8] `https://gunicorn.org/`
[9] `https://www.mysql.com/`

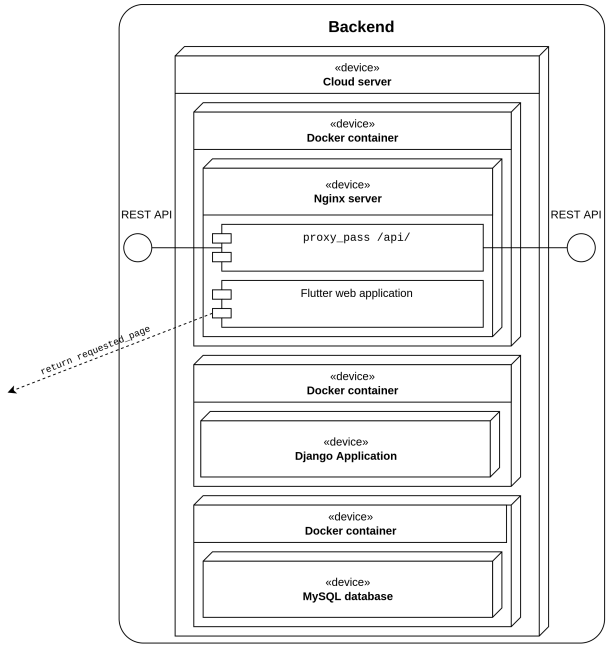

Figure 5: Deployment diagram of the backend server software.

In general, there are three levels of authentication for a user in the backend: *unauthenticated*, *authenticated* and *authenticated and verified*. When a user is not logged is, it is called unauthenticated. If the user has 2FA enabled and has logged in (meaning an authentication token exists in the database, see further), but has not yet entered their 2FA-code, it is authenticated, but not yet verified and thus not able to perform any requests. Only if the user has both authenticated and has verified its 2FA-code it is able to perform requests.

If the user creates a new account with the frontend application, their record will be added to the `users`-table, but this account is not yet functional. Upon registration an email is sent to the user with an activation link. Once the user has clicked on this link, their account will be activated and can be used to log in.

A user can log in via the frontend application which sends the entered `email` and `password` to the backend via an encrypted SSL/TLS-connection. The backend validates the credentials and if they are correct, an *authentication token* is generated and sent back to the frontend. This token has to be used in *all* API-requests which query or pass data from and to the database. Furthermore, this token indicates if a user is logged in or not. If a token is present in the `knox_authtoken`-table, a user is considered logged in. To prevent attackers from logging in with this token from a separate device, the number of tokens is limited to one token per user. If someone tries to log in with the credentials of an already logged in user, an error is returned. This token also makes it possible for the frontend to implement an automatic login system for the users.

From the frontend application, users are able to log themselves out. From a backend's perspective, this means deleting the authentication token from the database. In order for the logout to succeed, the authentication token has to be sent with the request. This prevents unauthorised logging out of users.

3.4.2.2 Serialization of database models

All data is transmitted in JSON-format in the API, but it cannot be directly inserted in this format in the database. One of the tasks of the backend application is *serializing* the data between the JSON- and database formats. This not only happens when data is queried from the database, but also when data is posted to the API. In this direction, the serializing function becomes more important, because it *validates* the sent data. The data has to match the requirements of the schema of the database (e.g. certain columns of the database are non-null, others have to be unique, etc.). If not, an error message is returned to the sender and the record will not be stored. This procedure is also the main defence against Structured Query Language (SQL)-injection.

On the backend level, models are related to each other, which makes it possible to build complex data structures. Possible relations are `has_many` or `has_one`, which indicates that a parent model can have multiple or only one child model, respectively. Figure 6 shows a schematic overview of these relations.

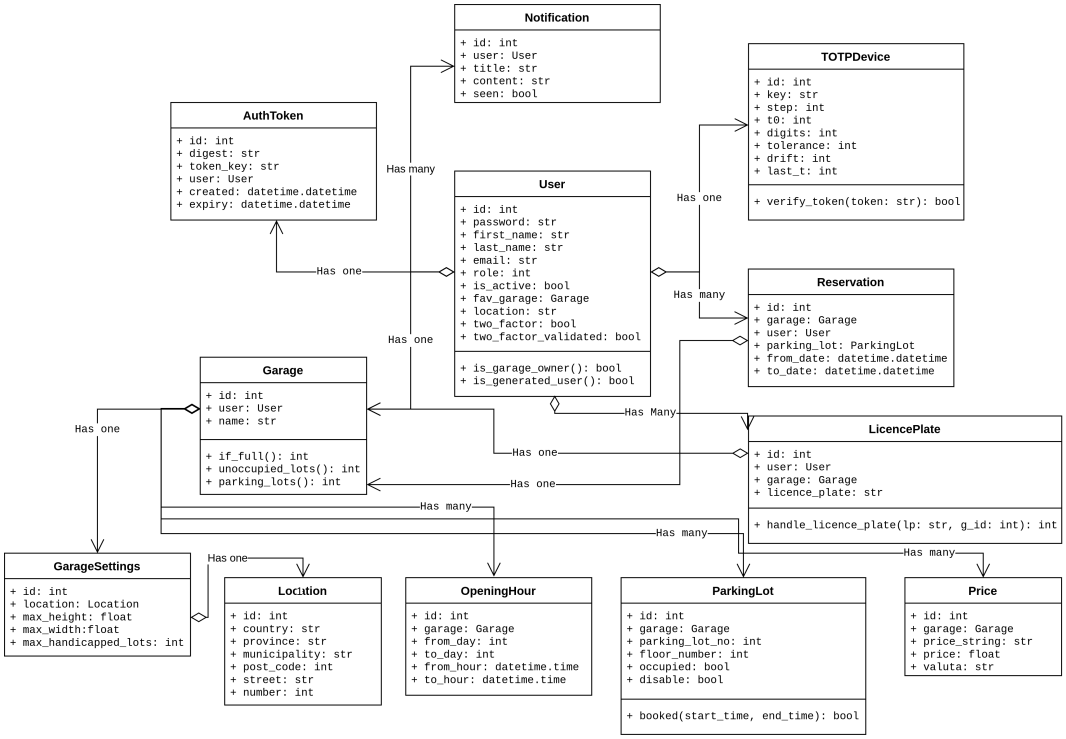

Figure 6: Class diagram of the backend, which shows the relations between the different models. Note that the relations are only marked one-way, while they exist both ways in practice.

3.4.2.3 User roles

All users in the database have one specific role, which is associated with a set of *permissions* for this user. There are three roles: `GENERATED_USER`, `NORMAL_USER` and `GARAGE_OWNER`. Upon registration with the frontend application, the default role is `NORMAL_USER`. In order for a user to register as a `GARAGE_OWNER`, the user has to provide the necessary legal documents which proves its ownership of a garage. The set of permissions of the `GARAGE_OWNER` is a superset of the user permission set, with the added permissions of adding and deleting garages and adding, disabling and deleting parking lots.

The `GENERATED_USER` is a special role which is created upon entering of a user without an account in the parking garage. The ANPR-cameras send registered text of the licence plate to the backend. Then the backend checks if the licence plate is already registered in the database. If not, a new dummy user, with the role `GENERATED_USER` is created with the associated licence plate. This account only serves as a visualisation of the parameters of the user's park (e.g. duration) and has no functionalities which are found in a normal user's account. Upon exiting the garage, the backend checks the associated role of the user; if it is a `GENERATED_USER`, the account – with the associated licence plate – is deleted.

A user can also be banned from using the system, setting its `role`-attribute to inactive. When a user does not show up at a reservation for a third time, its account will subsequently be banned. With each no-show, a notification in the application will notify the user about the imminent banning of its account.

3.4.2.4 Automatic number plate recognition

The backend implements ANPR to identify cars coming in and out of the garage. It is performed on the image which is sent to the backend via the Raspberry Pi (see Section 3.2.1.1). Its goal is to reliably detect the licence plate from different angles, light conditions and quality of the image. There are multiple ways to solve this problem. It is possible to use machine learning, however this requires a large dataset of good quality images to train the model. Creating these datasets takes a lot of resources, so this approach is out of the scope for this project. Another way to detect the text on the licence plates is to firstly find the licence plate on the image and then read the text in the region of the licence plate [Rosebrock, 2022]. The advantage is that there are already publicly available Optical Character Recognition (OCR) tools to find text in images.

The licence plates can be located because of their predictable shape. They are rectangular with a certain range of possible aspect ratios and have mostly contrasting colours. For this purpose, an algorithm can find

candidates of licence plates by converting the image to greyscale and applying a threshold to it. The image now only has black and white pixels, indicating the dark and light regions. In these pixels it can then look for rectangular regions. Out of these candidates, the algorithm can select the real licence plate based of the (amount of) text on it. It can also filter them based on given licence plate formats. In Belgium, for example, most licence plates have the '1-ABC-123' format. The filters can be changed easily because it is an argument to the OCR-function. This approach will not work in real applications because of unknown formats or custom licence plates but this approach is sufficient for this project.

To read the text on the licence plates, two OCR tools are used: EasyOCR [JaidedAI, 2022] and Google Vision API [Google, 2022]. EasyOCR is free but less accurate, this is perfect for filtering out the licence plate candidates without any text on it. Google Vision API is a paid service, it costs $1.5 for 1000 requests with 1000 free requests per month, but it is almost error-free. Google Vision is only used on the final candidate to stay below these 1000 requests.

3.4.2.5 Payment

The backend supports two main payment options: manual payments and automatic payments. For the manual payments, users can use all mainstream payment methods (e.g. Bancontact, iDeal), for automatic payments only card payments are supported. All payments are handled by the Stripe platform.[10] They provide simple integration of more payment methods, as well as an easy to use dashboard to get an overview of all payments. This project uses their checkout (manual payments) and invoice (automatic payments) APIs.

When a user wants to pay, the backend has to know how much it should charge, regardless of the payment method. It calculates the required amount based on the prices of the garage stored in the database and the duration for which a user is parked. The garage can have multiple prices, each based on a different parking duration. To get the best price the algorithm starts with the price with the longest duration and subtracts it from the parking time as much as possible, then the same thing happens for the next price and so on. For example, when there are prices for 1 day, 8 hours and 1 hour present in the database, and the user parked for 12 hours, they have to pay the price for 8 hours once and the price for 1 hour four times. The cumulative amount is then charged to the user.

The manual checkout can be used by all users. When a request for manual payment is sent to the backend, it sends all the prices and their quantities (as explained above) to Stripe. Stripe subsequently creates a checkout URL, which is then returned by the backend as well. Finally, the frontend redirects the user the checkout page using the requested URL. Figure 21 in Appendix F visualises this schematically with a sequence diagram.

To use automatic payments, a user has to add a card to their account. This can be done by sending the necessary details (card number, expiration date and Card Validation Code (CVC) code) to the backend.[11] After that, a new customer is created on Stripe, this is necessary for storing the new card payment method. When a user with a connected card tries to leave the garage, the backend can automatically charge the card using the invoice API and the customer on Stripe. Figure 22 in Appendix F displays this process schematically with a sequence diagram.

The backend can detect succeeded or failed payment attempts using *webhooks*.[12] This means that *Stripe* will make a request to the backend API for each new event. When the payment was successful, the licence plate of the user is updated so that it can leave the garage. If the payment failed, the backend sends an email to the user with a new payment URL.

The frontend application also allows garage owners to make an account with Stripe and so receive the payments from the users who paid in the garage.

3.4.3 Security

As explained in Section 2.1 security is one of the most important aspects of the backend. Of course, security is not limited to the backend alone, as it is often a conjunction between the frontend and the backend
There are numerous attacks possible against a backend service, so it is not possible to list them all. The following sections will describe the most common vulnerabilities and how they are prevented. Table 3 gives a schematic overview of the following paragraphs.

[10] `https://stripe.com`

[11] Note that the backend *does not* store these credentials. They are only needed to create a customer at Stripe, where after they are deleted from the backend systems.

[12] Webhooks are functions which can be called based on certain events happening on the web page, e.g. a user making a payment.

3.4.3.1 Injections attacks

Generally, during an injection attack, an attacker will try to inject malicious input in the backend system, most often done via the frontend application, which allows user to input data [IBM, 2021]. The most common types are Cross Site Scripting (XSS), in which the attacker tries to input (most-often JS) code into the remote server and SQL injection, in which the attacker tries to input SQL-commands into the server.[13] If any of these succeed, it can lead to the Remote Code Execution (RCE), which makes the server vulnerable for leaking sensitive information or to redirect users to the malicious sites. Depending on the severity, an attacker is able to perform any command on the server. The primary defence against injection attacks is the sanitising of user input.[14] This includes escaping any non-alphabetic characters. A second key point is setting a max-length on *all* input fields, in which the user can enter data. This specifically prevents database corruption through memory corruption. The use of Docker container, provides an isolated environment for the backend to run in. This reduces the risk of command injection attacks.

3.4.3.2 Access Control Misconfigurations (ACMs)

From a backend's perspective, there are three Access Control Levels (ACLs), associated with the roles, described in Section 3.4.2.3. Each ACL has a strict associated set of permissions, which prevents that users perform actions which they should not perform (e.g. altering information about a garage) and that users cannot alter objects, on which they have no ownership (e.g. altering information about a different user). The backend is designed in such a way that requests that the user parameter is determined automatically from the authentication token, without sending it directly with the request. This prevents IDOR and the malicious editing of objects.

Apart from determining the user automatically from the request, the backend implements an additional set of permissions to the role `GARAGE_OWNER`. They are allowed to add garages and parking lots to the system, as well as to alter or delete garage and parking lots which they own.

3.4.3.3 Request Forgeries

CSRF-attacks and Server-Side Request Forgery (SSRF)-attacks are different but related attacks, which are caused by a lack of origin validation with the server. In the former, an attacker misuses the user credentials to perform actions as if the attacker was the user, in the latter the attacker targets services which are running internally on the server, causing it to leak sensitive information.

The backend implements a rigorous origin-validation system, with the use of JSON Web Tokens (JWT)-tokens.[15,16] Each time a request is made, the frontend application will generate a new JWT-token. It is signed with a secret key only known to the backend server and the frontend application. The JWT-token only lives for 5 seconds, preventing the capture and reuse of these tokens.

Based on the deployment diagram, the backend has to be able to accept requests from two different origins: the local garage system and the frontend application. All requests from any other location are rejected. This prevents both types of attacks mentioned above: if the attacker gets hold of the user's authentication token, he/she will not be able to perform any requests with it, as only the frontend application can send valid requests to the backend server. Furthermore, an attacker will not be able to gain access to internal services running on the server as it is only accessible via the frontend application, which can only interact with the Django framework.

The origin-validation of local garage system proceeds in a similar fashion, but the JWT-token has no expiry, as it cannot be intercepted as the data is sent via HTTPS and thus the sent data as well as the cookies are encrypted. This further prevents Man In The Middle (MITM)-attacks.

3.4.3.4 Other security measures

Apart from the above measures taken to enhance security of the backend, the backend also obliges users to choose strong passwords[17] and allows the use of 2FA. Once the user is logged in, he/she can add a Timed One

[13] Note that there are many more possible injection attacks, like buffer overflows, format string attacks and command injection, etc.

[14] Note the difference between *serializing* (see Section 3.4.2.2) and *sanitising* data. The former is needed for compatibility between the frontend and the backend, the latter is an essential tool for preventing injection attacks.

[15] `https://jwt.io`

[16] Note that this token is different from the authentication token which is used for authentication the uesr.

[17] The backend demands that a password is at least 10 characters long, contains a capital letter, a numerical symbol and a special character and may not be associated with any user information.

Time Password (TOTP)-device. The backend then returns a security code which the user can enter in Third-Party Authenticator (TPA) like Google Authenticator.[18] From then on the user has to be both authenticated and verified to perform any requests.

The backend also requires the users to validate their email address given at sign up. If the user signs up, their object will be added to the database, but it will be inactive. Users will not be able to log in until they activate their account. This limits the creation of accounts with false email addresses.

Table 3: Overview of the backend's security measures against different attacks.

Attack	How prevented?
SQL injection	Serializing all user input.
XSS	Serializing all user input (also in the frontend application).
IDOR	User data requests return only info about the current user.
ACM	Strict ACLs with added permissions.
CSRF	Use of a strict origin validation policy with JWT-tokens.
SSRF	Use of a strict origin validation policy with JWT-tokens.
MITM	Use of HTTPS.
Brute force	Rigorous password policy
Stolen password	Possibility of 2FA.

3.4.4 Privacy

Besides security, privacy is another core functionality of the IPS. As the backend is responsible for storing sensitive information about the user, it is most suited to build a solid privacy policy.

3.4.4.1 Least to know principle

The backend implements the so-called 'least to know principle', which indicates that it only stores the information it actually needs to function properly. All other data is either not stored in the backend data (e.g. credit card details of the user) or deleted from the moment the data becomes irrelevant (e.g. information about a user's reservation is deleted two hours after the reservation is finished). Even in the event of a data breach, no relevant information about a user can be obtained. Other sensitive information, like passwords are hashed before they are stored in the database.

3.4.4.2 Licence plate validation

Some current parking applications make it possible for people to be followed based upon their licence plate [Verheyden, 2022]. This is a major privacy breach. In the frontend application, users can add a licence plate to their account, but it is not activated (and cannot be used to make reservations) until they submit a valid vehicle registration document proving the ownership of their licence plate. Furthermore, the `licence_plate`-column has to be unique, meaning that it is impossible for different users to add the same licence plate to their account. The frontend application provides an additional possibility to report a licence plate if a user thinks that someone else has registered their licence plate.

3.4.4.3 Anonymous use

The backend also fully supports anonymous use of the IPS, without giving in to user-friendliness. It is possible that a user drives towards a garage, without having the account and without having the application installed and yet have a similar parking experience to users with an account and in complete anonymity. In the case described before, the backend will generate a user with the associated detected licence plate. The local garage system will print out a paper ticket with a Quick Response (QR)-code on it. Scanning this QR-code opens the frontend application, hosted on the Nginx-server, with the user already logged in with the generated account. This account allows the user to see in which garage they are parked, the time they are parked and a way to pay the due amount. Upon exiting of the garage, this account, with the associated licence plate will be fully deleted from the database. From a backend's perspective, the park ceases to exist to moment the user exits the garage, which renders it untraceable.

[18] `https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&gl=US`

4 Design motivation

The design discussed in this report was not chosen without careful consideration of all options. The first decision made, was which micro controller to use. Common types of micro controllers such as a Raspberry Pi or an Arduino are readily available and are suitable for the project's needs. The Raspberry Pi 3B was used because the department had it available.

Secondly, the detection of entering or leaving cars can be done by cameras with motion detection software. This means that the camera is constantly running and detects whether or not there is any movement. The positive side of this method is that there is no need for any sensors or other extra hardware. The downside is that it might be too much to handle for the Raspberry Pi. An alternative option is working with sensors (e.g. a distance sensor) to detect the cars. This is less heavy for the Raspberry Pi, but adds an extra cost to the garage. Another option would be to leave the cameras filming and run the algorithm directly on the video footage. But this may also be too much to handle for the Raspberry Pi. Therefore, the option with the sensors is chosen for this project.

Thirdly, the detection of available parking spaces can be done by several sensors. The most useful ones are UDMS or light sensors. The latter is more expensive and does not offer any extra advantages over the UDMS. For this purpose the UDMSs (HC-SR04) are used in this project.

Fourthly, The backend application is written in Django, an open-source web framework, written in Python. The eventual decision was made based on the following concerns: 1) Python is known to all group members; 2) Django is a very explicit framework, which does not include a lot of magic features like Ruby on Rails does; 3) Django describes itself as the "framework for perfectionists with deadlines", which is exactly suited for the job [Django, 2022].

Lastly, the frontend application will be written in Dart, with the Flutter framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React or AngularJS). The main benefit for Flutter over the other frameworks is that it can run on any operation system (Android, iOS, MacOS, Linux, Windows, etc.), that it provides type safety and null safety (Dart is a strongly typed language) and that it supports hot reloads, which makes development much easier [Flutter, 2022]. Furthermore, two of the team members already worked with Flutter.

Of course the price of the different components played also a big part in the decision making. The prices of the individual components and the total price can be found in Table 4. The leftmost column gives the name of the component used in the model. The second column shows how many pieces of that component are needed. Column 3 shows the price per piece and column 4 the total price for a specific component. The budget for this project is 250 euros. The design uses a total of 224.87 euros, displayed in the second to last row of Table 4. This indicates that the limit of the budget is nearly reached with a surplus of 25.13 euros.

Table 4: Final budget overview.

Component	**Amount**	**Price/piece**	**Total**
DORHEA Raspberry Pi Mini Camera	2	11.95	23.9
Ultrasonic Module Distance	8	3.95	31.6
MDF plates 6mm	3	2.4	7.2
Green LED lights	6	0.35	2.1
Red LED lights	6	0.33	1.98
Resistors	12	0.2	2.4
Raspberry Pi extension cable	2	4.99	9.98
Micro Servo Motor	2	7.21	14.42
Raspberry Pi 3B V1.2	2	59.95	119.9
LCD-Screen	1	1.49	1.49
Jumper cables (20 pieces)	2	4.95	9.9
Total Price		224.87	
Remaining		25.13	

5 Case example

Earlier, this report gave an abstract explanation of how the system should work and the more concrete implementation using the different components of both the software and the hard system. Subsequently, this section gives a real-world example of a user experience in the parking garage. Due to the large size of the flowcharts, they are included in Appendix G.

The process begins with the user who drives towards the entry barrier. An UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of the licence plate. This picture is then analyzed by the Google Vision API. Then the recognised string from the licence plate is sent to the backend with a `POST`-request to `api/plate`. The system supports two use cases: either the user has a registered account with a licence plate, or the user has not. In both cases the user should be able to use the parking garage. The backend checks which of the two cases the received licence plate falls in. In the former, the barrier will be opened and the `licence_plate`-table is updated to include the time of arrival. In the latter case, the backend will create a new user account and print a paper ticket with a QR-code which contains a link to the created account. With this dummy account the user can view all the information about his park. This dummy account is deleted when the user exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 23 in Appendix G shows a schematic overview of the entering process.

After entering the garage, the user drives to the pre-booked parking lot, in which case the occupancy of the parking lot is already set to `True` or to a parking lot of choice. In the latter case, an UDMS detects the cars, so that the Raspberry Pi can sent an API-request to the backend to update the respective table. Note that only if the parking lot is booked, the parking lot is associated with the licence plate and thus with the user. Figure 24 in Appendix G shows a schematic overview of this process.

When exiting the garage, it is recommended that users pays their tickets in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier for users who might have forgotten to pay.

In a similar way as when entering, a UDMS detects the car and the ANPR-camera takes a picture, which is sent to the Google Vision API for analysis. Subsequently, the recognised text is sent to the backend via the same URL. The backend can distinguish the images for entering and exiting the garage via the `licence_plates`-tables which stores whether a licence plate is currently inside the garage. This table also contains a column which indicates if the user connected to the licence plate has already paid. If this is the case, the barrier will open. In the opposite case, there are two possibilities: the user has an account which supports automatic payments, in which case the payment will happen in situ and the barrier will open consequently. In the case in which the user has not paid, nor has an account which supports automatic payment, a paper ticket will be printed with a QR-code which redirects the user to a payment-environment. Once the user has paid its ticket, the barrier will open. Figure 25 in Appendix G shows a schematic overview of the exiting process.

Note that the need of paper tickets is not fully eliminated in this user flow, but is only used as a back-up system if the user does not have an account or forgot to pay. In both cases, the user will be able to use the parking garage with almost the same features as a user who installed the application.

Garage owners can change all the settings of their garage. This applies to non-essential settings (like the amount of parking lots for electric cars or the maximum height for cars in the garage) as well as too crucial ones like the location or prices of the garage. Garage owners can do this in the same app as their customers. For them, there is an extra list with all their owned garages. By selecting a garage, they can change the settings on a custom settings page. When they confirm their changes, they will also show up on the app of the customers.

6 System analysis

This section critically analyses the IPS, discusses its strengths and weaknesses and which possible solutions could be implemented to solve some of the issues still in the system. This section is split into three parts, covering the general system, the frontend application and the backend application.

6.1 General

In general the IPS does all the things that are expected of it and works as intended. However this is assuming that the user behaves as expected and follows all the guidelines set be the parking garage. To a certain extent, the users have to be trusted, which inevitably creates problems.

Firstly, the system assumes that people won't park in a reserved spot if the indicator LED is red. However it is possible that users do park illegitimately. In that case, the system automatically changes the user's reservation to a parking lot that is still empty at the time of the their reservation. From the moment that all parking lots in the garage are occupied (either by a car parking on them or by a reservation) the LCD-screen at the entrance will show that the parking garage is full. The system will then no longer allow any users inside, except for the users which made a reservation in that time period. This assures a free parking lot for users who made a reservation.

Secondly there are some issues with the Raspberry Pi devices and its connection with the backend. Due to the fact that these devices are not able to connect directly to the Wireless Fidelity (WiFi)-network of the KU Leuven they have to connect to a hot-spot created locally. This results in large delays in the communication with the backend. Often it can take up to 20 seconds to send the taken picture to the backend to be processed. This is slowing down the entire system. A dedicated LAN network, setup with routers at the garage location would resolve this issue.

Thirdly, the infrastructure at the local garage is vulnerable to vandalism and/or system malfunctions. The sensors or the ANPR-cameras can be covered and so given none or faulty signals to the backend, which renders them obsolete. Providing sufficient back-up systems and making vandalism more difficult can reduce the frequency of the problem, but never totally avoid it.

Fourthly, the current prototype only implements a single LCD-screen which can show information to the user. This is insufficient and at least one extra screen is needed at the exit of the garage. If users drive to the exit barrier without having paid, it will not open. In that case, a LCD screen showing the user exactly why the barrier will not open is indispensable. In an ideal case, the local system is will show when it could not reach the backend (see following paragraph) to inform the users about this issue.

Lastly, the local garage system as well as the frontend application are totally dependent on the backend for functioning correctly. If the backend becomes unreachable (either by a malfunction in the local LAN-network or in the backend server), all the sensors in the local garage system become obsolete and the frontend application will no longer be functional. This again stresses the importance of high-availability of the backend infrastructure. Again, it is not possible to fully eliminate the chance of having a backend crash, only to reduce it by deploying several instances of the backend on multiple servers (e.g. with a container orchestration tool like Kubernetes[19]) and to implement Application Performance Monitoring (APM)-tools like Datadog[20], which would report any down-service immediately, such that appropriate actions can be taken.

Besides the previous mentioned points, the current implementation of the local garage system uses copper wires to connect the Raspberry Pi devices to the electronic equipment. This is not scalable the large parking facilities with hundreds or even thousands of parking lots. In that case, the communication between the Raspberry Pi and the electronic sensors should happen via a LAN-network, e.g. adding a router to set up a dedicated WiFi-network for the local garage.

6.2 Frontend

The main strength of the frontend application is its user-friendliness. The user can automatically pay upon leaving the garage and check the occupancy of a garage as well as reserve a spot in advance. However downloading the application and creating an account is not necessary to use the garage. a user will still be able to park but they won't be able to take advantage of the app's features.

One shortcoming of the system is that the user is the required internet connection in order to interact with the application. A possible solution to become less reliant on the user's internet connection is to create a caching system that stores the changes on the client device and uploads them to the backend when the user regains an internet connection.

Besides needing an internet connection, the application also makes many requests to the backend server, making the applications slow overall. The user has to wait for each request to finish before he/she can continue using the application. This can be reduced by implementing a secondary faster database like Redis[21], which

[19] https://kubernetes.io/de/
[20] https://www.datadoghq.com/
[21] `https://redis.io/`

can be deployed using a Content Delivery Network (CDN), which minimises the response time for request made by deploying multiple instances of the database on different servers across the world.

6.3 Backend

The backend prevents unauthorised access of sensitive user data. It does this in several ways. Firstly by not storing any information that is not needed and hashing sensitive user data before it is stored in the backend and secondly by making it difficult, as not impossible for unauthorised people to gain access to the backend, see Section 3.4.3.

The biggest shortcoming of the current backend server is the high latency for requests. This is especially detrimental for the entering and exiting system of the garage, as users have to wait in front of the barrier in order for the request to succeed. Moving the server from a local home server to a cloud based cluster of servers improves the latency and decreases the change of a server failure.

Security-wise, the current implementation of the backend is vulnerable to Denial of Service (DOS) and Distributed Denial of Server (DDOS)-attacks. The implemented origin validation (see Section 3.4.3) does not prevent a multitude of requests sent to the domain of the backend. In order to prevent DOS and DDOS-attacks, the backend system should be very scalable, which the current implementation is only to a certain extent, as it runs a single machine.[22] Distribution of the server load on multiple servers across the world and using a CDN will help to lower the risks of a successful DOS or DDOS-attack. Besides that, using APM-tools also helps to detect attempts of these attacks faster [DSM, 2020].

6.4 Scalability

This section describes shortly the possibilities to expand this system to extent it beyond the current implementation.

Both the frontend and the backend are already for handling a large number garages, each with a great amount of parking lots. Of course, like explained in the previous paragraphs, the current system has to be optimised for handling an increasing number of requests, this by deploying the backend to a cloud-based server facility, using client-side cashing, as well as a secondary (in-memory) database and using a CDN.

The local garage system in its current form is not yet scalable. Parking garages with hundreds of even thousands of parking lots cannot be handled by a handful of IoT-devices. A more powerful on-site gateway, which will collect and transport the data to the backend, then becomes indispensable. Like mentioned earlier, a dedicated LAN-network also becomes necessary, as cable wiring is too expensive and cumbersome.

7 Conclusion

The main goal of this project was to design an IoT-infrastructure which makes parking easier, faster and foremost safer. The main difficulty of designing this infrastructure, was the requirement of it being a safe system to park a car without violating the users privacy.

The IPS proves to be a safe and privacy-friendly way of parking, which is easy to use, both for regular users and garage owners. The main system consists out of three main parts: the local garage system, a frontend application and a backend.

The local garage system is controlled by two IoT-devices, which collect data from multiple electronic sensors in the parking garage. Another thing that these devices provide, is the communication with the backend. The current obstacles for the local garage system are cable wiring and the lack of a reliable LAN-network

The backend – deployed on a single server – mainly provides a secure way to store and retrieve user information in a privacy friendly way. Besides that, the backend implements multiple measures to enhance overall security. High latency, as well as being vulnerable to severe DDOS attacks proved to be a challenge for the current implementation of the backend.

The frontend application is the main entry point for end-users to interact with the backend. With a multitude of flows, it provides a clear and comprehensible way for users to create reservations, view information about their current park and to pay the bills of the parking garage. The lacking of a proper cashing system introduces lag in the application.

[22] In the case of a severe DDOS-attacks, 1 terabit of information per second can flood the server, which it is unable to handle [Goodin, 2016]. Note that the backend *does* handle concurrency very well in normal use cases.

The current prototype of the entire system has to be improved and optimised in order to scale to a large number of user requests with in-memory database, client-side cashing and a dedicated LAN-network in the local garage system.

8 Course integration

This project is a sequel of its predecessors P&O 1 and P&O 2. So most knowledge and experience that's been used for this project came from these courses. In these courses items like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects needed for creating a good project. As mentioned earlier, the experience that has been gained from these courses is also very important. From these courses the skill of working in a team were developed, which was very important for this project and will remain important for future projects.

Furthermore, methodology of computer science was an important course for an introduction to programming and understanding complex algorithms. This course was taught in Python and this knowledge was needed for the Raspberry Pi and licence plate recognition. Along with Python, Dart was used and this language was easy to learn because of this course.

Just like methodology of computer science was used for the licence plate recognition, other courses like calculus and linear algebra were needed for neural networks. Calculus was useful for solving the optimisation problems in the neural network, to find the best solution. Linear algebra is used in the neural networks for solving large systems of linear equations.

Another course that was very useful, was technical drawing for creating our physical design in Solid Edge. In this course the skills were taught for creating a 3D-design of an object or product and understanding the 2D-drawings of it. Other knowledge that was used for the physical aspect of the project was circuit design. This was taught in P&O 2 and was used for the sensors and lights. For creating these electronic circuits, knowledge from the course electrical networks was also necessary. This course was needed for understanding how to connect different electronic components with each other.

Of course was the knowledge of all these courses not enough to make and realise this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[4411, 2022] 4411 (2022). Betaal je mobiliteit met één app. [Online]. Last accessed on 11/11/2022. Retrieved from https://4411.io/nl-be/.

[Chakon, 2017] Chakon, O. (2017). Deploy Django app with Nginx, Gunicorn, PostgreSQL & Supervisor. [Online]. Last accessed on 23/10/2022. Retrieved from https://hackernoon.com/deploy-django-app-with-nginx-gunicorn-postgresql-supervisor-9c6d556a25ac.

[Computer Hope, 2020] Computer Hope (2020). What is the difference between a 32-bit and 64-bit CPU? [Online]. Last accessed on 24/10/2022. Retrieved form https://www.computerhope.com/issues/ch001498.html.

[Decker and Hughes, 2022] Decker, B. D. and Hughes, D. (2022). Groepsopdracht voor het vak Probleemoplossen en Ontwerpen.

[Django, 2022] Django (2022). About the Django Software Foundation. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.djangoproject.com/foundation/.

[Docker, 2022] Docker (2022). Key features of Docker Compose. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.docker.com/compose/features-uses/.

[DSM, 2020] DSM (2020). DDOS Prevention. [Online]. Last accessed on 16/12/2022. Retrieved from https://www.dsm.net/it-solutions-blog/prevent-ddos-attacks.

[Goodin, 2016] Goodin, D. (2016). Record-breaking DDoS reportedly delivered by ¿145k hacked cameras. [Online]. Last accessed on 16/12/2022. Retrieved from https://arstechnica.com/information-technology/2016/09/botnet-of-145k-cameras-reportedly-deliver-internets-biggest-ddos-ever/.

[Google, 2022] Google (2022). Vision AI. [Online]. Last accesed on 26/10/2022. Retrieved from https://cloud.google.com/vision/.

[Gupta, 2022] Gupta, L. (2022). What is JSON? [Online]. Last accessed on 11/11/2022. Retrieved from https://restfulapi.net/introduction-to-json/.

[IBM, 2021] IBM (2021). Injection attacks. [Online]. Last accessed on 14/12/2022. Retrieved from https://www.ibm.com/docs/en/snips/4.6.0?topic=categories-injection-attacks.

[JaidedAI, 2022] JaidedAI (2022). Jaidedai/easyocr: Ready-to-use ocr with 80+ supported languages and all popular writing scripts including latin, chinese, arabic, devanagari, cyrillic and etc.. [Online GitHub Repository]. Last accessed on 26/10/2022. Retrieved form https://github.com/JaidedAI/EasyOCR.

[Nginx, 2022] Nginx (2022). About Nginx. [Online]. Last accessed on 23/10/2022. Retrieved from https://nginx.org/.

[Rosebrock, 2022] Rosebrock, A. (2022). OpenCV: Automatic License/Number Plate Recognition (ANPR) with python. [Online]. Last accessed on 26/10/2022. Retrieved from https://pyimagesearch.com/2020/09/21/opencv-automatic-license-number-plate-recognition-anpr-with-python/.

[STATBEL, 2022] STATBEL (2022). Wagenbezit per huishouden. [Online]. Last accessed on 11/11/2022. Retrieved from https://statbel.fgov.be/nl/themas/datalab/wagenbezit-huishouden#:~:text=37%2C3%25%20heeft%20%C3%A9%C3%A9n%20wagen,koppels%20met%20een%20inwonend%20kind.

[Verheyden, 2022] Verheyden, T. (2022). Ethische hacker waarschuwt voor privacylek in parkeerapps als 4411. [Online]. Last accessed on 11/11/2022. Retrieved from https://www.vrt.be/vrtnws/nl/2022/09/08/parkeerpapps/.

Appendices

A General deployment diagram

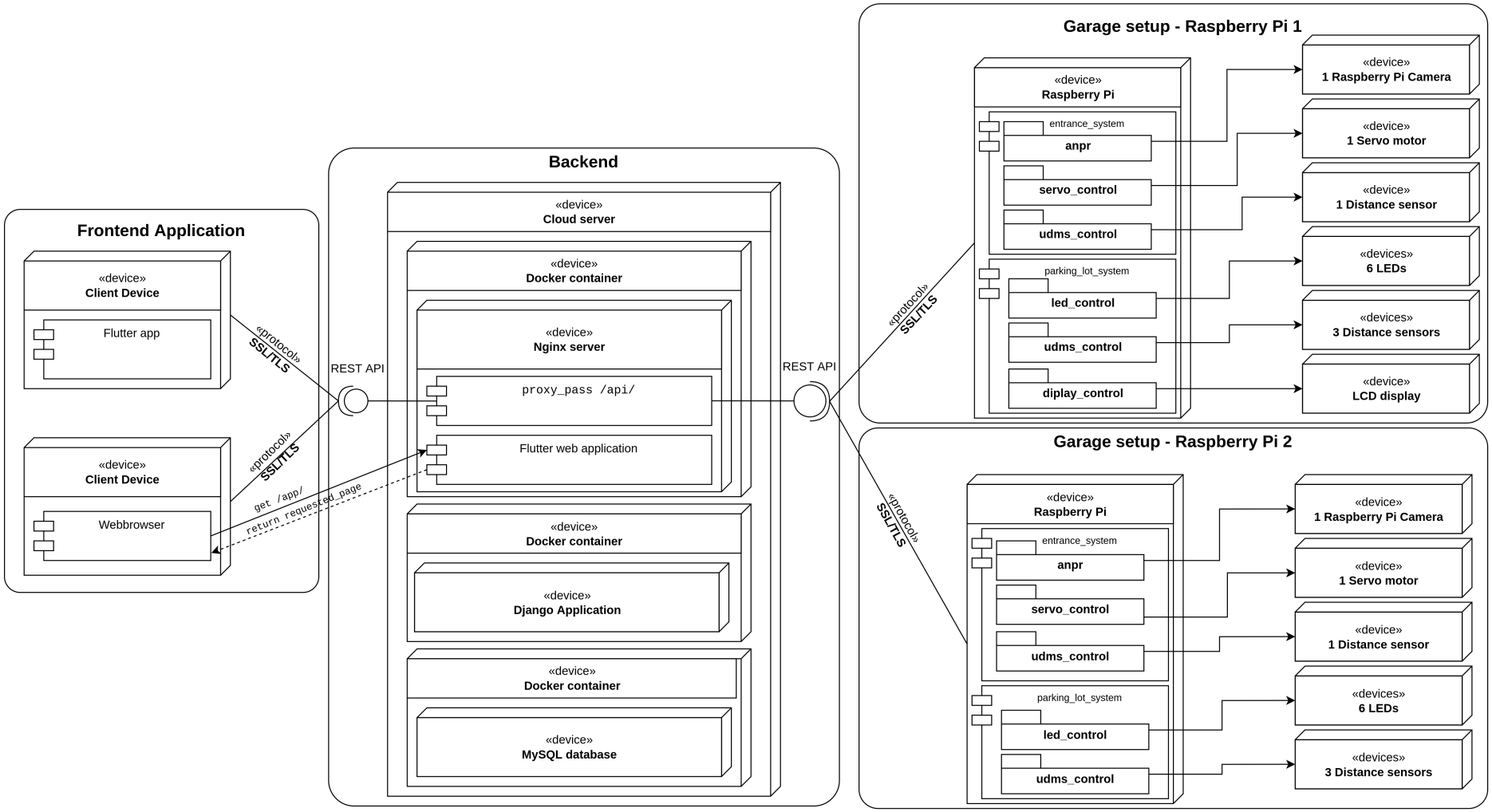

Figure 7: General deployment diagram of the entire IPS.

B App diagrams

Figure 8: General app diagram which displays the user flow within the frontend application. The pages are enumerated, where a new number indicates a change of flow. Pages with the same number belong to the same logic flow within the application. The routes of the back button are not indicated with an arrow as they represent the reverse direction of the arrow which points to this page.

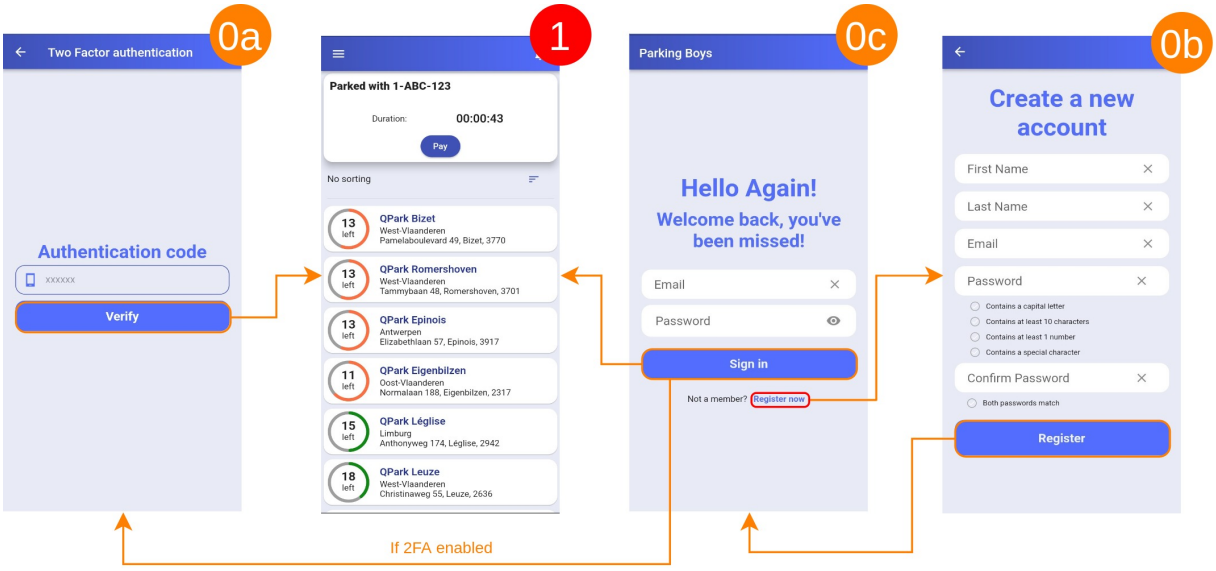

Figure 9: App diagram of the authentication flow (Flow 0).

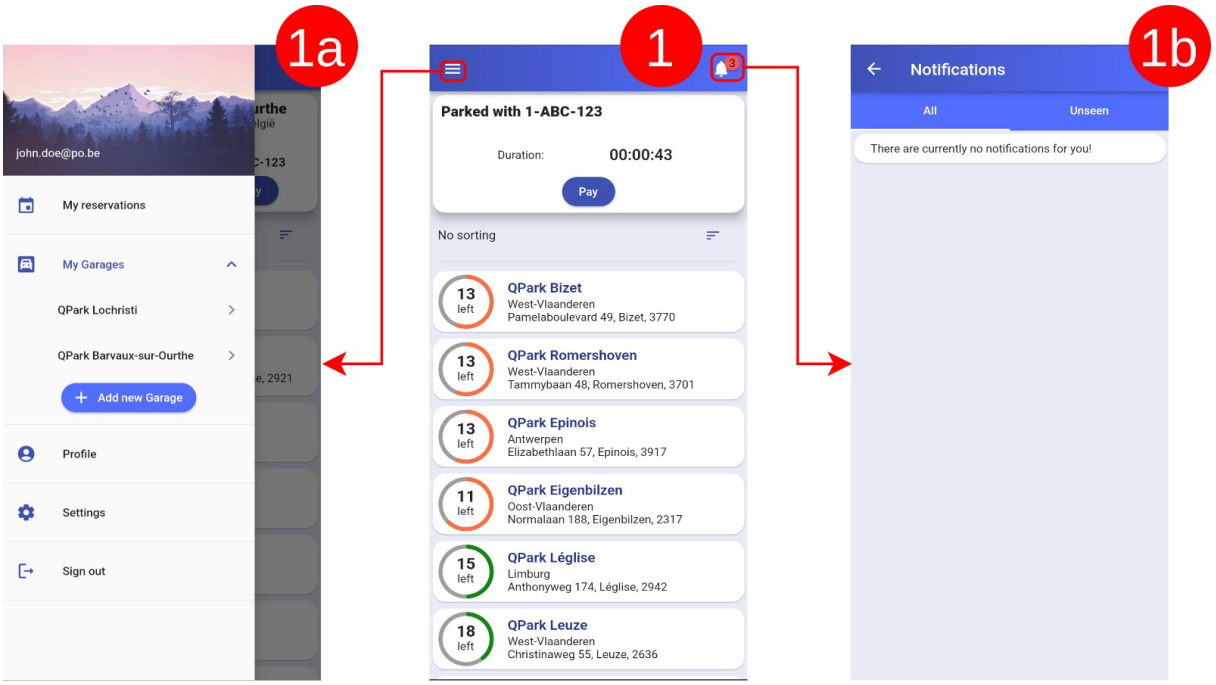

Figure 10: App diagram of the home flow (Flow 1).

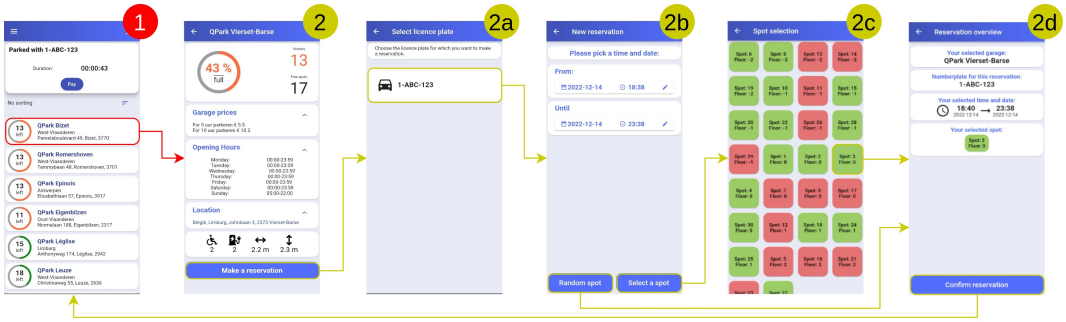

Figure 11: App diagram of the reservation flow (Flow 2).

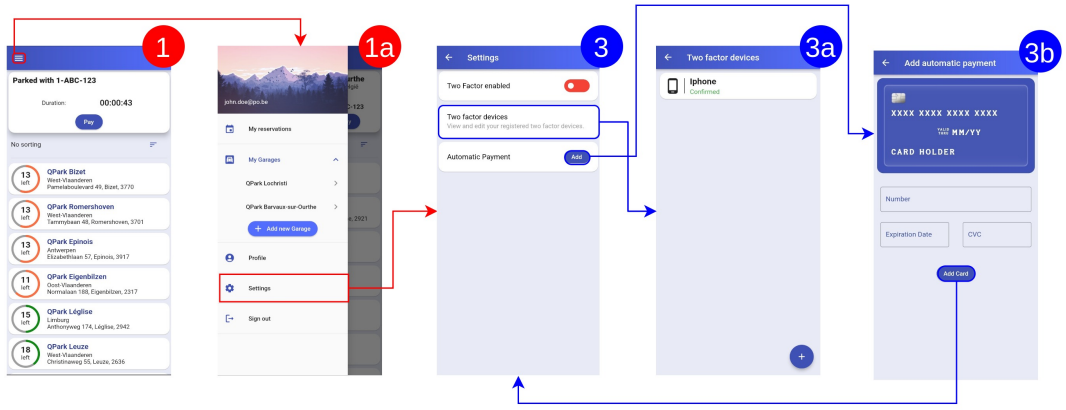

Figure 12: App diagram of the user settings flow (Flow 3).

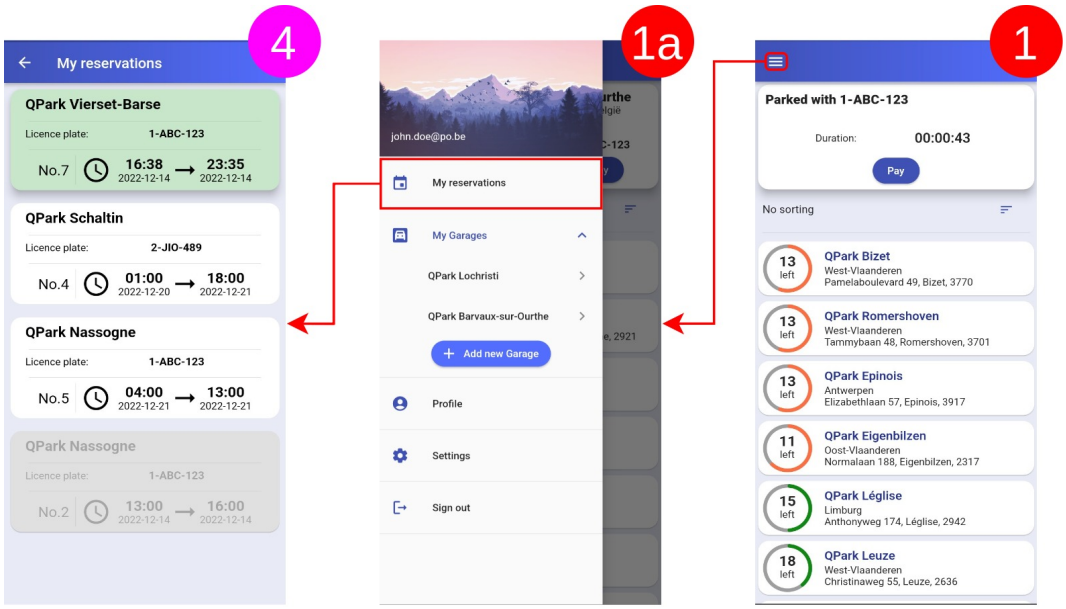

Figure 13: App diagram of the user reservation flow (Flow 4).

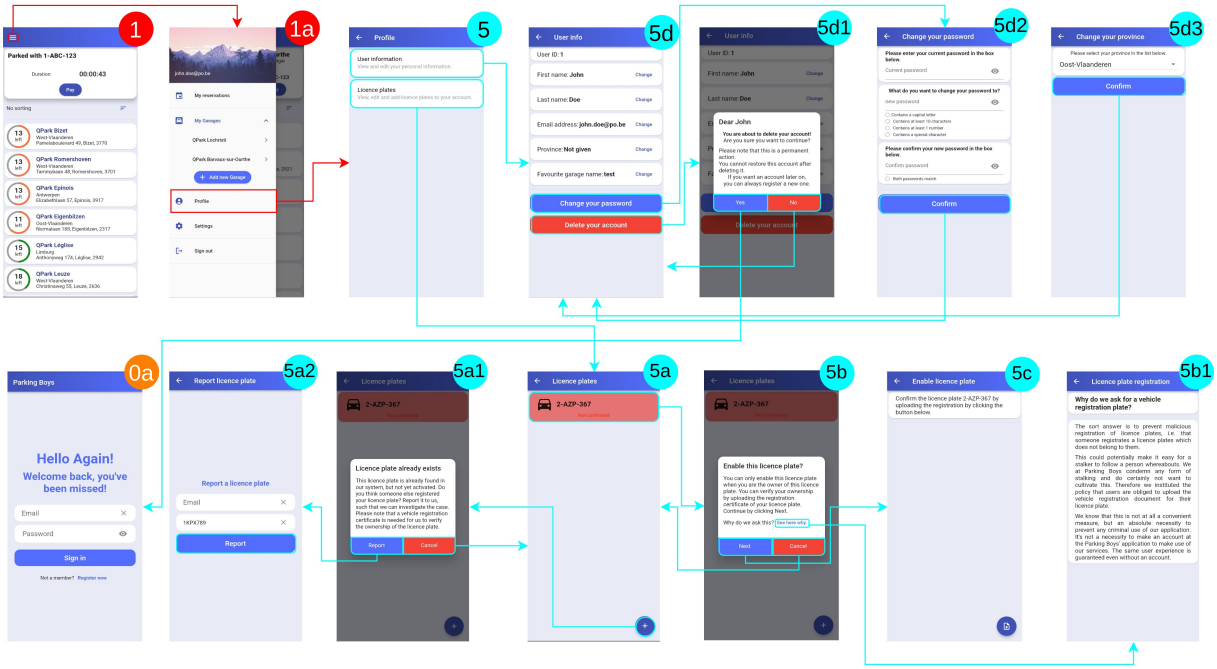

Figure 14: App diagram of the profile flow (Flow 5).

Figure 15: App diagram of the garage settings flow (Flow 6).

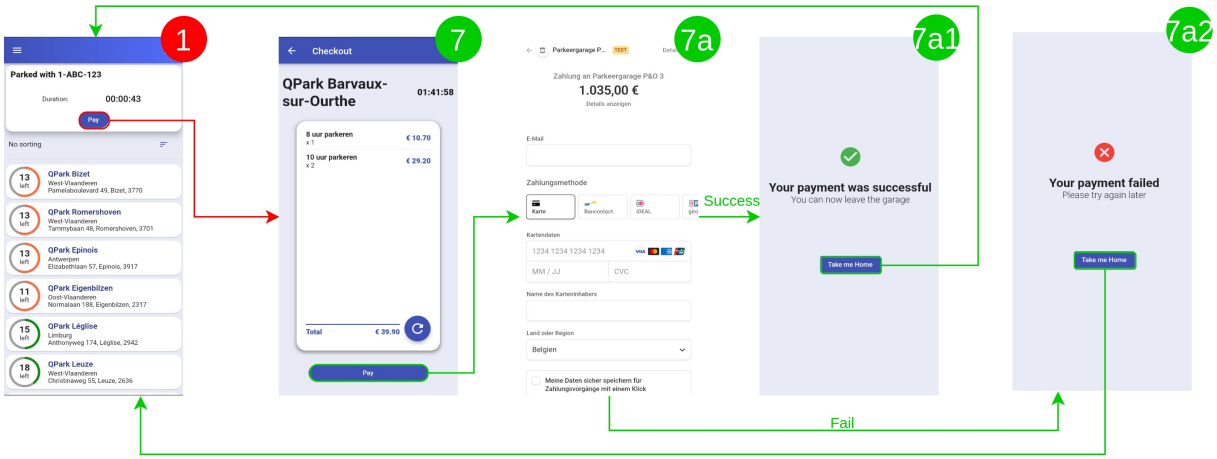

Figure 16: App diagram of the payment flow (Flow 7).

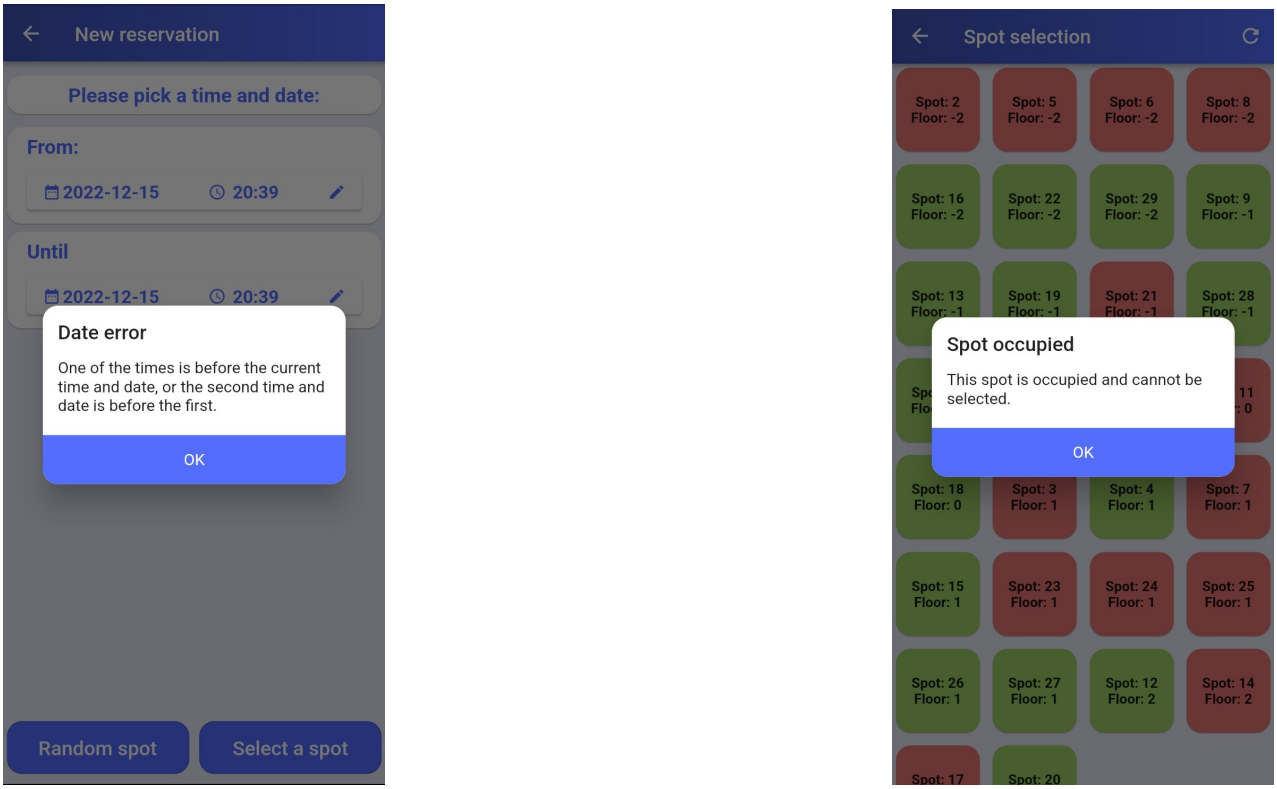

Figure 17: Examples of error pop ups in the frontend application.

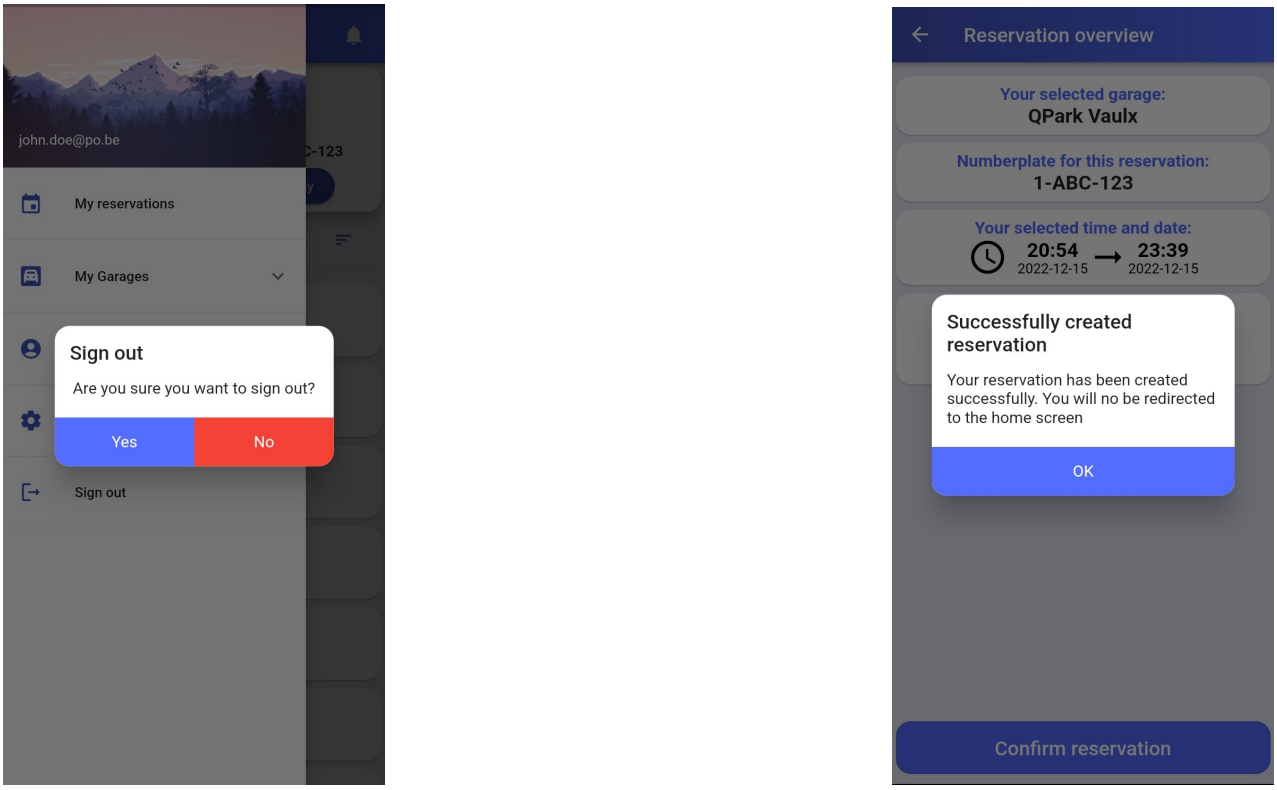

Figure 18: Examples of information pop ups in the frontend application.

C App pages

Table 5: An overview of all the different pages in the frontend application, together with their page code and a short description.

Page code	Page name	Short description
0a	Login page	Lets the user enter their email and password and authenticates it with the backend.
0b	Register page	Allows the creation of new accounts by inputting all necessary information by the user.
0c	2FA Page	Allows the user to enter the 2FA-code from their TOTP-device.
1	Home page	Central page in the application which is the entry point for all major flows.
1a	Home banner	Banner which allows the user to navigate to all important flows and to sign out.
2	Garage info page	Provides an overview of all information about a garage, including its occupancy, prices and opening hours. Is the entry point for the reservation flow.
2a	Licence plate selection page	Allows the user to select for which licence plate they want to make a reservation.
2b	Make reservation page	Allows the user to enter their start and end time of the reservation they want to make. Validates them before continuing.
2c	Spot selection page	Allows the user to choose a parking lot for which they want to make a reservation.
2d	Reservation overview page	Provides an overview of the reservation the user is about to make. On confirm, the user is redirected to the home page
3	Settings page	Entry point for the users altering the user settings.
3a	Two factor devices page	Allows the user to add, update or delete TOTP-devices.
3b	Add automatic payment page	Allows the user to enter their credit card details, which enables automatic payment.
4	User reservation page	Overview of the user's reservations, highlighting the current reservation and disabling past reservations.
5	Profile page	Entry point for altering and viewing user information.
5a	Licence plates page	Allows the user to view, add and confirm licence plates.
5b	Confirm licence plate dialog	Allows the user to choose to enable their licence plate.
5b1	Registration explication page	Explains the user why a vehicle registration document is necessary for the application to enable their licence plate.
5c	Confirm licence plate page	Allows the user to upload a vehicle registration document to enable their licence plate.
5d	User info page	Allows the user to view and edit their personal information.
5d1	User deletion pop up	Allows the user to choose if they want to delete their account.
5d2	Change password page	Allows the user to change their password by entering a new one.
5d3	Change province page	Allows the user to change their location.
6	Garage settings page	Allows a garage owner to alter the information about their garage, including the parking lots, prices, opening hours and location.
6a	Add prices page	Allows a garage owner to add new prices to their garage. The prices will be synchronised with the Stripe servers.
6b	Add garage page	Allows a garage owner to add a new garage to the system. This page also allows to add opening hours, prices and a location to the garage.

7	Checkout page	Provides an overview of the due amount and the parked time to the user.
7a	Payment page	Allows the user to pay their parking bill.
7a1	Payment success page	Provides confirmation to the user that their payment was successful.
7a2	Payment failed page page	Provides confirmation to the user that their payment was unsuccessful.

D Mechanical part list

Table 6: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	2
DORHEA Raspberry Pi Mini Kamera	HE0304-002	2
Ultrasonic distance measuring sensor	HC-SR04	8
Micro servo motor	OKY8003	2
Red LED (3 mm)	COM-00533	6
Green LED (3 mm)	COM-09560	6
Resistors (20 kΩ)	SFR2500002002FR500	12
Jumper cables	/	≈ 40
Raspberry Pi camera extension cable	B087DFJ2RP	2
LCD screen	ST7735	1

E Backend API slugs

Table 7: Overview of all general URL slugs which are supported by the backend application.

Slug	Methods	Short Description
api/user	GET, PUT, DELETE	Get, update or delete user information.
api/user/change-password	PUT	Update a user's password, provided the old one.
api/garages	GET, POST	Get all the available garages or post a new one.
api/garage/<int:pk>	GET, PUT, DELETE	Get, update or delete information about a single garage.
api/prices/<int:pk>	GET, POST,	Get all the prices for a garage with id pk or post a new one.
api/price/<int:pk>	GET, PUT, DELETE,	Get, update or delete a price with id pk.
api/opening-hours/<int:pk>	GET, POST	Get all the opening hours for a garage with id pk or add a new one.
api/opening-hour/<int:pk>	GET, PUT, DELETE	Get, update or delete opening hours with id pk.
api/parking-lots/<int:pk>	GET, POST	Get all the parking lots for a garage with id pk or post a new one.
api/parking-lot/<int:pk>	GET, PUT, DELETE	Get update or delete a parking lot with id pk.
api/assign-parking-lot/<int:pk>	GET	Get a random free parking lot in the garage with id pk, given a start and end date.
api/garage-settings/<int:pk>	GET, PUT, DELETE	Get, update or delete the garage settings of a garage with id pk.
api/licence-plates	GET, POST	Get all licence plates for a user or add a new one.
api/licence-plate/<int:pk>	GET, PUT, DELETE	Get, update or delete a licence plate with id pk.
api/reservations	GET, POST	Get all user's reservations or add a new one.
api/reservation/<int:pk>	GET, PUT, DELETE	Get, update or delete a reservation with id pk.
api/notifications/<int:pk>	GET	Get all user's notifications.
api/notification/<int:pk>	PUT, DELETE	Update or delete a notification with id pk.

Table 8: Overview of all URL slugs for authentication which are supported by the backend application.

Slug	Methods	Short Description
`api/auth/login`	`POST`	Login a user given a email and password.
`api/auth/logout`	`POST`	Logout a user, deleting its auth token.
`api/auth/activate-account`	`GET`	Activate a user's account.
`api/auth/totp/disable`	`POST`	Disable 2FA for a user.
`api/auth/totp`	`GET`, `POST`	Get all the user's TOTP-devices or add a new one.
`api/auth/totp/<int:pk>`	`PUT`, `DELETE`	Update or delete a TOTP-device with id `pk`.
`api/auth/totp/login/<int:code>`	`POST`	Post the 2FA-code to verify the user (`code` is a six-digit number).

Table 9: Overview of all URL slugs for the local garage system, which are supported by the backend application.

Slug	Methods	Short Description
`api/rpi/images`	`POST`	Post the image taken by the Raspberry Pi to the backend for image analysis.
`api/rpi/parking-lot`	`GET`	Get the parking lots from the garage where the Raspberry Pi is installed.

Table 10: Overview of all URL slugs for payment which are supported by the backend application.

Slug	Methods	Short Description
`api/checkout/create-session`	`POST`	Create a checkout session for the user to pay.
`api/checkout/preview`	`GET`	Get the items which the user has to pay (i.e. time quantities parked).
`api/checkout/webhook`	`POST`	Listen to incoming requests from the Stripe servers for checkout updates.
`api/stripe-connection`	`POST`, `DELETE`	Add or remove customers from Stripe.
`api/invoice/webhook`	`GET`	Listen to invoice updates from the Stripe servers.

F Sequence diagrams

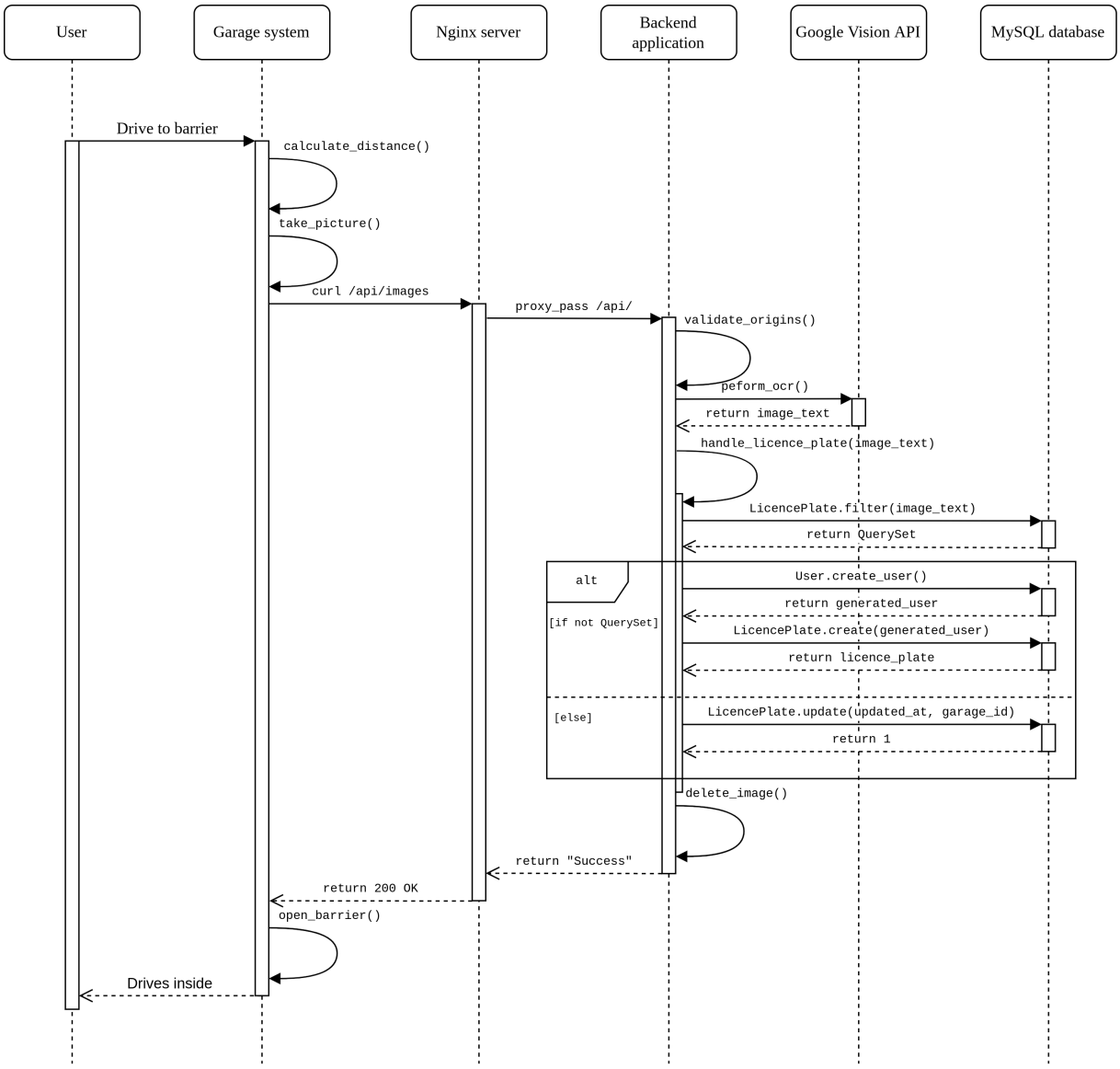

Figure 19: Sequence diagram of the licence plate registration in the local garage system.

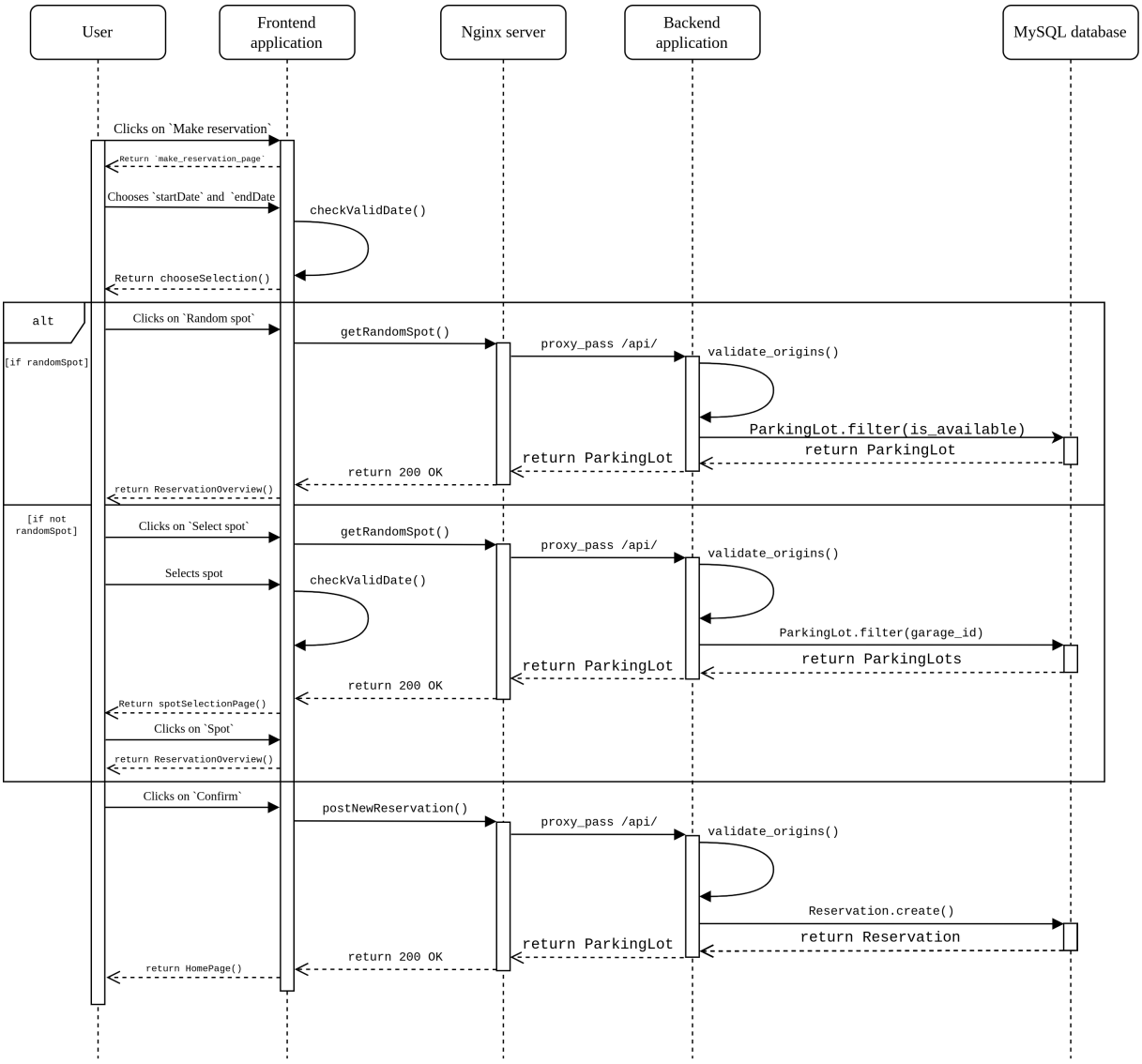

Figure 20: Sequence diagram of the reservation flow.

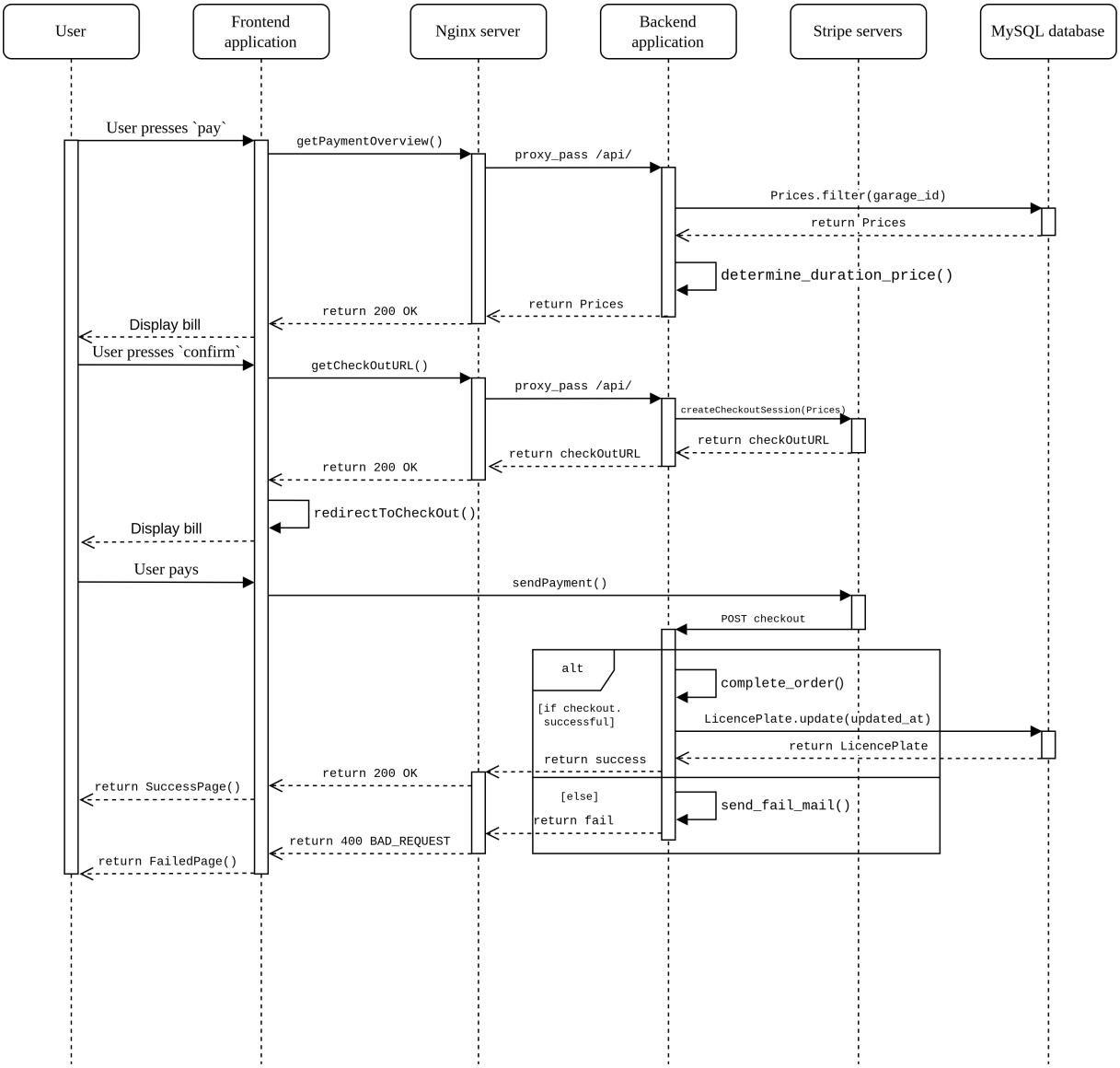

Figure 21: Sequence diagram of a manual payment. Note that this represents a payment without errors.

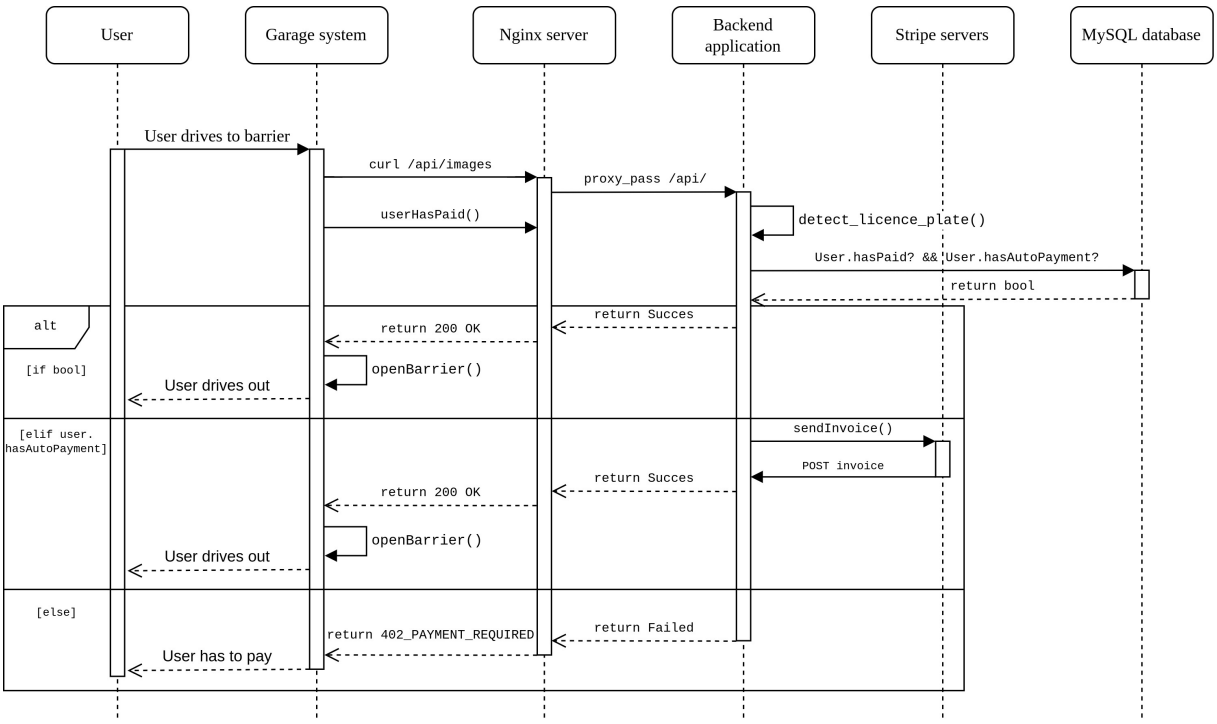

Figure 22: Sequence diagram of a automatic payment. Note that this represents a payment without errors. The `detect-licence-plate()`-function is implemented with the sequence diagram of Figure 19.

G Flowcharts

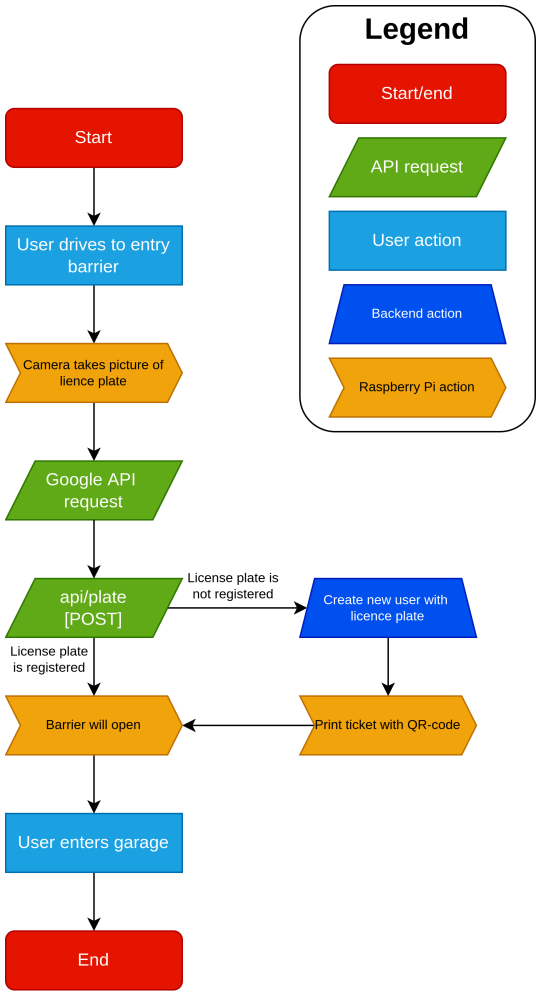

Figure 23: Flowchart of the entering process of the garage in both hardware, software and user terms.

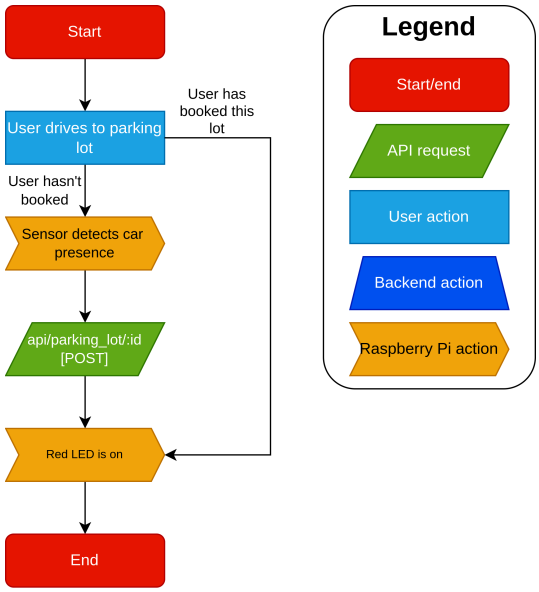

Figure 24: Flowchart of the car detection process of the garage in both hardware, software and user terms.

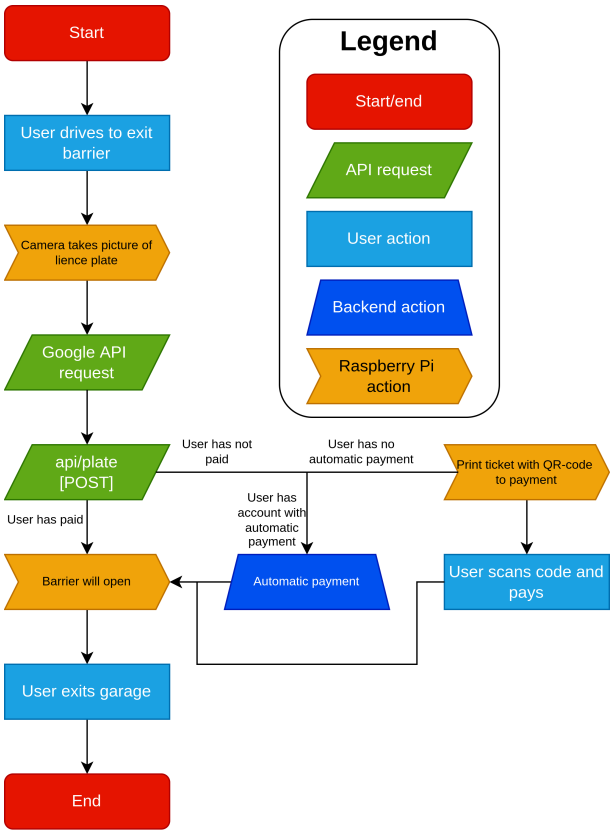

Figure 25: Flowchart of the exiting process of the garage in both hardware, software and user terms.