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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

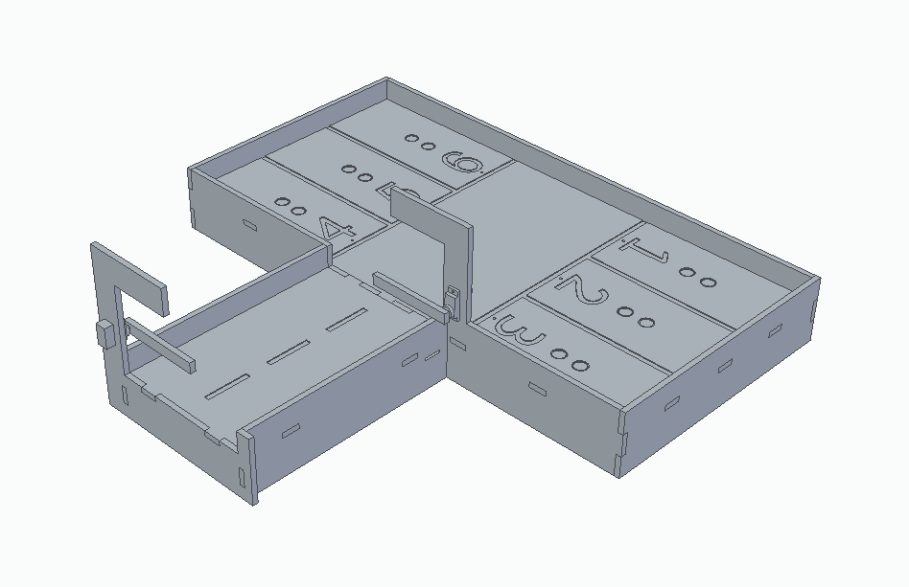

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfill: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signaling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to gray-scale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

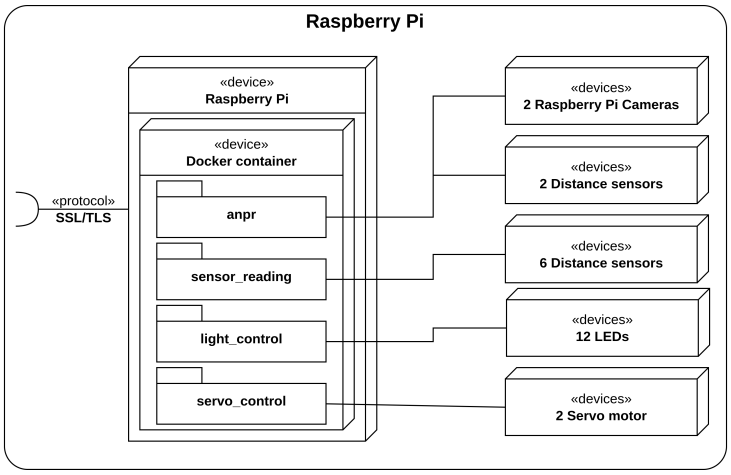

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] `https://flutter.dev/`
[8] `https://reactjs.org/`
[9] `https://angularjs.org/`

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to our backend API.

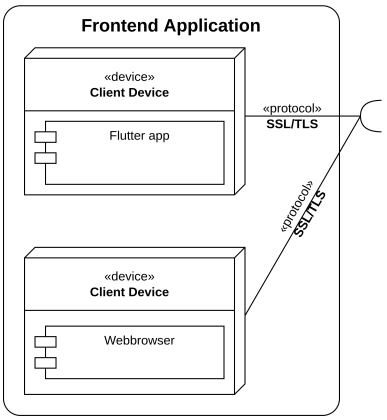

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

Firstly, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

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

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as: Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

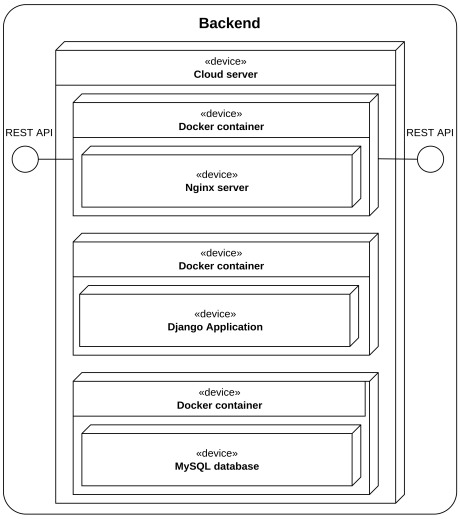

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote cloud. The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. 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 4 shows the deployment diagram of the entire backend software.

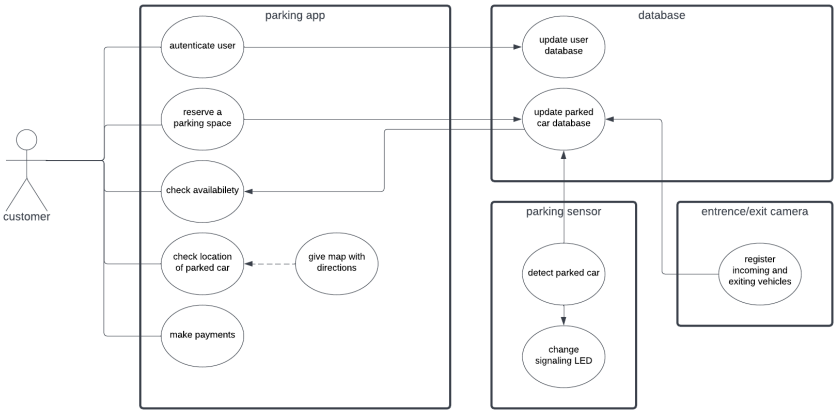

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend's perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

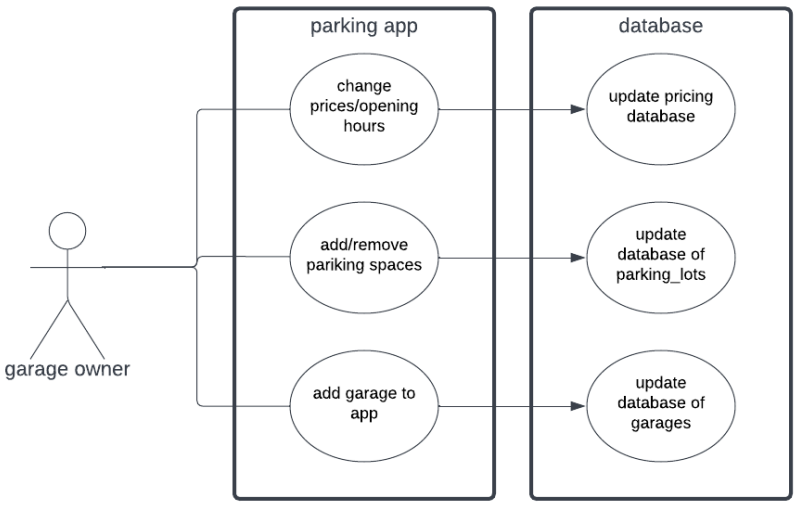

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	**Item**
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). `django-admin` and `manage.py`. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). `django-rest-knox`. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

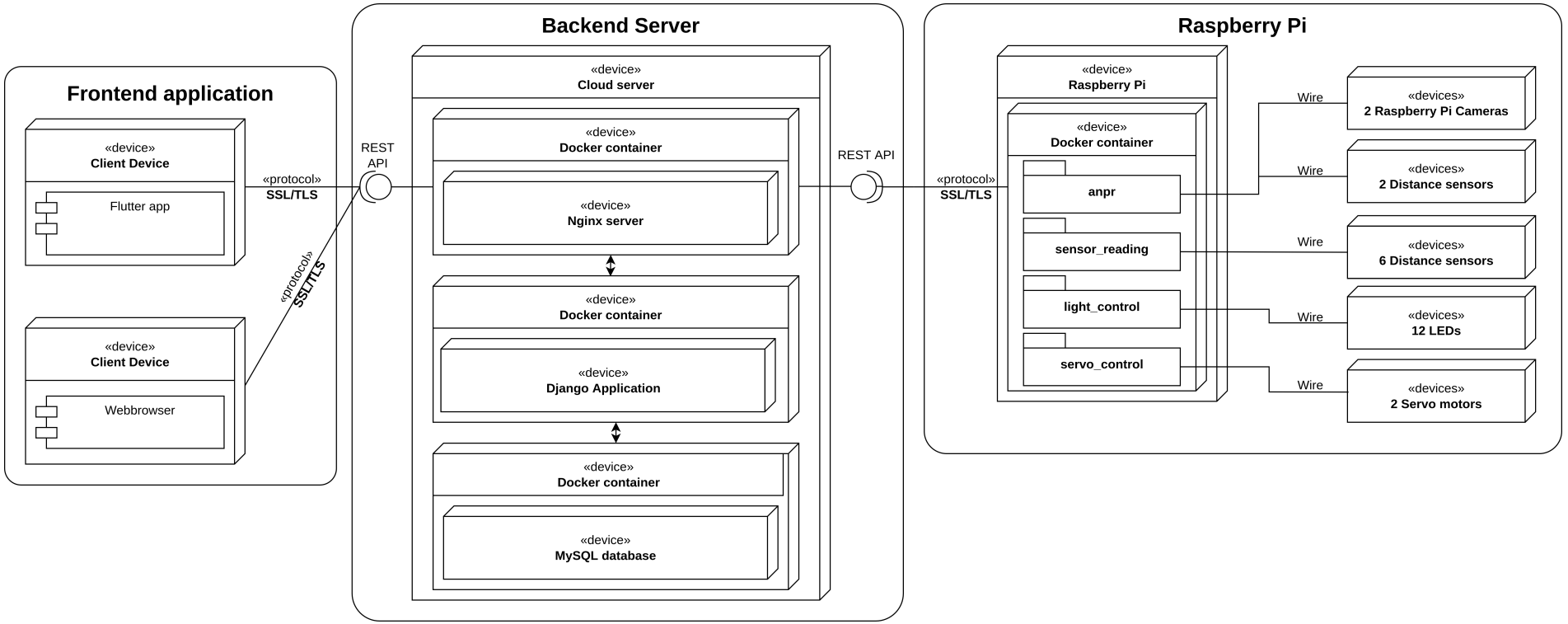

Figure 7: General deployment diagram.

B Application design

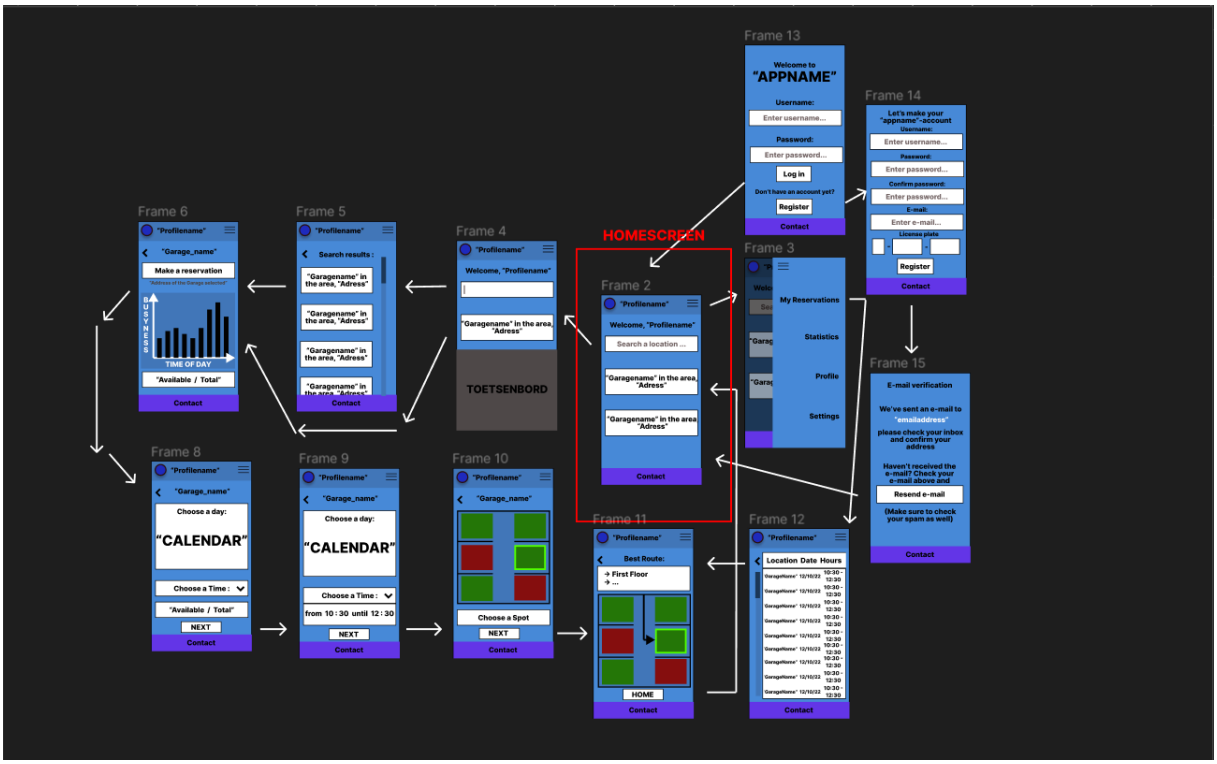

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

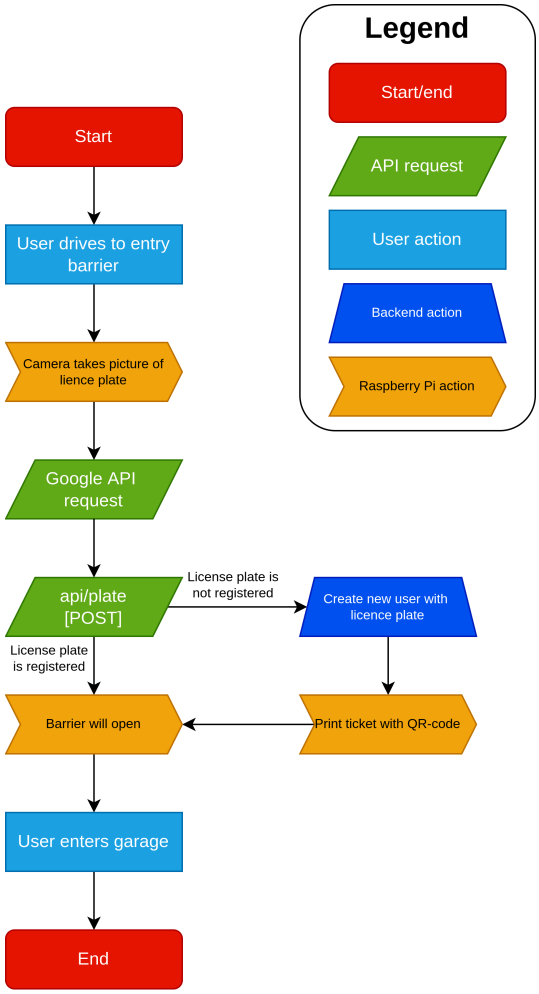

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

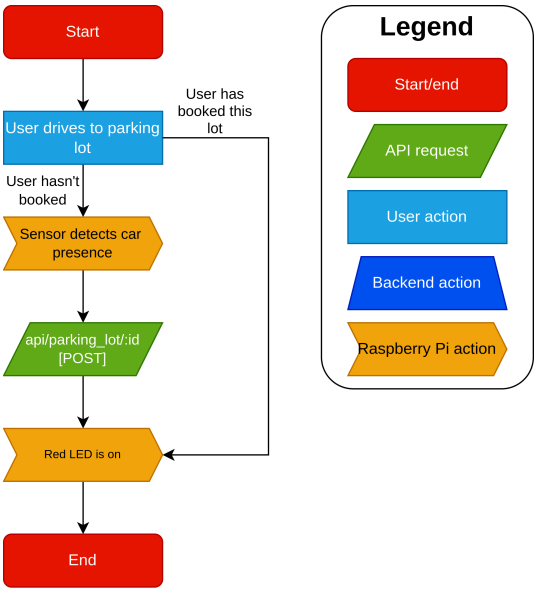

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

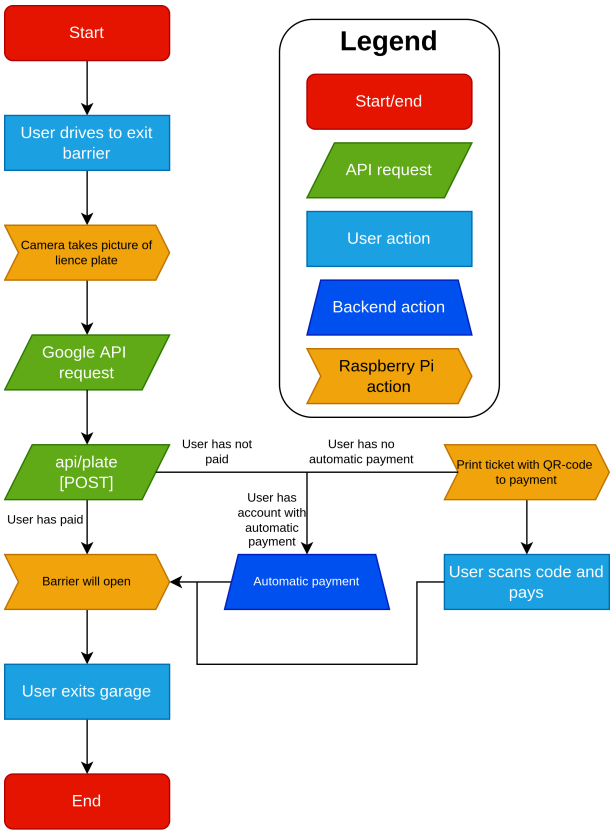

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.

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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

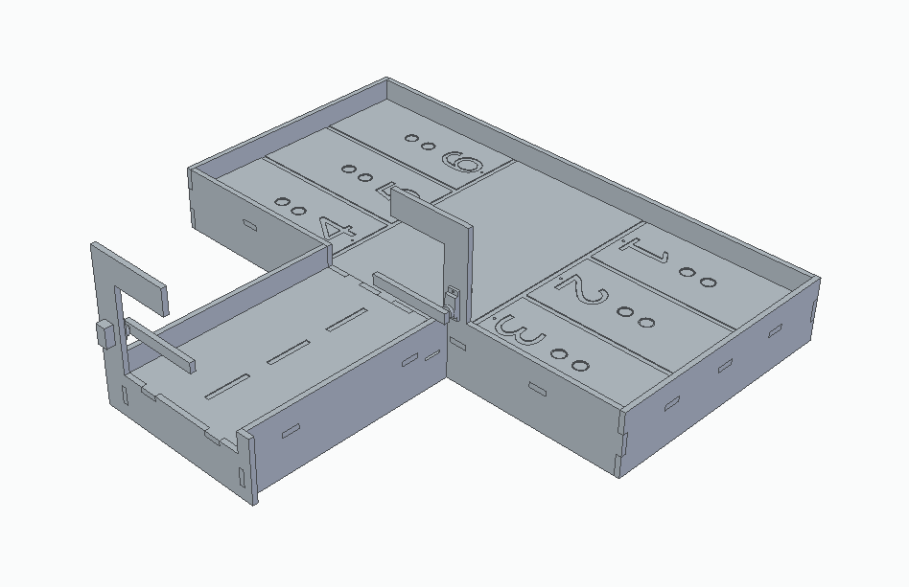

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfil: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signalling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to grayscale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

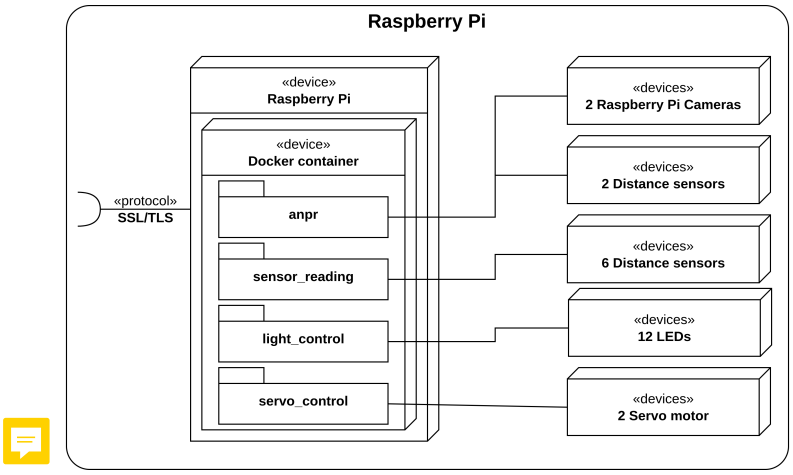

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] https://flutter.dev/
[8] https://reactjs.org/
[9] https://angularjs.org/

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to backend API.

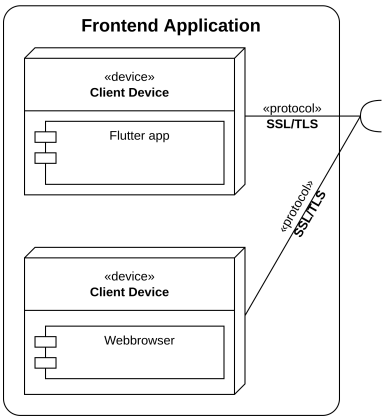

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

Firstly, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

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

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as: Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

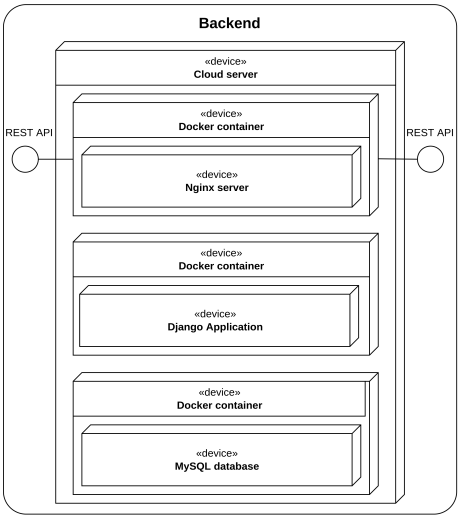

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote cloud. The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. 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 4 shows the deployment diagram of the entire backend software.

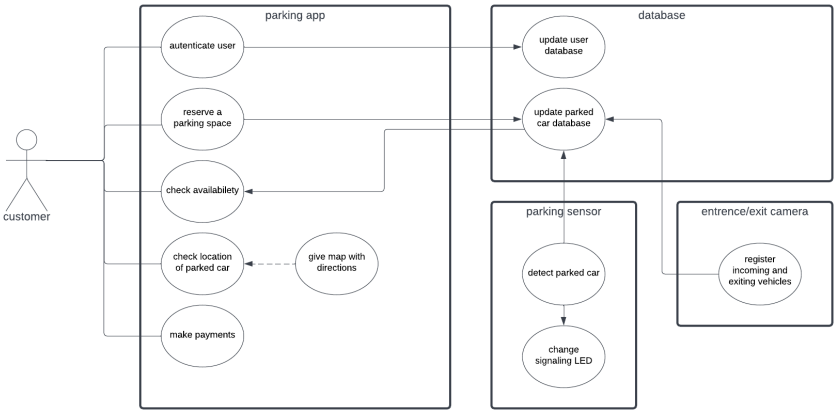

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend's perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

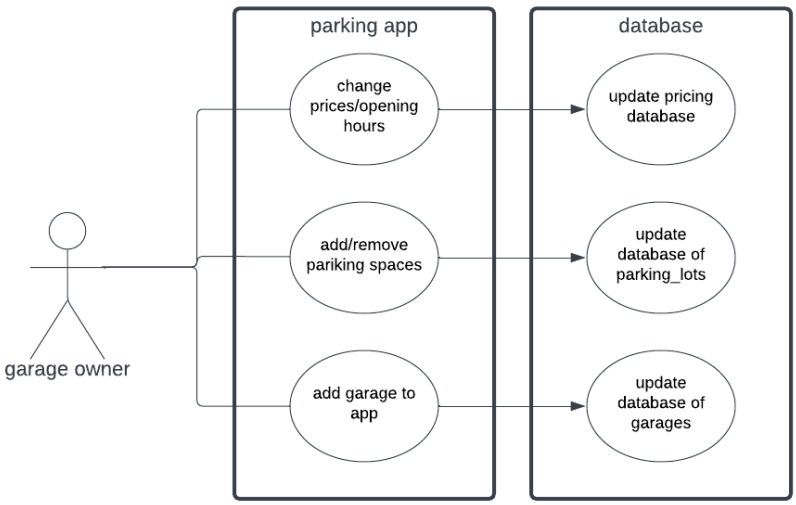

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	**Item**
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). `django-admin` and `manage.py`. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). `django-rest-knox`. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

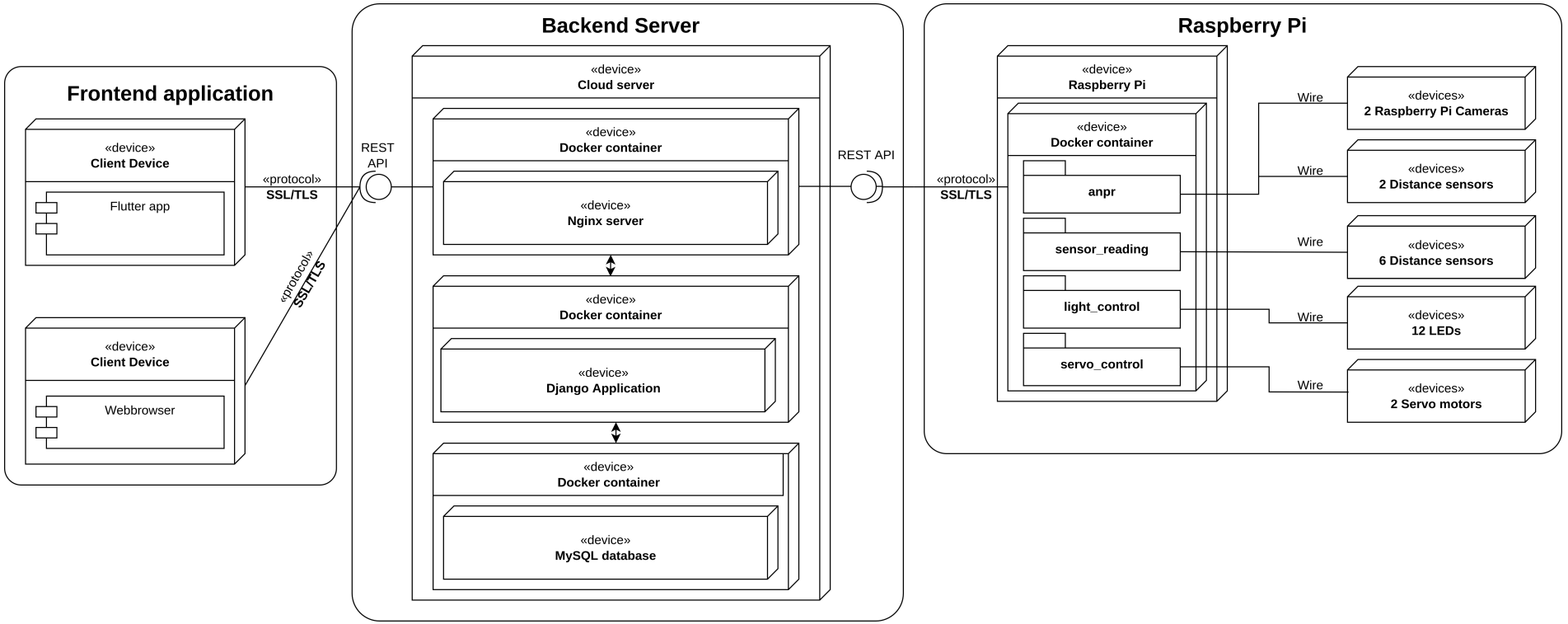

Figure 7: General deployment diagram.

B Application design

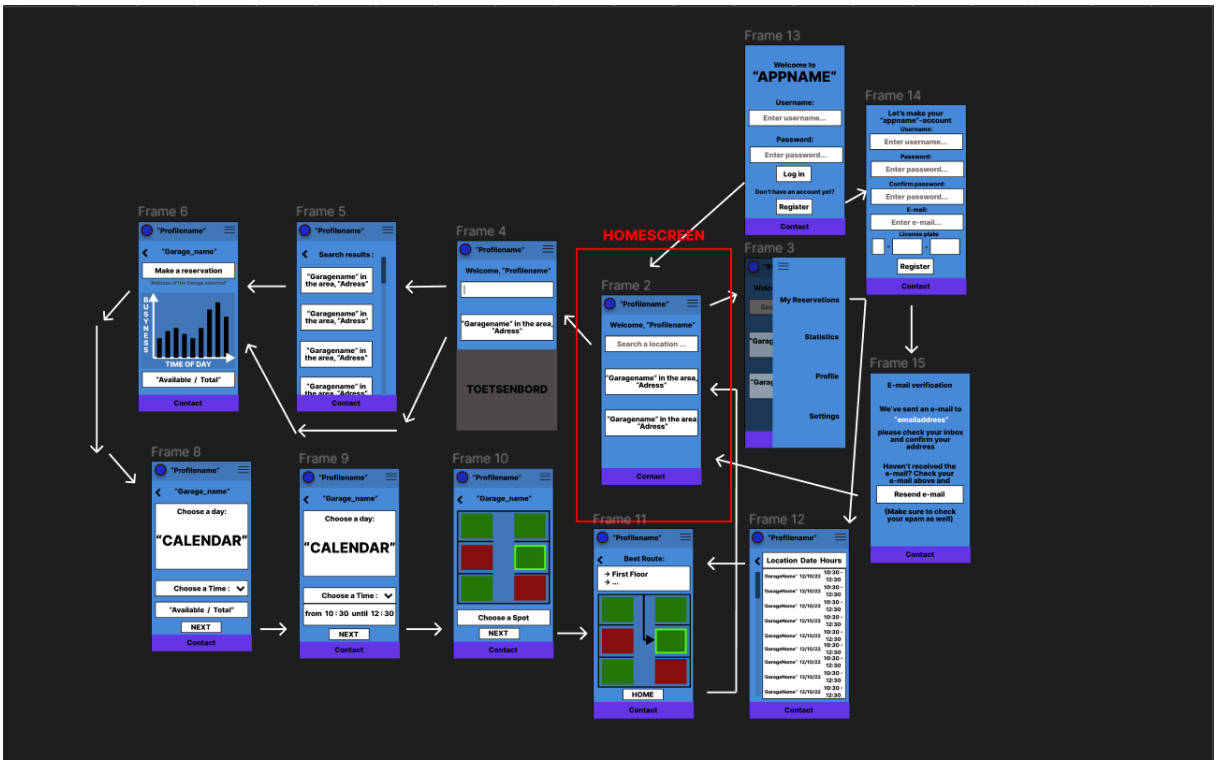

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

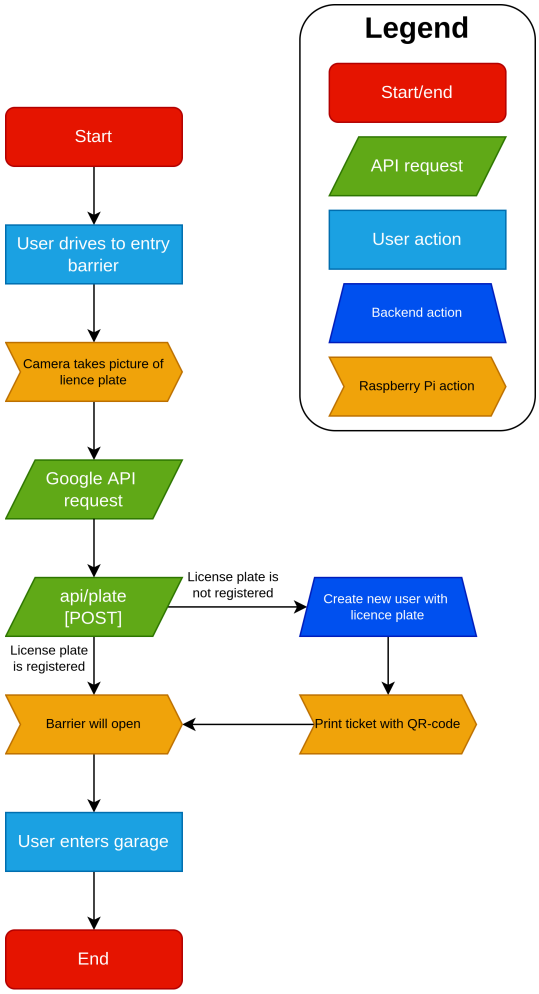

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

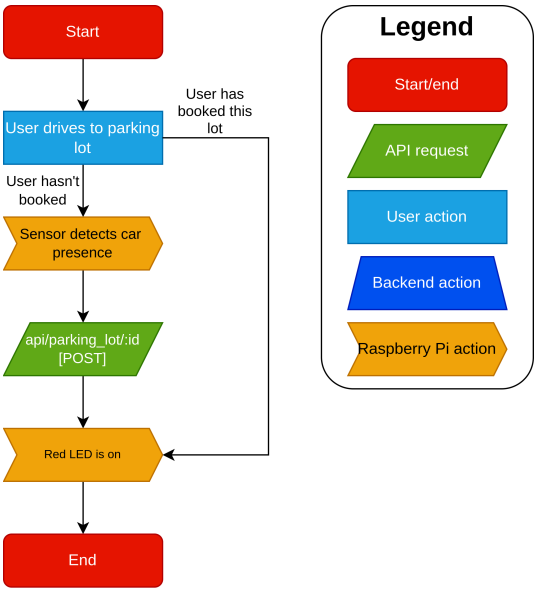

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

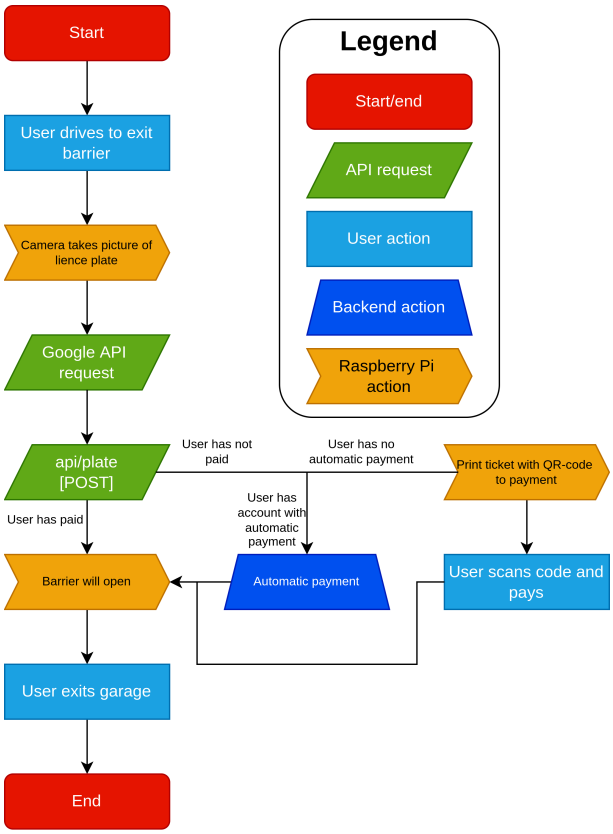

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.

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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

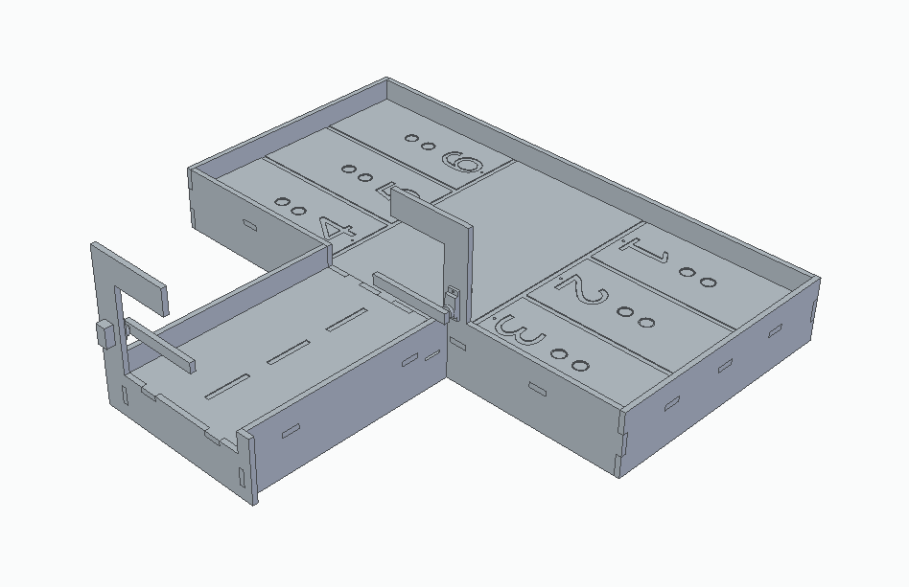

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfill: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signaling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to gray-scale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

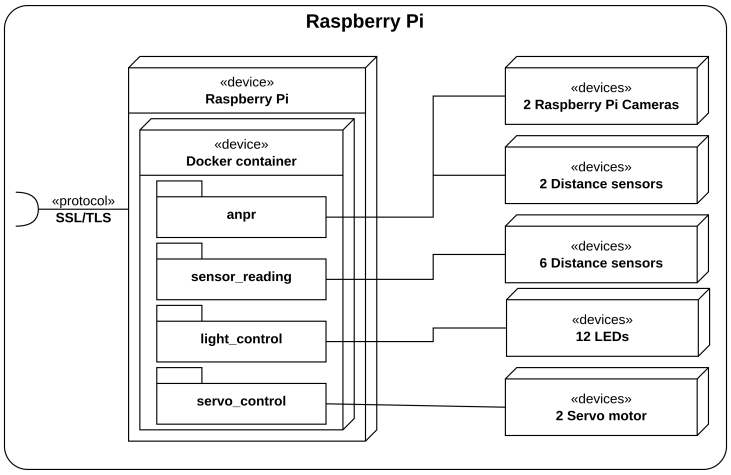

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] `https://flutter.dev/`
[8] `https://reactjs.org/`
[9] `https://angularjs.org/`

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to our backend API.

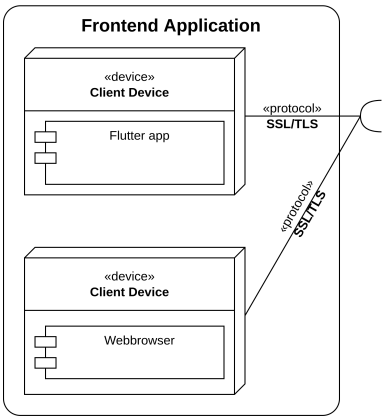

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

First, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

[10] https://nginx.org/

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

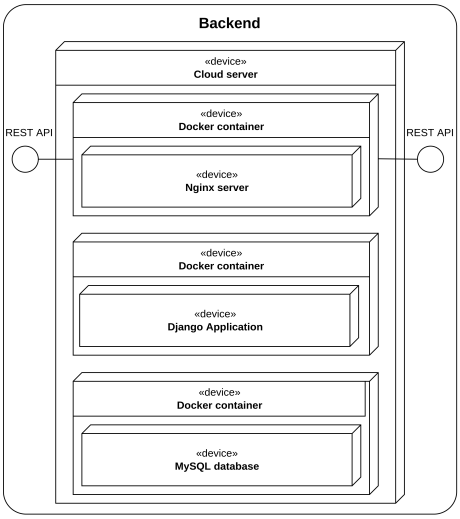

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote clo The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. In total, there are three containers for the Nginx server, one for the Gunicorn gateway which runs the Django application and one for th ySQL database. The containers run as a single service with Docker Compose, which makes communication between the different containers effortless [Docker, 2022]. Figure 4 shows the deployment diagram of the entire backend software.

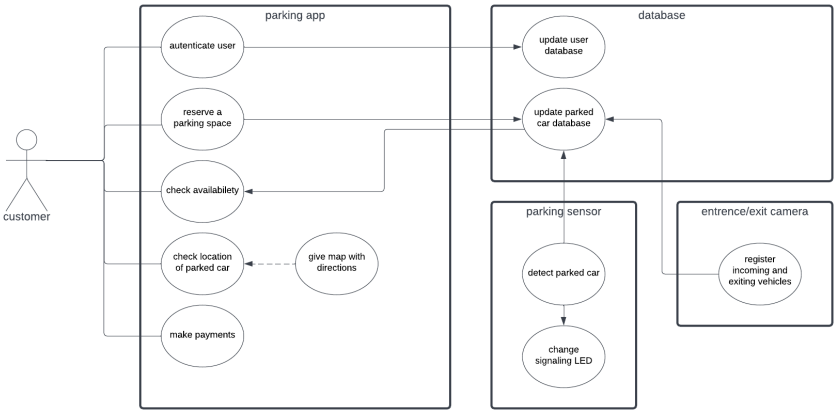

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend's perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

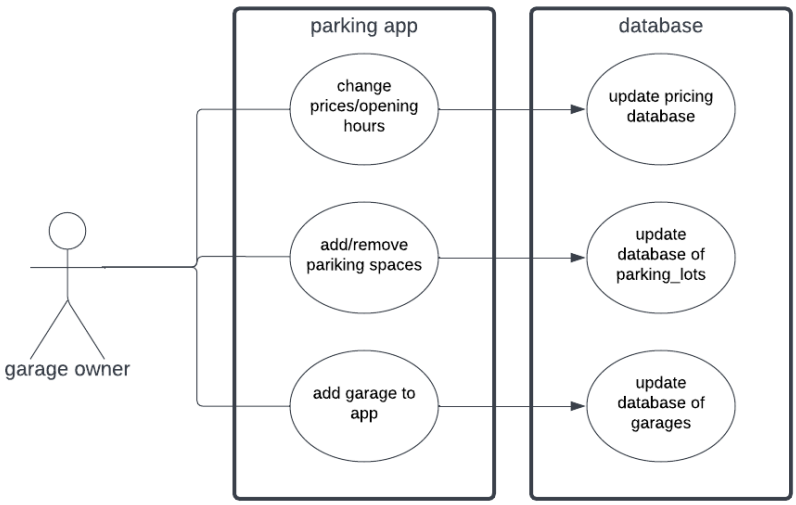

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	**Item**
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). `django-admin` and `manage.py`. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). `django-rest-knox`. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

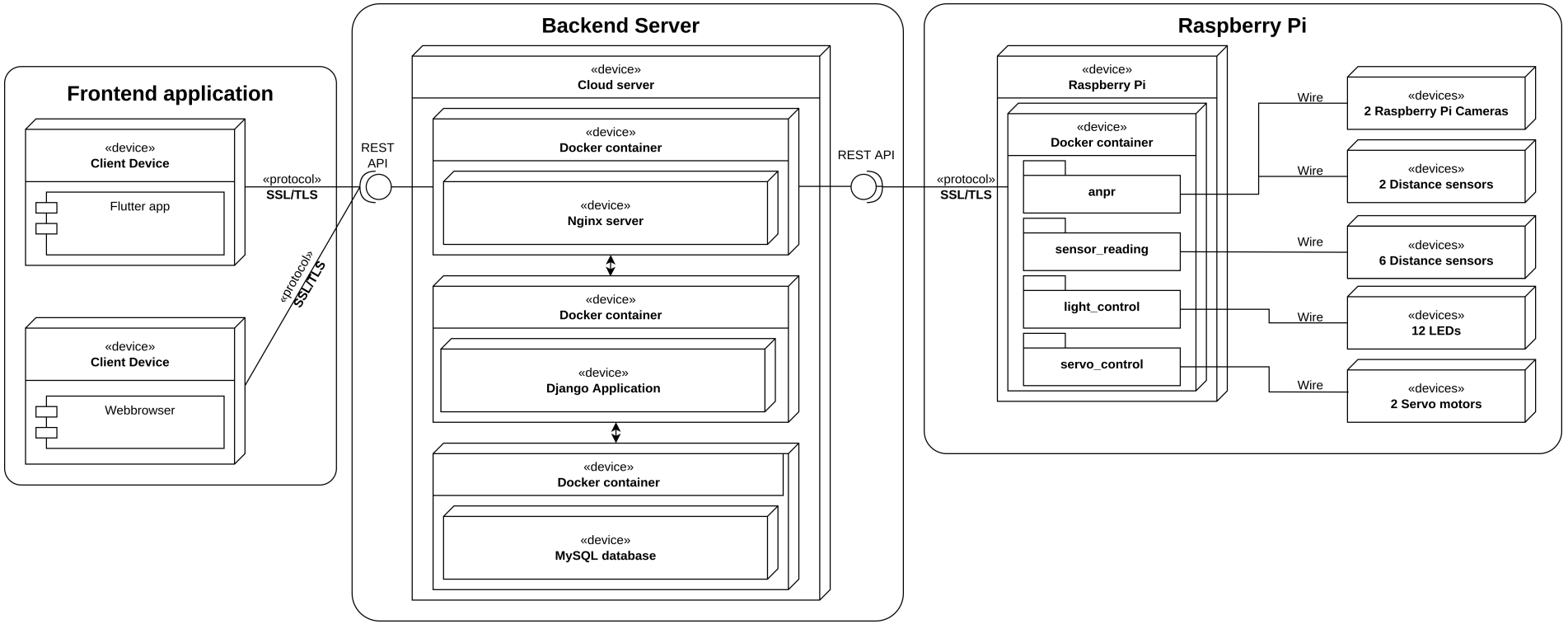

Figure 7: General deployment diagram.

B Application design

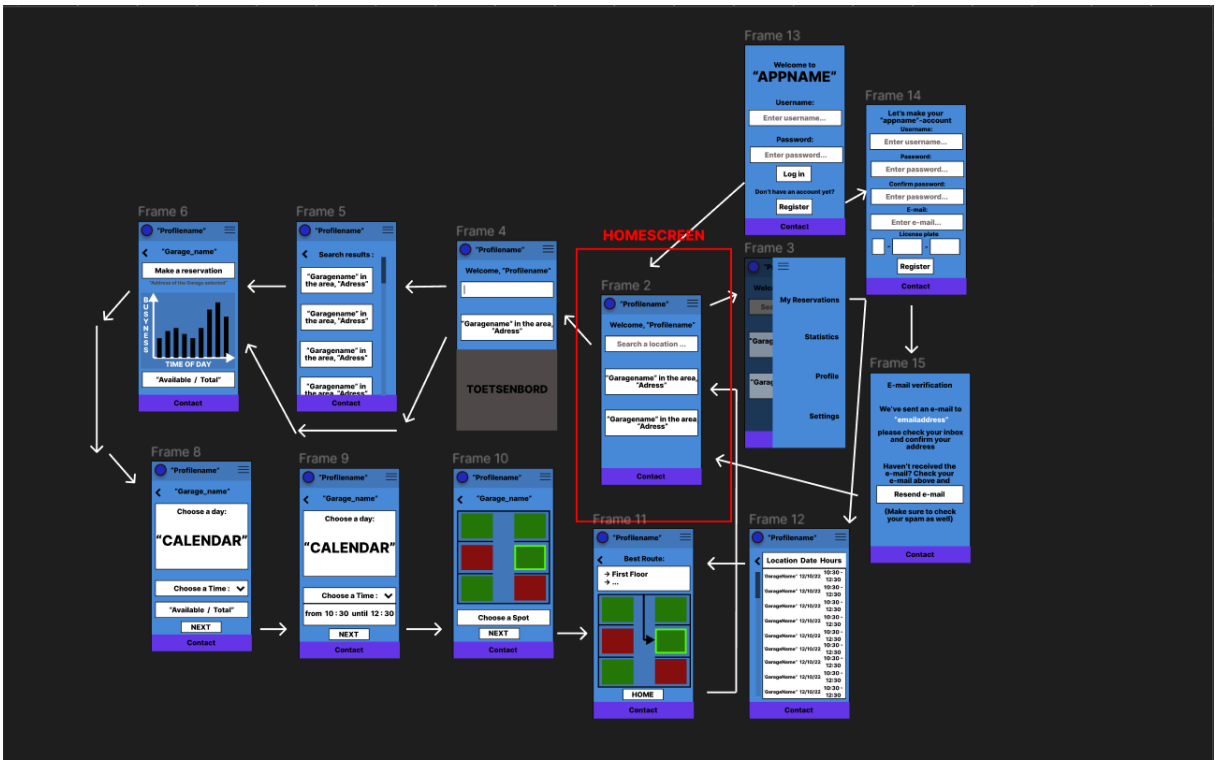

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

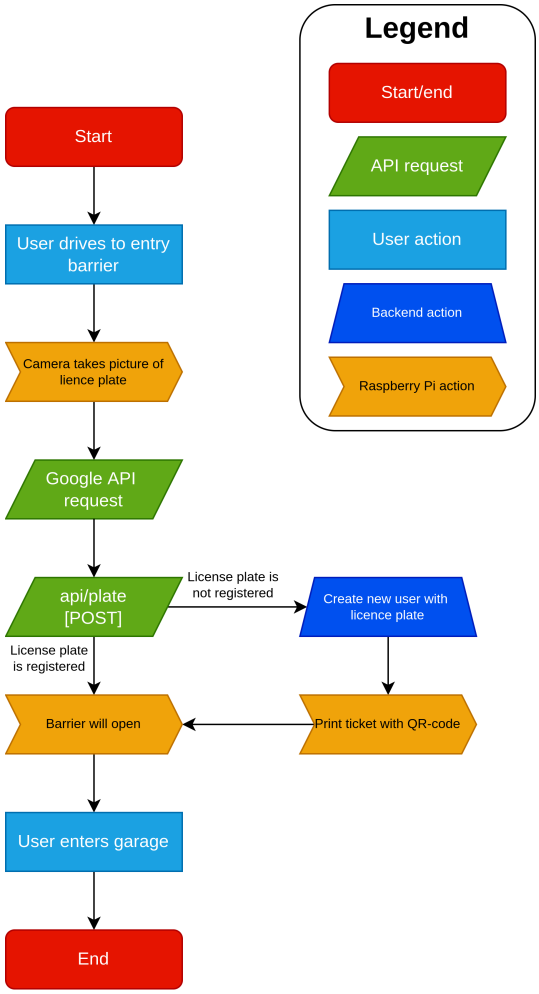

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

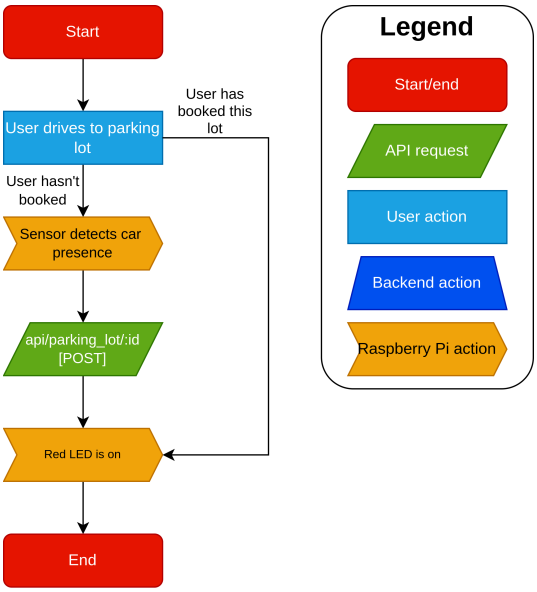

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

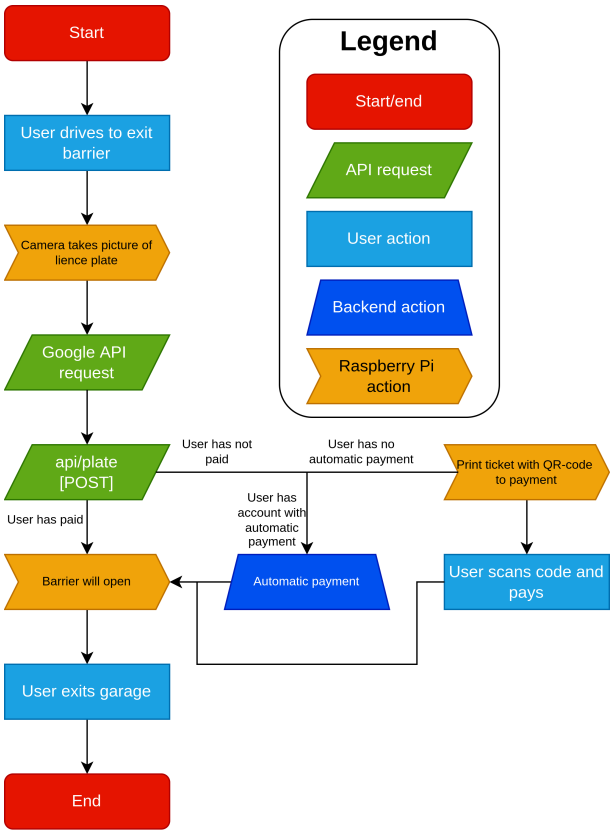

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.

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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

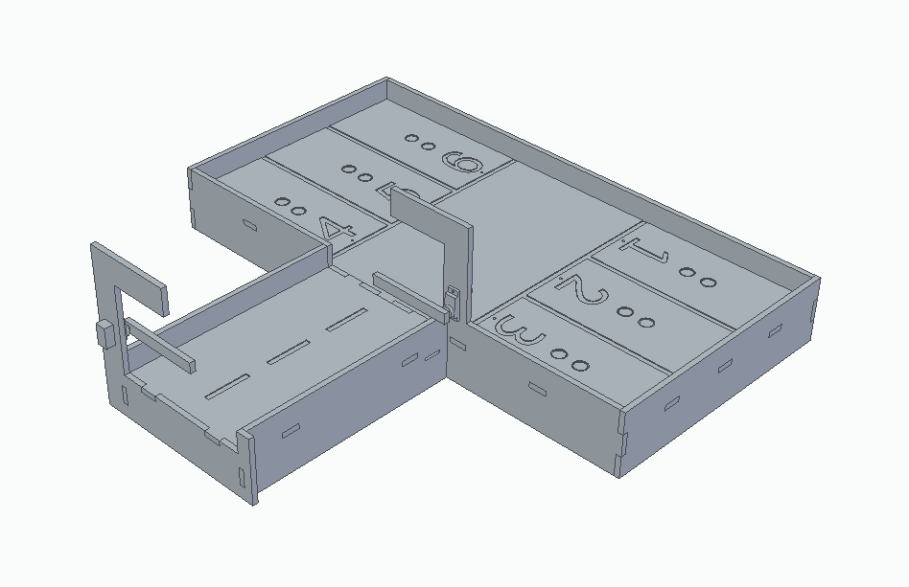

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfill: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signaling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to gray-scale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

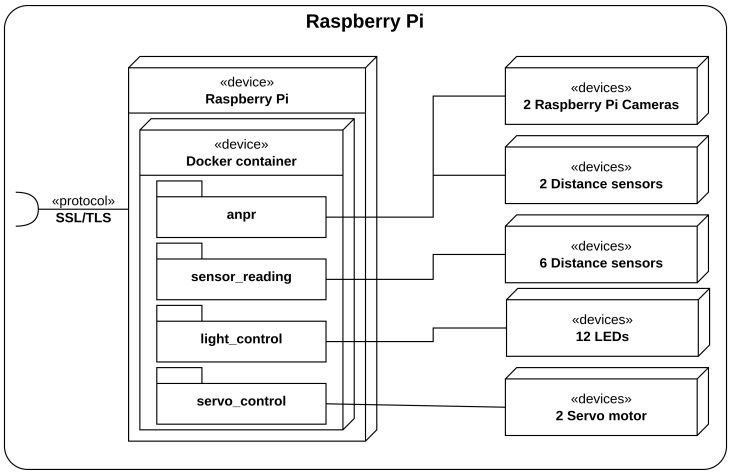

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] `https://flutter.dev/`
[8] `https://reactjs.org/`
[9] `https://angularjs.org/`

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to our backend API.

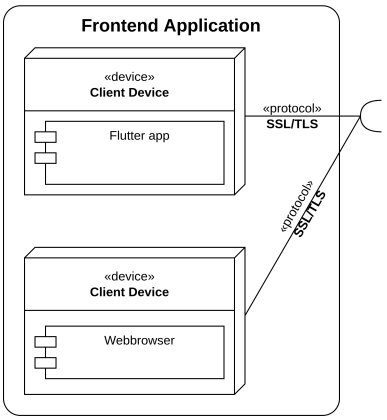

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

Firstly, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

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

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as: Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

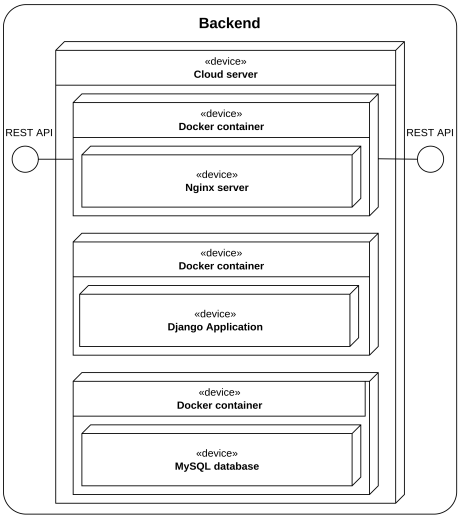

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote cloud. The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. 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 4 shows the deployment diagram of the entire backend software.

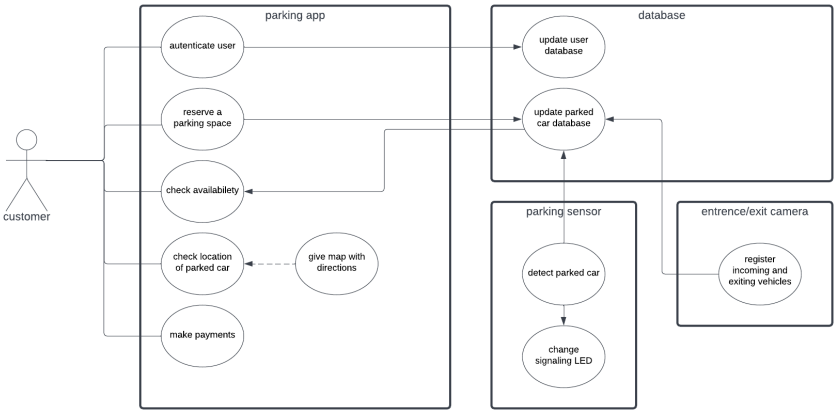

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

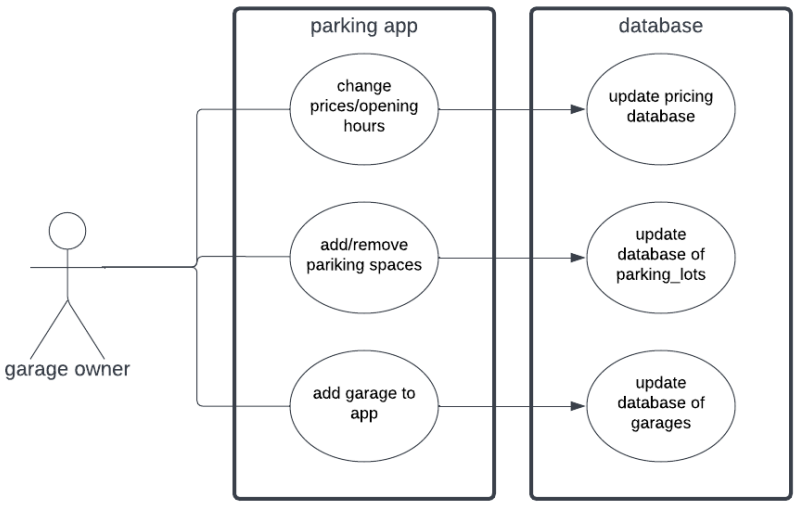

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	**Item**
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). `django-admin` and `manage.py`. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). `django-rest-knox`. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

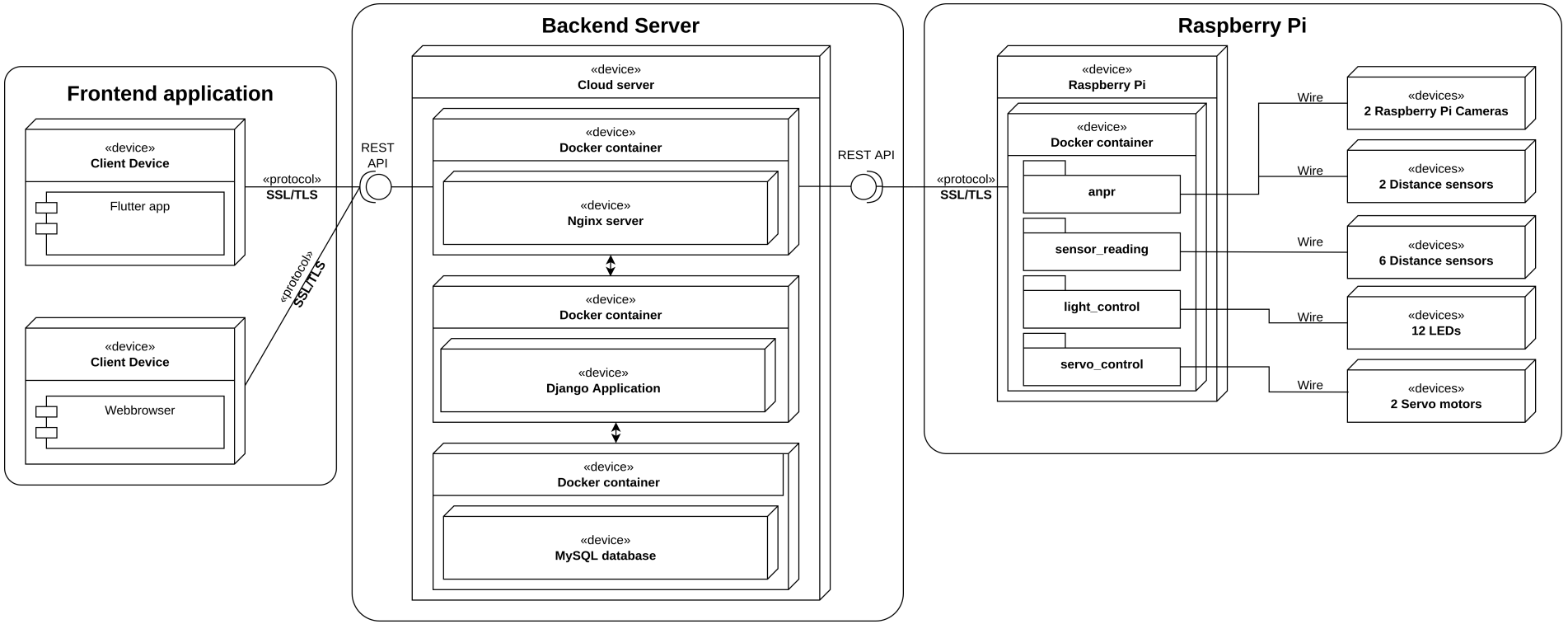

Figure 7: General deployment diagram.

B Application design

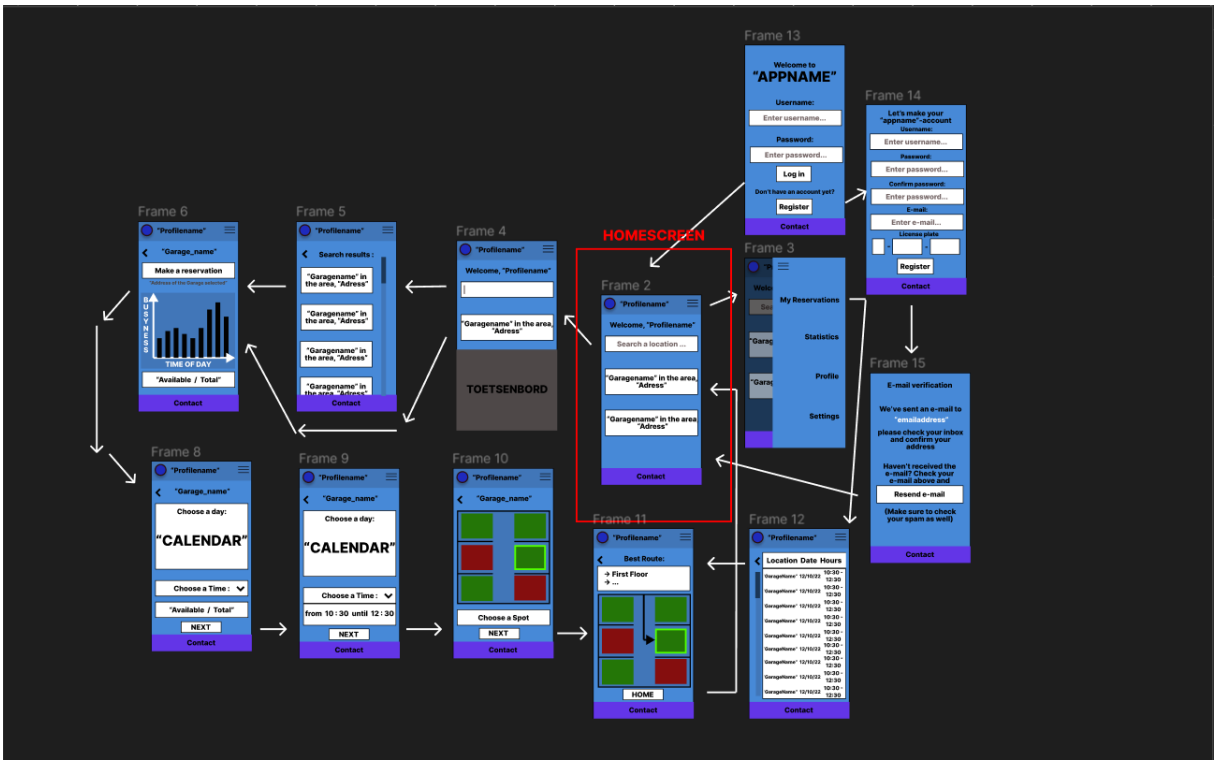

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

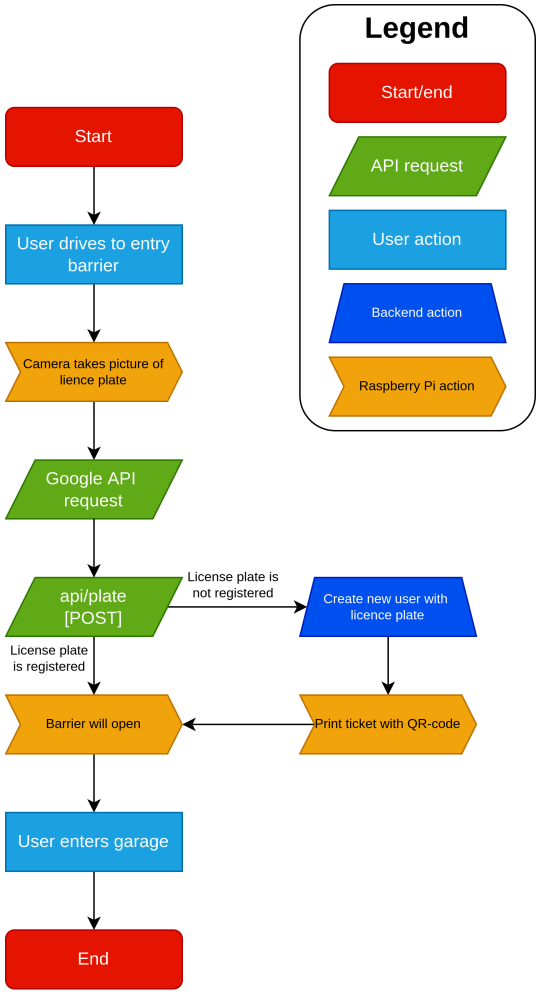

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

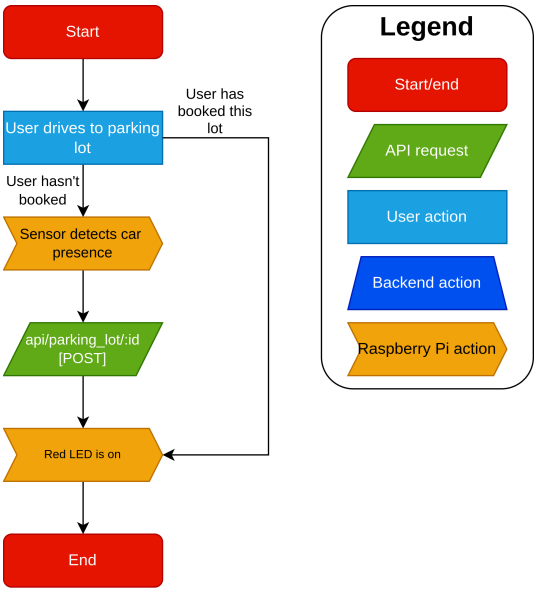

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

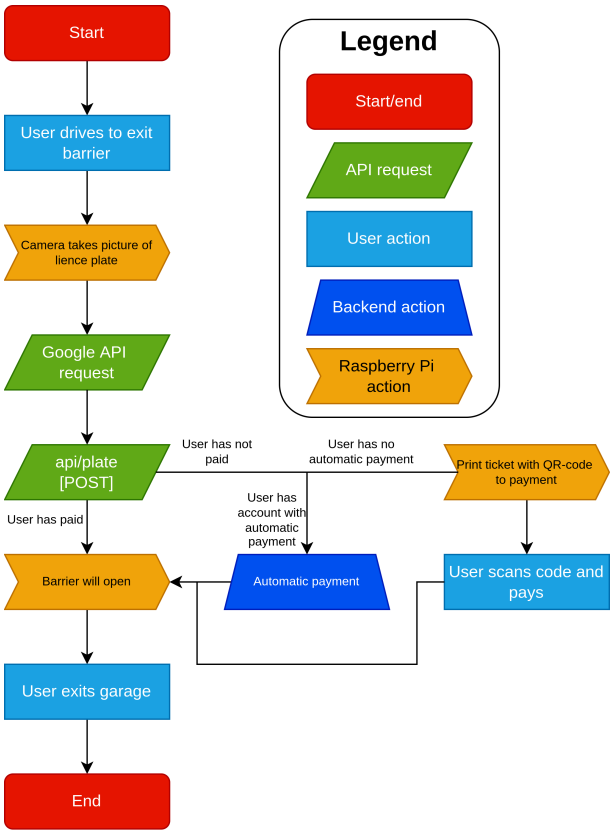

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.

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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

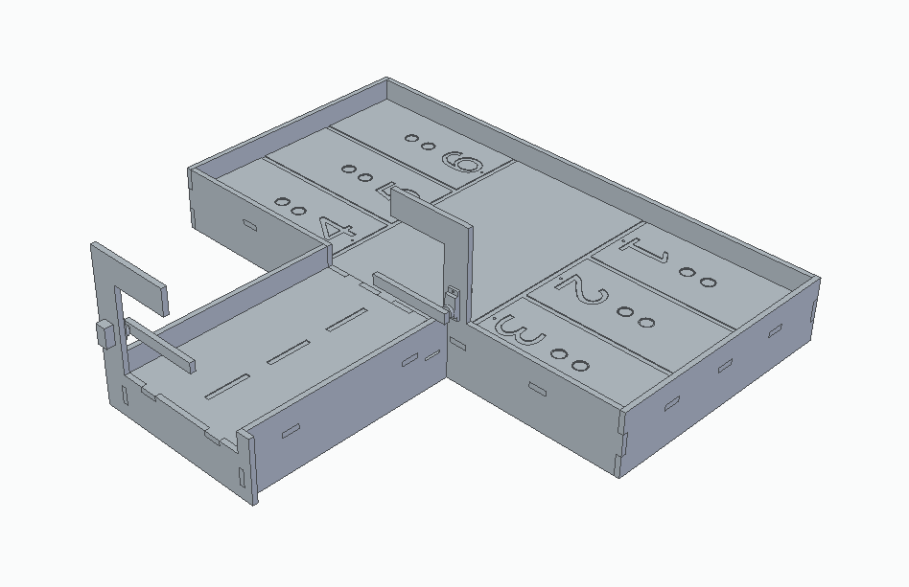

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfill: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signaling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to gray-scale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

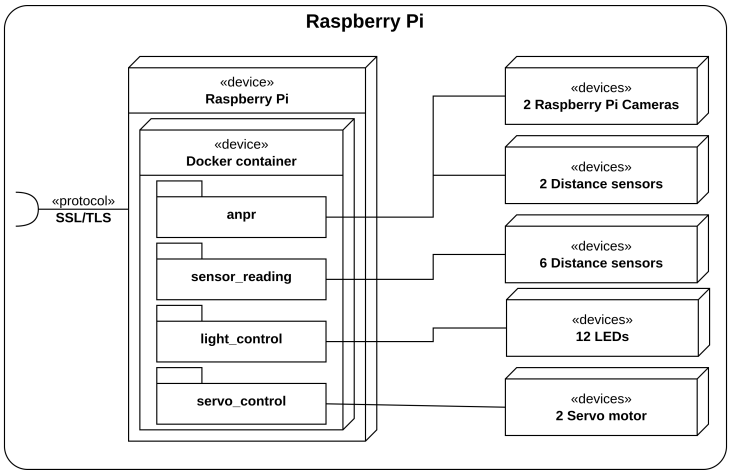

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] `https://flutter.dev/`
[8] `https://reactjs.org/`
[9] `https://angularjs.org/`

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to our backend API.

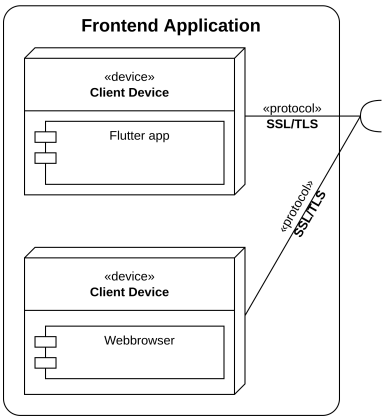

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

Firstly, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

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

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as: Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

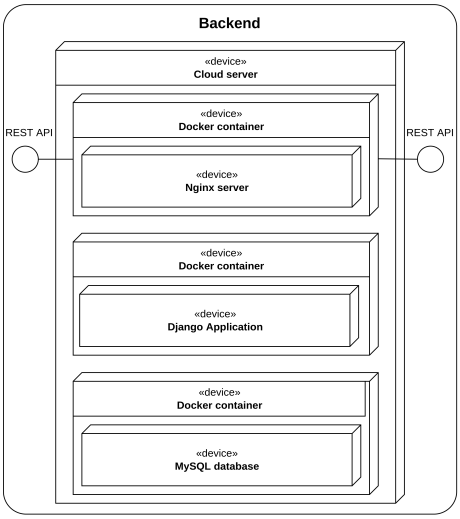

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote cloud. The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. 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 4 shows the deployment diagram of the entire backend software.

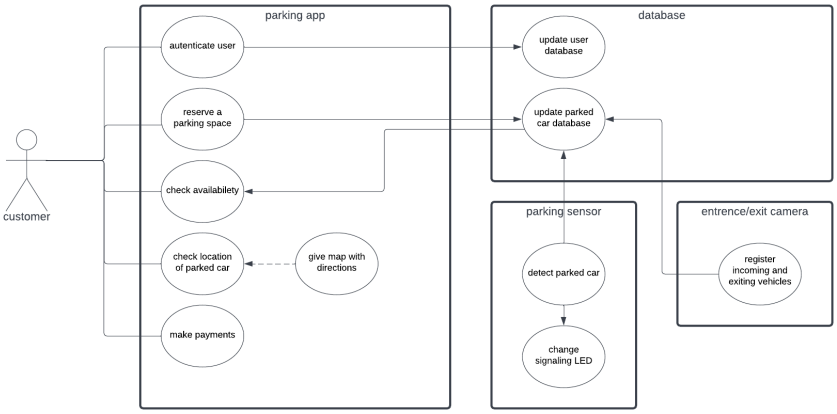

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend's perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

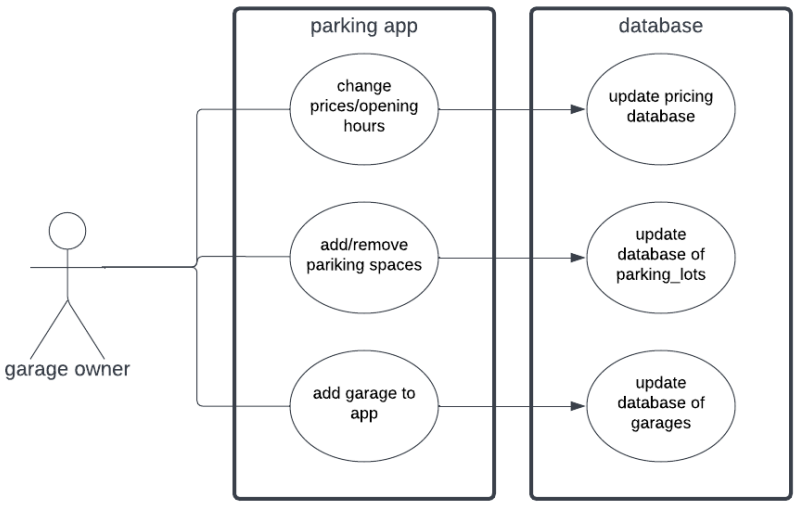

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	Item
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). django-admin and manage.py. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). django-rest-knox. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

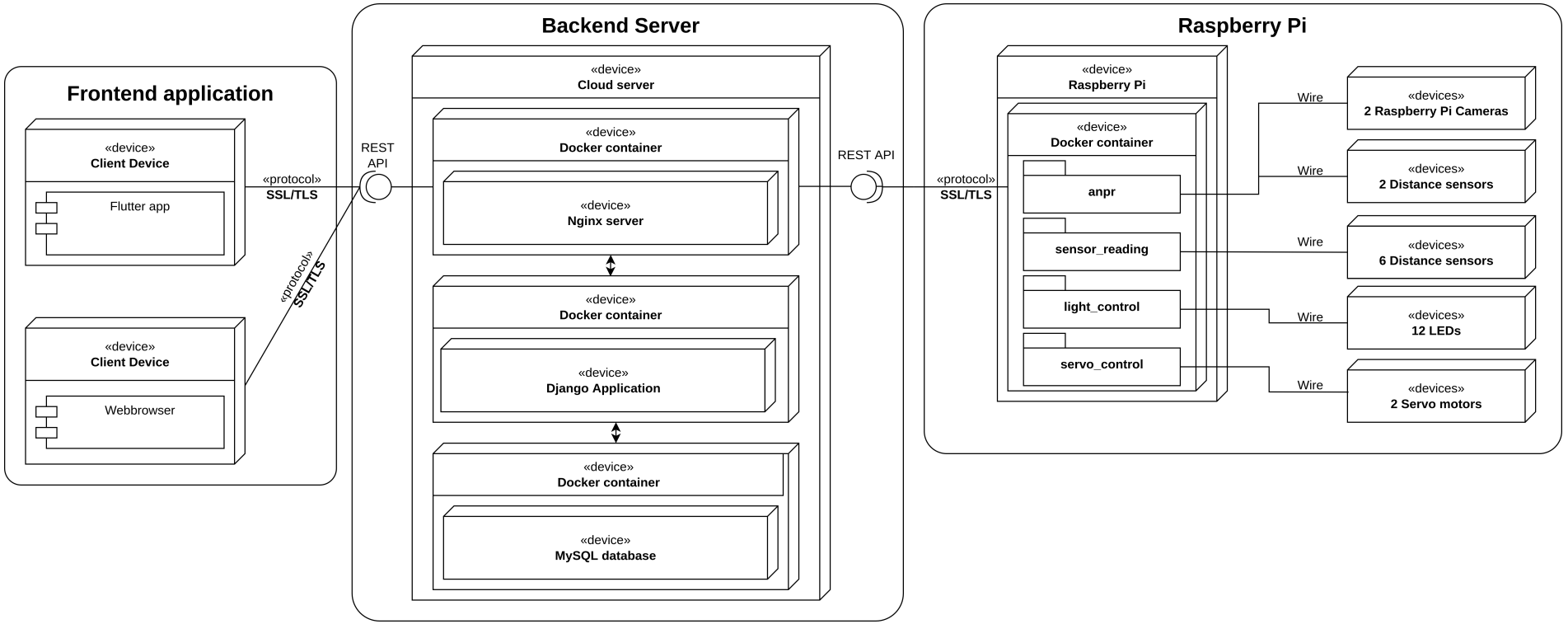

Figure 7: General deployment diagram.

B Application design

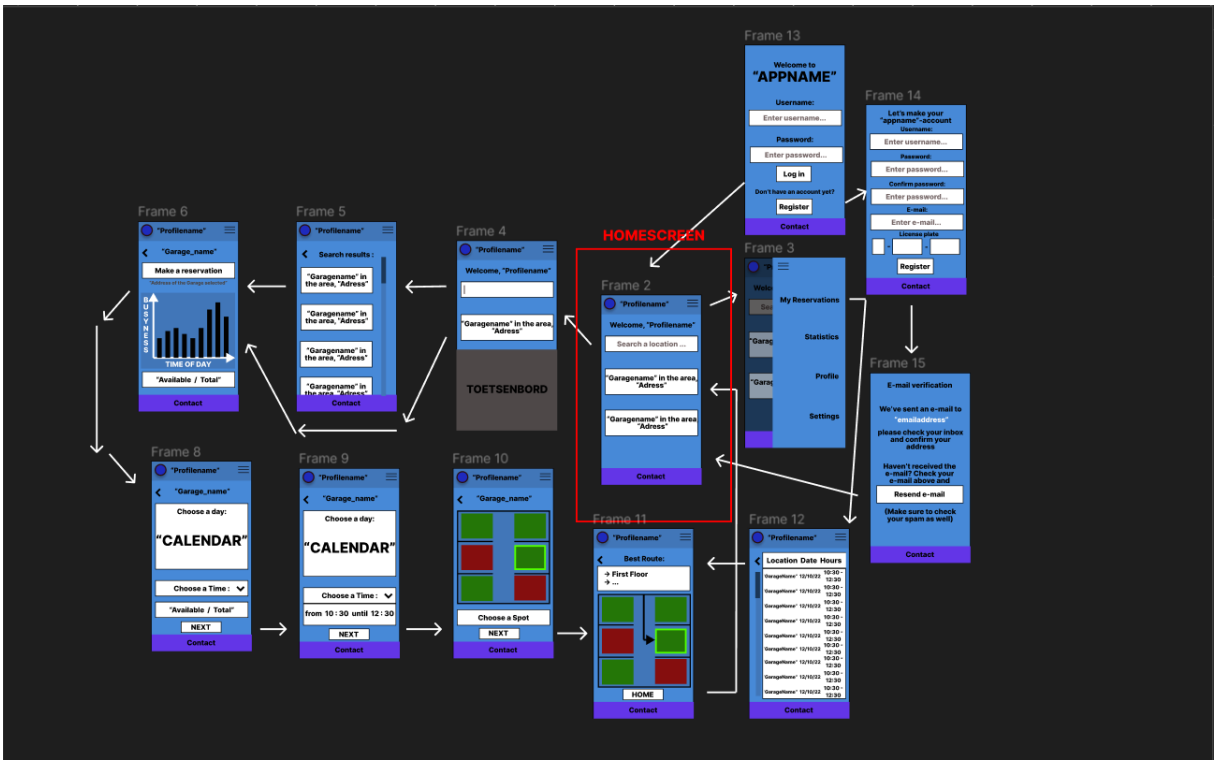

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

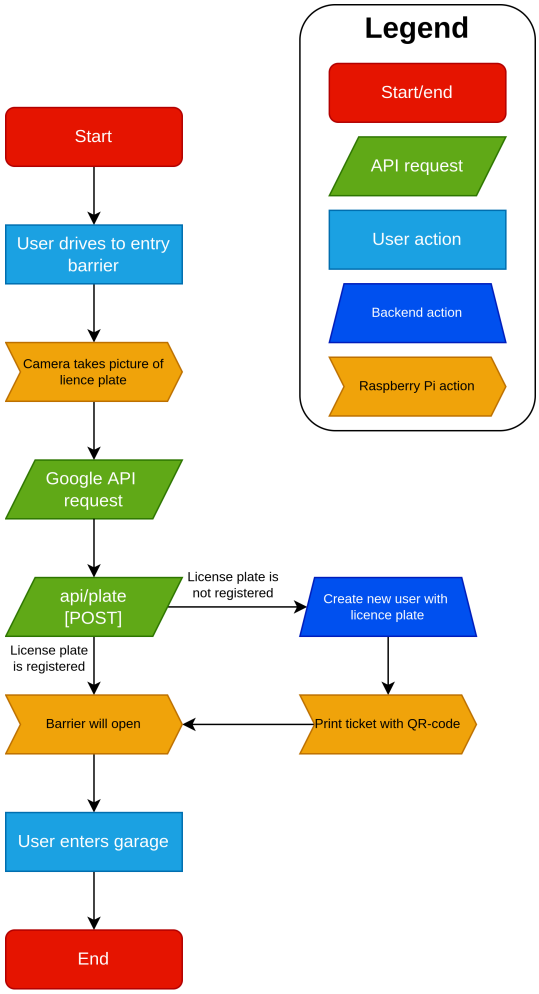

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

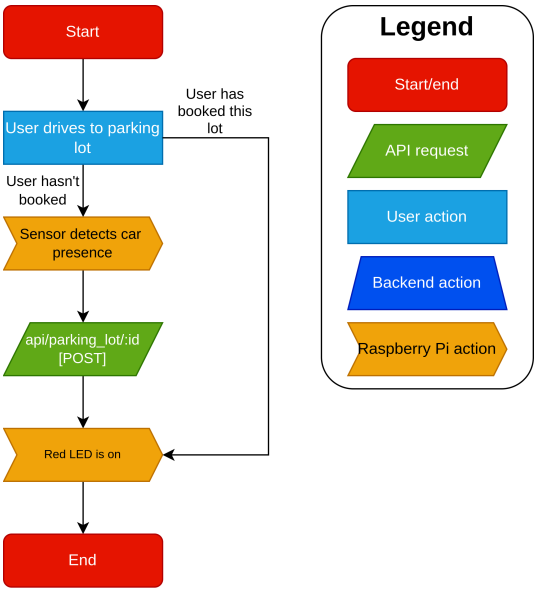

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

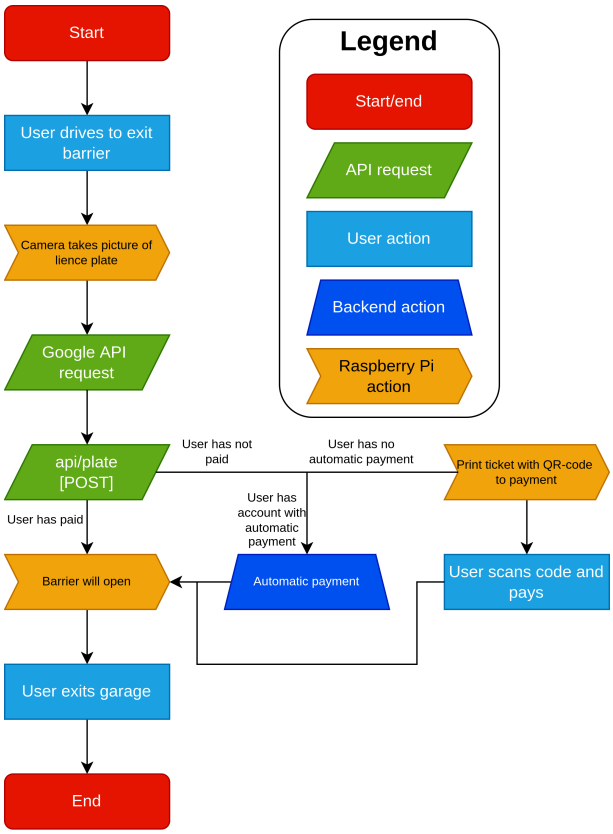

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.

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

PRELIMINARY 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.

Contents

List of Figures

List of Tables

List of Abbreviations

ANPR	Automatic Number Plate Recognition
API	Application Programming Interface
CPU	Central Processing Unit
GDPR	General Data Protection Regulation
HTML	HyperText Markup Language
HTTPS	HyperText Transport Protocol Secure
HTTP	HyperText Transport Protocol
IOT	Internet of Things
JSON	JavaScript Object Notation
MVC	Model-View-Controller
OCR	Optical Character Recognition
ORM	Object Relational Mapper
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
UDMS	Ultrasonic Distance Measuring Sensor
VPN	Virtual Private Network
WSGI	Web Server Gateway Interface
JS	JavaScript

1 Introduction

This intermediate report describes our current progress towards designing an intelligent parking garage. The 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 our garage and park his/her car here 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. The parking garage has an application via which clients will be able to check how many available spaces there are and reserve a space if they want to. The app is equipped with a manual pay-function or users can enter their banking-details and the payments will happen automatically when they leave the garage. This garage must be build within a budget of 250 euros.
This intermediate report will describe both the mechanical and software aspects of the current state of the design. The section about the software will be spilt in the three main parts of the system: the Raspberry Pi, the frontend application and the backend server. Then, the user experience is described in the section 4. Consequently, an overview of the planning of the upcoming weeks is given, together with an overview of the budget. The last section describes the course integration of the project with the different courses of 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 Mechanical design description

This section describes the mechanical design of the physical parking garage scale model which has to be built for the final demonstration. First, the current design will be explained, after which the possible alternatives and motivations for this design are illustrated.

2.1 Final parking garage design

The demonstration will be given with a scale model of the parking garage. Figure 1 shows a 3D-model of this garage.[1] This model has 6 working parking spaces. The entry and exit of the garage are blocked by a barrier which are controlled by a micro servo motor (OKY8003).[2] Both the entrance and exit are equipped with an Ultrasonic Distance Measuring Sensor (UDMS) to detect when a car tries to enter or exit the garage. The entry barrier will open when the licence plate is detected by the camera above the barrier. The exit barrier will open when the client has paid and the licence plate is detected by the second camera.

Each parking space is equipped with a UDMS and a red and green LED light. The UDMS detects if a car is located above it, and consequently if the parking spot is occupied. The LED lights indicate the state of the spot. If the green LED is on, the parking space is available. When the red light turns on, the parking spot is

[1] All images without explicit source attribution are proprietary.

[2] See Table 3 in Appendix C for a complete overview of all used parts and their model numbers. Their respective prices are found in Table 1.

occupied. At the entrance of the garage, there is a seven segment display that shows the amount of available spaces.

The model of the parking garage was first designed using Solid Edge and then laser cut out of MDF-plates in Fablab.

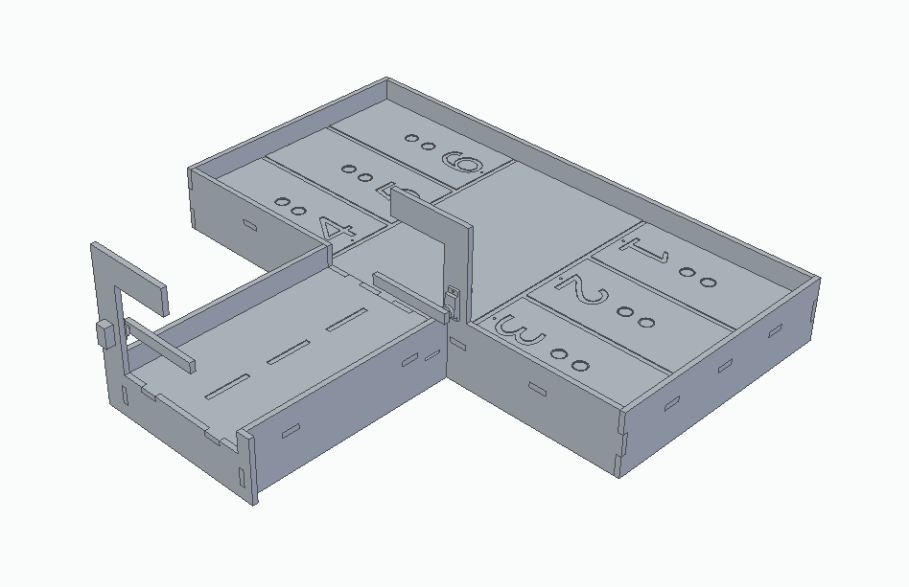

Figure 1: 3D-model of the parking garage.

2.2 Design alternatives

This problem can be solved in many different ways. One of the first problems is the decision of what 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 you don't need 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. If a car is detected the camera takes a picture and sends a request to the Google Vision Application Programming Interface (API) (see Section 3.1.1 for more details). 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 doesn't offer any extra advantages over the UDMS. Therefore our design will use the UDMSs (HC-SR04).

2.2.1 Design motivation

As shown in Section 2.2 the elements of the final design were not chosen randomly. The model has 6 parking spaces. This provides enough space to give a thorough demo without requiring too much time and money to build. The model itself is build with MDF-plates. These are cost-efficient and still deliver good quality for a base structure.
The spots are equipped with two separate LED lights, because the department is already in possession of the one-colored lights. This reduces the waiting time and keeps the cost of the garage low.

The detection of cars (both in the parking spots and at the entrance/exit) happens with ultrasonic sensors. These are cheap, easy to use and are specific enough for their intended use. The entry and exit also have this sensor. Because of the inefficiency of constantly running the cameras. The Raspberry Pi won't be able to

handle two constant information flows.

The cameras used for the ANPR are 'DORHEA Raspberry Pi Mini Kamera'. These cameras are compatible with the Raspberry Pi. They use an image of 5 MP, which should be enough for the ANPR to recognise the licence plate. It is not worth it to use a 12 MP camera because the ANPR would work in both cases and the 5 MP cameras are a lot cheaper.

3 Software design

The software system consists of three main parts: the Raspberry Pi, the frontend application and the backend server.[3] Figure 7 in Appendix A shows the complete deployment diagram of the system. The following paragraphs will each explore one of these components in more detail, given special attention to the functionalities and the deployment of the different modules.

3.1 Raspberry Pi

The Raspberry Pi runs four Python packages inside a Docker container (see Section 3.1.2) for the four main functionalities which the Raspberry Pi has to fulfill: 1) ANPR; 2) read in sensor data from the distance sensors; 3) control the signaling LEDs; 4) control the servo motors of the barrier. The separate Python packages provide a segmented approach to installing all the dependencies of the different packages.

3.1.1 Automatic number plate recognition

The Raspberry Pi uses ANPR to identify cars coming in and out of the garage. Its goal is to reliably detect the licence plates 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 such dataset 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. Therefore, an algorithm can find candidates of licence plates by converting the image to gray-scale and applying a threshold on 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 on 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. This 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 the 1000 free requests.

3.1.2 Deployment

The Raspberry Pi used in this project runs a 32-bit Operating System (OS), which makes it impossible to run 64-bit software, e.g. the 64-bit version of Python.[4] The ANPR uses some 64-bit packages like `torch`[5]. Therefore, all the functionality of the Raspberry Pi is packaged in 4 Python packages, which are deployed with Docker[6] in so-called *containers*. More specifically, this deployment will use Docker Compose, which can run multiple containers as a single service, making it possible for the different containers to interact and communicate with each other [Docker, 2022]. The container itself also bundles all the dependencies with an OS, which prevents a very tedious setup of all dependencies, which can be very different due to the different

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

[4] 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].

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

[6] `https://www.docker.com/`

OS that the Raspberry Pi uses (32-bit Raspbian). Note that the backend also uses Docker due to a similar reason (see Section 3.3.3). Figure 2 shows a schematic overview of the interaction between the software on the Raspberry Pi and the hardware components.

The Raspberry Pi has to communicate to the backend database in order to update the different tables regarding the garages and parking lots and to query the database, to – for example – receive information whether the user has paid or not.

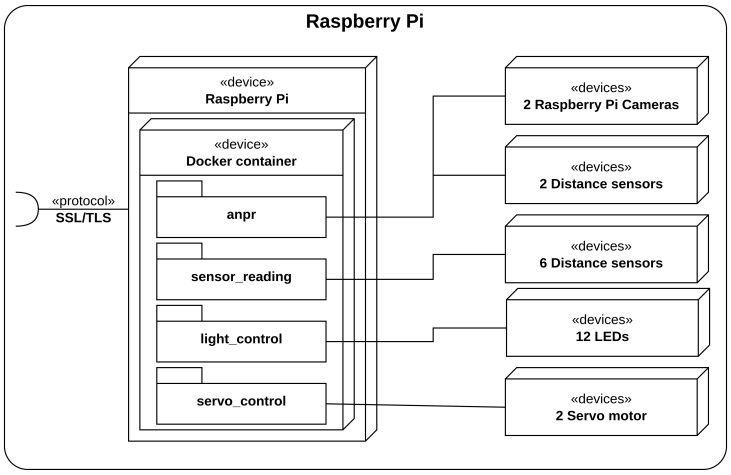

Figure 2: Deployment diagram of the Raspberry Pi dependant system.

3.2 Frontend application

The frontend application will be written in Dart, with the Flutter[7] framework of Google. Other valid alternatives were primarily JavaScript (JS)-frameworks (e.g. React[8] or AngularJS[9]). 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.

3.2.1 First app design

The first page that users see when they open the app is the login screen, here they will have the option to either login with 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 they have to fill in their first name, last name, licence plate, e-mail address and password. These credentials are used to create a secure account for a user that is linked to their licence plate for the automatic payment system. The user is required to confirm the 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 able to register their account.

Once the account is created, the user can login by hitting the "Sign in"-button. When that's done, the homepage will pop up and the app is going to make a request to the backend server to load all the possible garages. While the app is connecting there will be a progress indicator on the screen and the user will still be able to access other buttons like the navigation bar. After all the garages have been loaded into the app, you can select one of the garages to make a reservation. You can filter out the possible garages by giving in the name of the location where you want to book a reservation. Next the reservation screen will pop up of the selected garage. On this screen the user can observe the busiest times of the day and how many spots are left in the garage. If the user is satisfied of this garage, than he/she can book a reservation. There will be the option to choose the time and day and an available spot.

For users who are not familiar with using the app or a mobile app in general, there will be a guide in the navigation bar. Furthermore, the user can see his/her statistics, profile, reservations and adjust his/her settings. A detailed schematic of the entire app can be found in Appendix B in Figure 8.

[7] `https://flutter.dev/`
[8] `https://reactjs.org/`
[9] `https://angularjs.org/`

3.2.2 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 our demonstration, 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 (see Section 3.3.3), namely Nginx[10]. Apart from being a reverse-proxy for the backend, it also hosts the static files (HTML and JS). Figure 3 shows the deployment diagram for the frontend application. Note that the two client devices represent both options of the client of connection to our backend API.

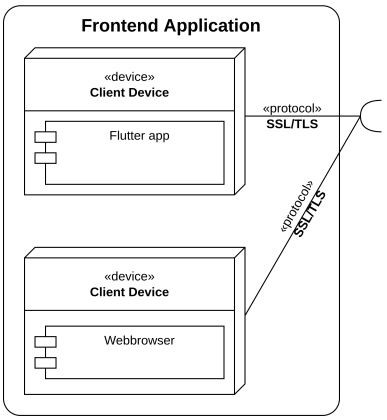

Figure 3: Deployment diagram of the frontend application.

3.3 Backend server

3.3.1 Functionality

For the purpose of our project, the backend is defined as the combination of the servers, database and backend application written in Django.

The main functionality of the backend is storing and retrieving all data relating to the garages, parking lots and users. This is accomplished with a Representational State Transfer (REST) API, which sends the data in a JavaScript Object Notation (JSON)-format. The use case of the project requires two main objectives of the backend: security and availability.

Security is one of the main spearheads of this project. The following measures have been taken on a backend level.

Firstly, communication between the frontend application and the backend takes place with Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS). SSL/TLS encrypts and signs data which is send over the web, in order to guarantee data confidentiality and integrity. This way, no third party can intercept the data send between the client and the server [Cloudflare, 2022].

Secondly, all user-identifiable data is hashed in the database, this includes email-addresses and passwords. Due to the small number of combinations of licence plates, hashing has no added benefit (it could easily be cracked with a brute-force attack). This way, even if the database would be corrupted, hackers don't get access to confidential user information.

Thirdly, API access is only granted to users who are logged in. Therefore, the backend uses a *token authentication* system. On every login, a token is automatically generated, which has to be sent in the API-request, as an authentication header. Furthermore, this token can also be used to dynamically log in the user to a mobile application, eliminating the user's need to re-enter their credentials. To make this procedure more secure, the

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

backend uses an extra package, `django-rest-knox`, which – contrary to the standard tokens provided by the Django Rest Framework (see below) – hashes the tokens in the database [James1345, 2013].

The second main objective is availability. The backend server should be reachable from everywhere at all time. In practise, this would be an absolute necessity. If not, clients wouldn't be able to exit or enter the parking garage, nor being able to do payments. For the purpose of the demonstration, the backend is deployed on the DNetCloud infrastructure. This cloud is only reachable via a Virtual Private Network (VPN)-connection, so a VPN-client has to be installed on every device communicating with the backend (i.e. the Raspberry Pi and the client devices). For the purpose of the demonstration, a VPN-client will be installed on these devices. Of course this procedure isn't workable in a real-world scenario, in which the backend would be hosted by a cloud provider like Amazon Web Services (AWS)[11] or Google Cloud Platform[12].

3.3.2 Infrastructure

The backend application is written with Django, an open-source web framework, written in Python. Before this framework was chosen, alternatives such as: Ruby on Rails[13] (written in Ruby), Next[14] (written in JavaScript/TypeScript) or Laravel[15] (written in PHP). 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 doens't 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 what we need [Django, 2022].

In general, a web framework serves three purposes. First and foremost, it serves as an abstraction for the database tables and provides *models* (which are Python classes in Django) for defining data types. This is done with an Object Relational Mapper (ORM) which can transform Python code into Structured Query Language (SQL)-queries [MDN Contributors, 2022]. Secondly, it defines *controllers* which can map URLs to functions which can run on the server. Lastly, *views* allow to write HyperText Markup Language (HTML) with embedded functionality. Together, they form a Model-View-Controller (MVC)-framework. Views aren't used in our backend, because the backend serves as an REST API.

In extension to the Django framework, the backend application uses the Django Rest Framework[16]. A Python package specialized in building REST APIs.

The Django application interacts with a MySQL-database[17], an open-source Relational Database Management System (RDBMS). An important property of a MySQL-database is the fact that it's *schema-full*, meaning that all data entering the database has to be structured in the form of the database field and columns. This is a welcome addition which compensates the lack of a solid type system for Python. The other widely used RDBMS is PostgreSQL[18], but this type of database is mainly used for building complex relation models and provides a lot of extra features which isn't necessary for our project [IBM, 2022].

3.3.3 Deployment

The deployment of a production-proof server system for a Django application which supports SSL/TLS and concurrency isn't self-evident, as the Django `python manage.py runserver`-command (the default way to start the Django development server) may not be used for production purposes [Django Docs, 2022].

Nginx serves as an industry standard for a fast and lightweight server. The important feature for the backend is that it can handle SSL/TLS and redirect HyperText Transport Protocol (HTTP)-requests to HyperText Transport Protocol Secure (HTTPS)-requests [Nginx, 2022].

Besides a web server, the Python application needs a way to communicate between the web server and the actual application (in casu our Django application like written above). This is achieved with a Web Server Gateway Interface (WSGI). The backend uses Gunicorn[19] 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 to overall availability of the system [Chakon, 2017].

[11] `https://aws.amazon.com/`
[12] `https://cloud.google.com/`
[13] `https://rubyonrails.org/`
[14] `https://nextjs.org/`
[15] `https://laravel.com/`
[16] `https://www.django-rest-framework.org/`
[17] `https://www.mysql.com/`
[18] `https://www.postgresql.org/`
[19] `https://gunicorn.org/`

Together with the Django application, the web server and the gateway form the three essential building blocks of a production-proof deployment.

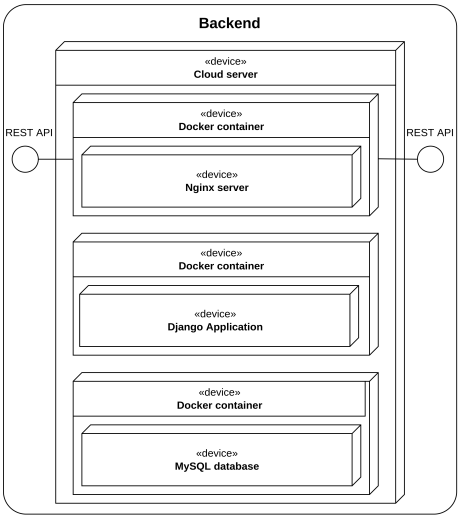

Figure 4: Deployment diagram of the backend server software.

The services above require a lot of dependencies and configuration files, which can make it tedious to set up on a remote cloud. The backend is therefore deployed in Docker containers, which bundles *Docker images* with an OS in a so-called isolated container. This way, all the dependencies are packed inside the container, which eliminates the need of doing a laborious setup. 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 4 shows the deployment diagram of the entire backend software.

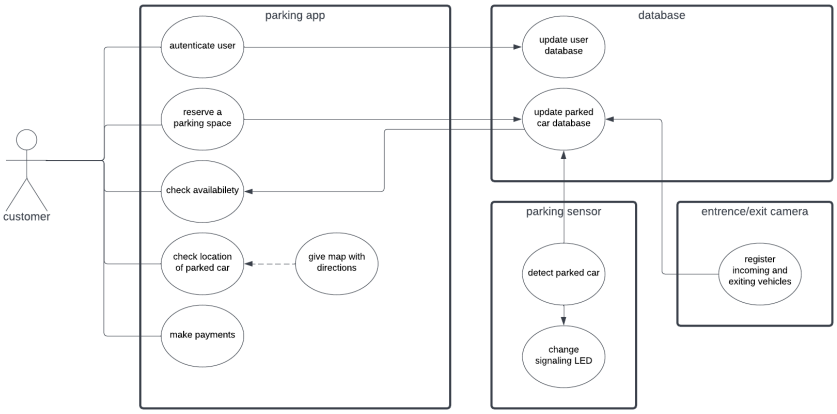

Figure 5: Use case diagram of a normal user.

3.4 Use case diagrams

From a backend's perspective, there are two types of users: a normal user and an garage owner. The former represent clients of the parking garage. All users get this type by default. The latter represent the 'administrator' of the garage, which has a superset of privileges over normal users. Therefore, the use case diagrams are divided in two parts, the use cases for a normal user and the use case for a garage owner.

A normal user should be able to interact with the app in several ways: the user can authenticate themselves by either logging in to an already existing account or creating a new account, reserve a parking space so it will be kept unoccupied, check the availability of parking spaces in a chosen parking garage, check the location of their vehicle in the parking garage and automatically or manually pay the due bill. All these functionalities

should be supported by the frontend application and will have different screens in the application (see Section 3.2). These screens then perform API-queries which will subsequently update the respective databases. Figure 5 shows a schematic overview of all the different functionalities which the user has to be able to perform.
If a user is authenticated as a garage owner, he/she will see an extra tab in the application with an extra set of possibilities regarding managing his/her parking garage. A garage owner should be able to – completely separate from the application designers – install new garages to the system. This means adding a garage and updating its information (e.g. location, opening hours, prices, etc.) and adding/disabling parking lots in the application. This cuts down the cost for the garages making use of our system. Figure 6 shows the different extra functionalities which the owner has to be able to perform in the application.

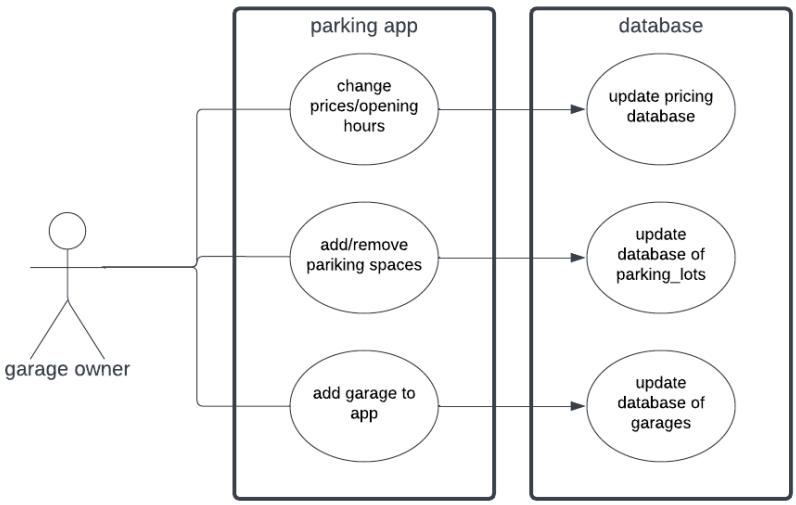

Figure 6: Use case diagram of an owner.

4 Example user experience

After an abstract explanation about the different components of both the software and the hard system, this section gives a real-world example of a user experience in the parking garage. Of course, in subsequent reports, these actions are going to be made more concrete with sequence diagrams. Due to the large size of the flowcharts, they're included in Appendix D.
The process begins with the user who drives towards the entry barrier. A UDMS detects the car and sends a signal to the ANPR-camera which takes a picture of licence plate. This picture is then analyzed by the Google Vision API. The recognised string from the licence plate is then 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 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 if the users exits the garage, compliant to current General Data Protection Regulation (GDPR)-guidelines. Figure 9 in Appendix D shows a schematic overview of the entering process.

After entering the garage, the user drives to his/her 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, a UDMS detects the cars, so that the Raspberry Pi can sent an API-request to 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 10 in Appendix D shows a schematic overview of this process.

When exiting the garage, it's recommended that the user pays its ticket in advance, to make the exiting-process run smoothly, but the system also supports payments in front of the barrier of 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 hasn't payed, 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 11 in Appendix D shows a schematic overview of the exiting process.

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

5 Budget

Table 1: Current budget state.

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	6.16	12.32
Raspberry Pi 3B	1	59.95	59.95
Total Price		151.43	
Remaining		98.57	

Table 1 gives an overview of the budget. 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. Right now the design only uses a total of 151.43 euros. You can see this in the second to last row of Table 1. This means that the budget is not nearly reached with a surplus of 98.57 euros.

6 Planning

In the first weeks of the project, a solid foundation has been laid, which can be built upon during the upcoming weeks. This, by no means, indicates progress will slow down. The following tasks are the most important to complete in the next weeks:

- solid communication between the frontend application and backend server;
- secure authentication support by the backend;
- two factor authentication support by both the frontend and the backend;
- Python modules for the Raspberry Pi;
- secure payment system in the frontend application.

Most of the tasks above are complex and difficult to implement correctly, but they are indispensable for our project. Also note that these points are the backbone of the project, many smaller tasks, like making it possible to reserve a parking lot aren't included, but will of course be fulfilled.
Table 2 shows the planning for the upcoming weeks of the semester. There has been decided to finish the three major parts of the system well in advance, in order to have enough time to fully connect the components so that they can interact flawlessly.

Table 2: the planning for the remaining weeks (weeks 6–12 of the first semester).

Due date	Item
04/11/2022	Communication between frontend and backend
11/11/2022	Secure backend authentication system
18/11/2022	Working two factor authentication system Working Python modules for the Raspberry Pi
25/11/2022	Working secure and automatic payment system in frontend

7 Conclusion

The work of the previous weeks has lead to a solid foundation on which can be built upon in the upcoming weeks. There's a concrete design of the major parts of the system, namely the frontend application, the backend server and the Raspberry Pi. Furthermore, a scale model of the parking garage has already been realised. All major functionalities of the different systems have been designed and connected to each other in theory. The main work of the forthcoming weeks is bringing the theory into practise and realising all the details of the different systems.

8 Course integration

This project is a sequel of it's predecessors P&O 1 and P&O 2. So the most knowledge and experience that's been used for this project came from these courses. In these courses things like writing reports (in LaTeX), keeping track of a logbook, making presentations, ect. were taught. These are basic aspects that are needed for creating a good project. As mentioned earlier, the experience that's 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 used for the Raspberry Pi and licence plate recognition. Along with Python, Dart and JS were used and these languages were 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 used for solving the optimization 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 realize this project. But it's a good basis to understand and learn new advanced topics in this field.

References

[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.

[Cloudflare, 2022] Cloudflare (2022). What is SSL? — SSL definition. [Online]. Last accessed on 22/10/2022. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-ssl/.

[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/.

[Django Docs, 2022] Django Docs (2022). `django-admin` and `manage.py`. [Online]. Last accessed on 23/10/2022. Retrieved from https://docs.djangoproject.com/en/4.1/ref/django-admin/.

[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/.

[Flutter, 2022] Flutter (2022). Hot reload. [Online]. Last accessed on 27/10/2022. Retrieved from https://docs.flutter.dev/development/tools/hot-reload.

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

[IBM, 2022] IBM (2022). PostgreSQL vs. MySQL: What's the Difference? [Online]. Last accessed on 22/10/2022. Retrieved from https://www.ibm.com/cloud/blog/postgresql-vs-mysql-whats-the-difference.

[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.

[James1345, 2013] James1345 (2013). `django-rest-knox`. [Online GitHub repository]. Last accessed on 23/10/2022. Retrieved from https://github.com/charlespwd/project-title.

[MDN Contributors, 2022] MDN Contributors (2022). Server-side web frameworks. [Online]. Last accessed on 22/10/2022. Retrieved from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[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/.

Appendices

A General deployment diagram

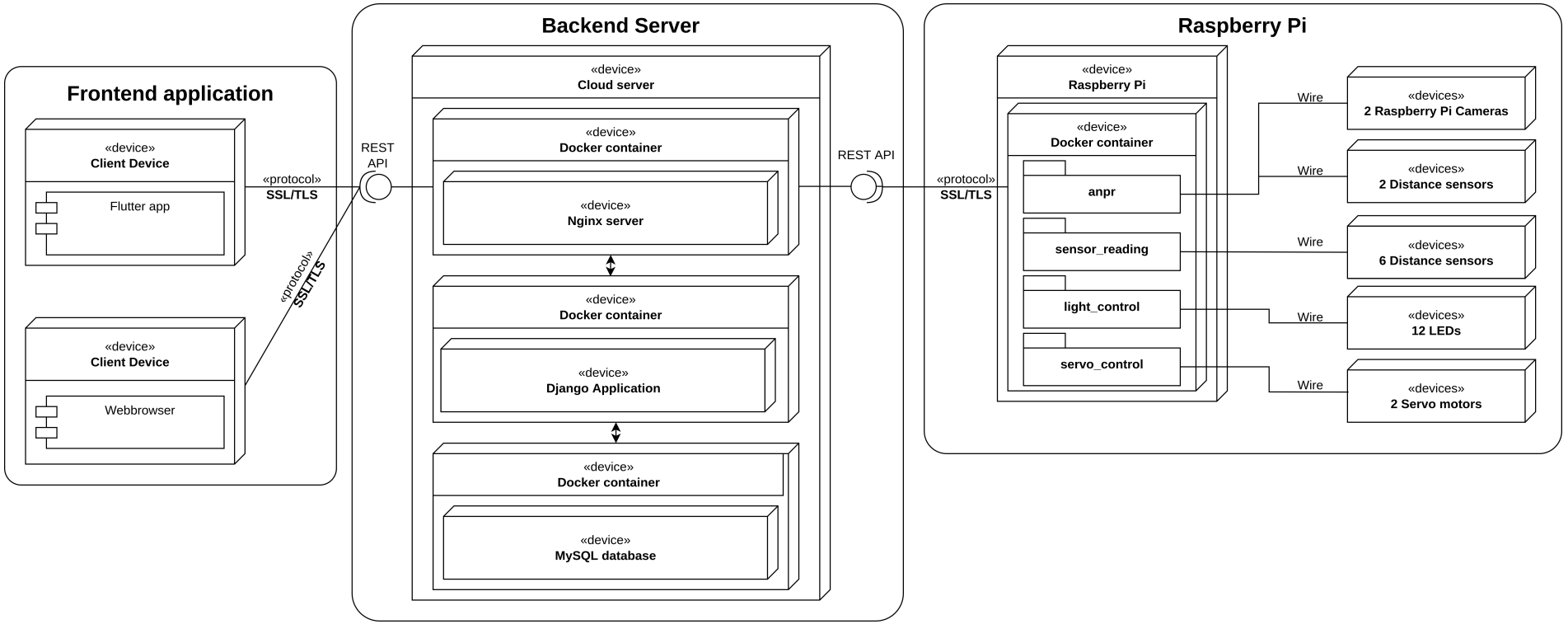

Figure 7: General deployment diagram.

B Application design

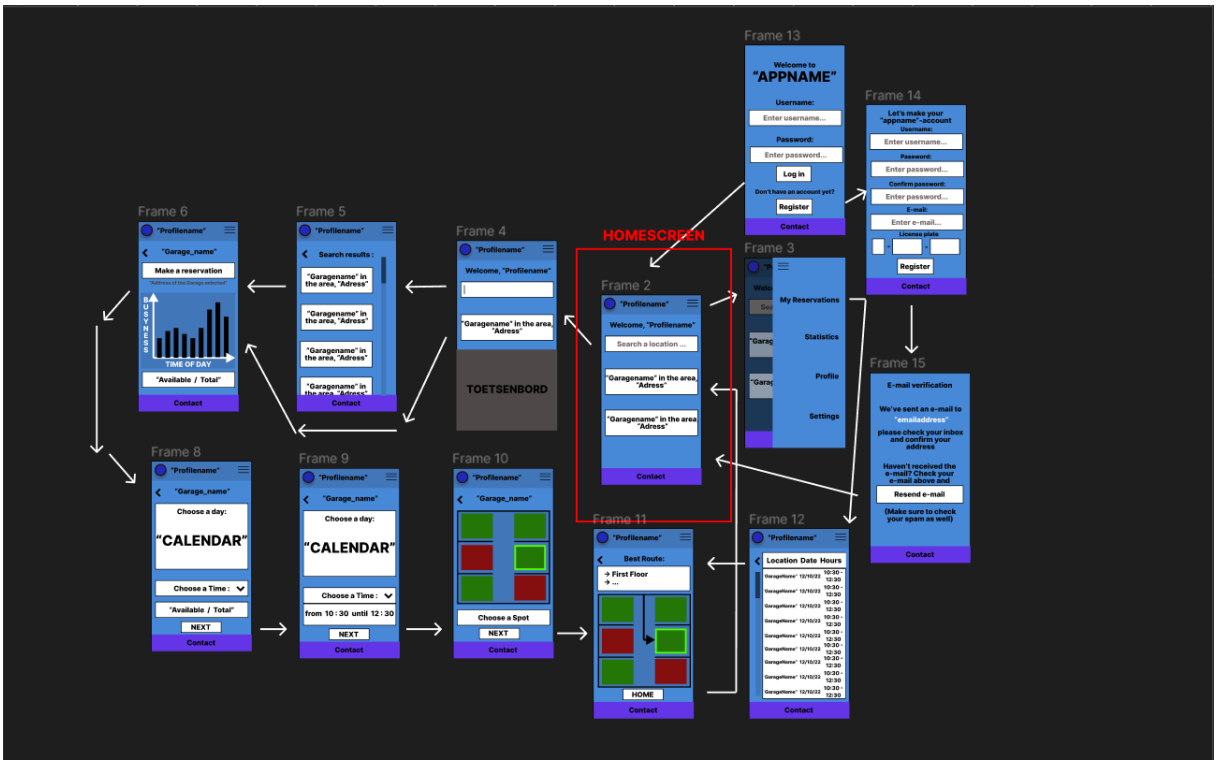

Figure 8: Current app design.

C Mechanical part list

Table 3: Overview of all used mechanical components and their model number.

Component name	Model number	Amount
Raspberry Pi	Model 3B	1
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	/	≈ 60
Raspberry Pi camera extension cable	B087DFJ2RP	2

D Flowcharts

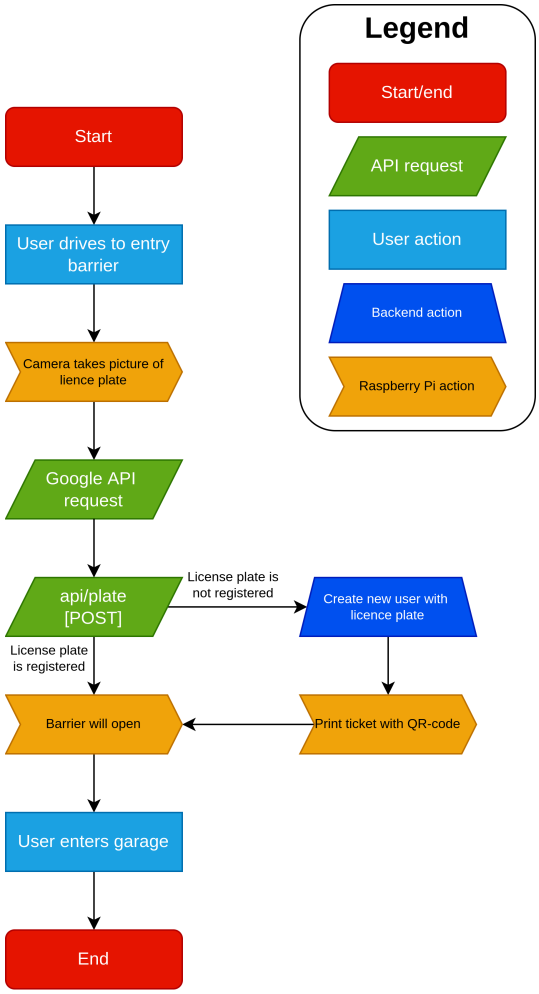

Figure 9: Flowchart of the entering process of the garage in both hardware, software and user terms.

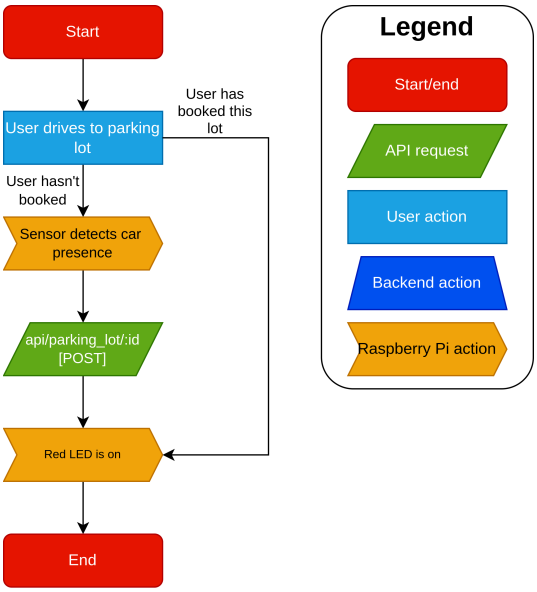

Figure 10: Flowchart of the car detection process of the garage in both hardware, software and user terms.

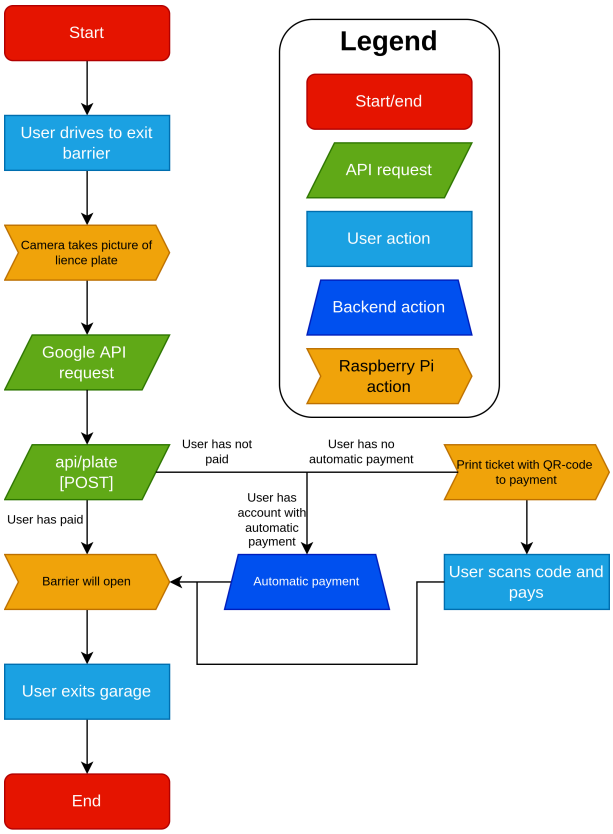

Figure 11: Flowchart of the exiting process of the garage in both hardware, software and user terms.