TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Maksim Maksimov 121088IAPB

DEVELOPMENT OF CONTROL AND MONITORING SOFTWARE FOR ROBOTS OPERATING IN A DISTRIBUTED SYSTEM

Bachelor's thesis

Supervisor: Martin Rebane
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Maksim Maksimov 121088IAPB

JUHTIMIS- JA MONITOORINGUTARKVARA ARENDUS HAJUSSÜSTEEMIS TOIMIVATELE ROBOTITELE

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Maksim Maksimov

18.05.2021

Abstract

The main focus of this thesis is to create a user interface based on modular and reusable components, as well as to create an prototype for controlling one or more robots.

The aim of this project is to create a robot, whose main task is to transport goods and medicine equipment between hospital floors and wards. This topic is especially important today during the pandemic, because it helps to reduce people's contact with other surfaces and eliminates the need for humans to move between different wards, thus making less chance for a person to become infected with the virus.

This is a group work of about 10 people in which my assignment is to make a working prototype of an application for controlling robots: a simple version that is needed to control only one robot directly from a tablet attached to it, and a more complex version for administering several robots from one place.

Since the project is at an active stage of development, many of the initial solutions were changed in favor of better ones after more detailed discussions with the team and new requirements, taking into account the security, user experience and extensibility of the future product.

The paper describes the complete development process for this solution, as well as the complexity and technological diversity of the project.

This thesis is written in English and is 44 pages long, including 5 chapters and 20 figures.

Annotatsioon

Juhtimis- ja monitooringutarkvara arendus hajussüsteemis toimivatele robotitele

Selle lõputöö põhirõhk on moodulitel ja korduvkasutatavatel komponentidel põhineva kasutajaliidese loomine ning prototüübi loomine ühe või mitme roboti juhtimiseks.

Selle projekti eesmärk on luua robot, mille peamine ülesanne on kaupade ja meditsiinitehnika transport haigla korruste ja palatite vahel. See teema on tänapäeval eriti oluline pandeemia ajal, sest see aitab vähendada inimeste kontakti teiste pindadega ja välistab inimese vajaduse liikuda erinevate palatite vahel, võimaldades seega inimesel vähem võimalusi viirusesse nakatuda.

See on umbes 10 inimesest koosnev grupitöö, kus minu ülesandeks on teha robotite juhtimiseks rakenduse toimiv prototüüp: lihtne versioon, mis on vajalik ainult ühe roboti juhtimiseks otse sellele lisatud tahvelarvutist, ja keerukam versioon mitme roboti ühest kohast haldamiseks.

Kuna projekt on aktiivses arengujärgus, muudeti paljusid esialgseid lahendusi paremate kasuks pärast üksikasjalikumaid arutelusid meeskonnaga ja uusi nõudeid, võttes arvesse tulevase toote turvalisust, kasutajakogemust ja laiendatavust.

Artiklis kirjeldatakse selle lahenduse täielikku arendusprotsessi, samuti projekti keerukust ja tehnoloogilist mitmekesisust.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 44 leheküljel, 5 peatükki ja 20 joonist.

List of abbreviations and terms

API	*Application Programming Interface* - a set of definitions and protocols for building and integration application software
CSS	*Cascading Style Sheets*
DOM	*Document Object Model* - object model for HTML
HTTP	*Hypertext Transfer Protocol*
JavaScript	The programming language for the Web
LESS	Dynamic preprocessor style sheet language that can be compiled into CSS
MJPEG	*Motion JPEG*
MQTT	*Message Queuing Telemetry Transport*
ROS	*Robot Operating System*
RPC	*Remote procedure call*
TypeScript	A typed superset of JavaScript that compiles to plain JavaScript
UI	*User Interface*
Stencil	Frontend framework for building *Web Components*
Vue	Model–view–viewmodel frontend JavaScript framework
Web Components	Set of features allowing for encapsulation and interoperability of individual HTML elements
WebSocket	Two-way connection channel between client and server.

Table of Contents

List of Figures

1 Introduction

The idea of creating this kind of project came from the North Estonia Medical Centre. The North Estonia Medical Centre is one of the top healthcare providers in the country. As a regional hospital, it has the highest-level competence to provide specialised medical care.

The main goal of this project, launched in September 2020, is to create a robot that will move medicines, as well as any other necessary goods between different floors and wards of the hospital without human assistance. The use of such a robot eliminates the need to move people from one part of the hospital to another, which in turn reduces the risk of spreading viruses and diseases. This topic is especially relevant today, during a pandemic. Also, this solution reduces the workload on staff.

My main goal in this project will be to create a user-friendly software for controlling and monitoring robots. The requirements that need to be achieved in this work are described in more detail in Section 2.4. Writing of this work will affect many aspects of the development of various types, ranging from the design of the application interface to designing the architecture and database, making Android application. For more information about the technologies in favor of which solutions were chosen, see Section 2.5.

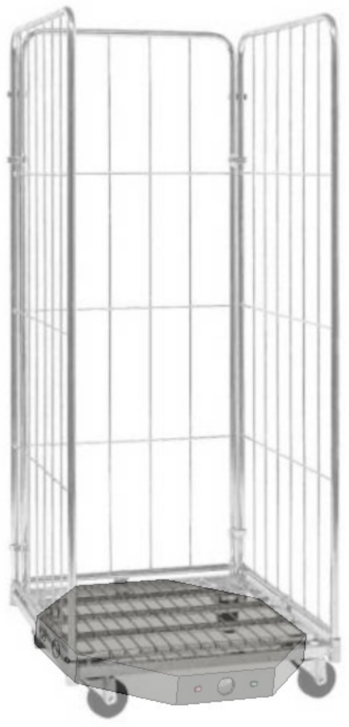


Figure 1: Autonomous mesh carrier robot

Figure 1 shows what an autonomous robot would roughly look like, to which a standard hospital mesh carrier is attached.

The main goal of the project is to create an initial prototype of a transport robot that meets the requirements of the hospital in collaboration with TalTech and PERH. As a result of the implementation of the project, it is expected that the autonomous transport robot will pass tests and work successfully, as well as be useful in practice in a regional hospital in Northern Estonia.

2 Project overview

The robots have the ability to exchange information with elevators, automatic doors, other robots, as well as with a central server. When moving around the hospital, the robot will use a line specially marked on the floor for it. Thus, it will always be clear to others which trajectory will be chosen.

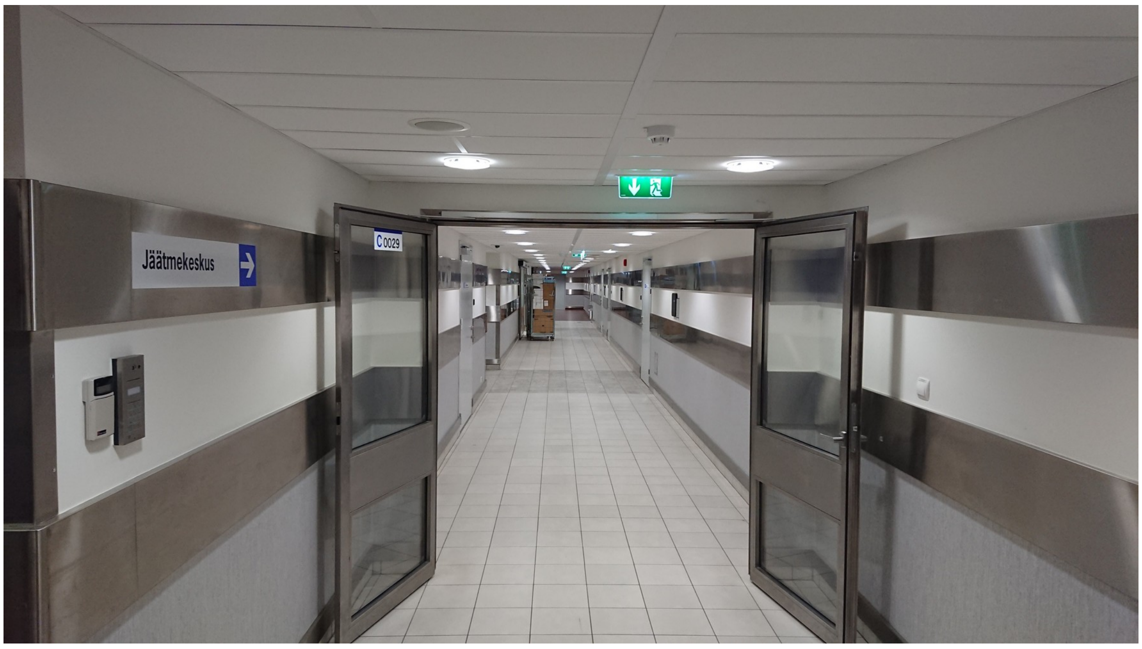


Figure 2: PERH environment example photo

One of the main things to do in the entire hospital is to mark lines on the floor for the robots to move. Figure 2 shows automatic doors, as well as an example of the room itself where robots will move.

The initial prototype of the robot is equipped with:

- Four electric motors
- Two electric motor controllers
- Lithium-ion Power Battery 36V 4.4Ah
- Omni-wheels for 360 degrees maneuverability
- Attached Android-powered tablet to control the robot
- Barebone Intel NUC PC with processor i7-10710U 1.1 GHz
- Intel RealSense Depth Camera D435

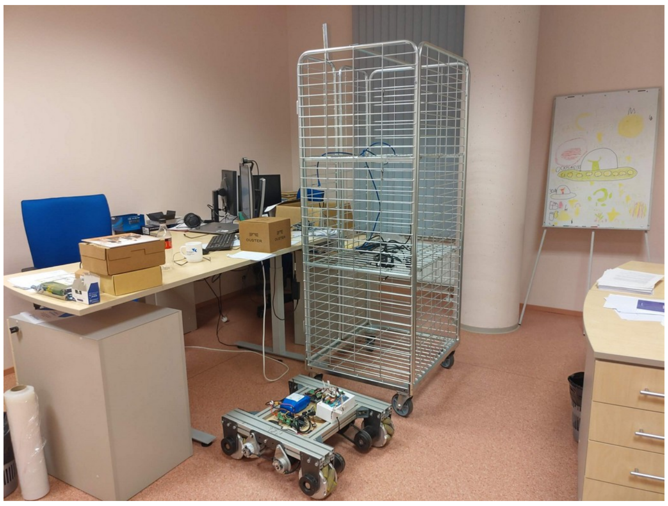


Figure 3: Robot prototype and mesh carrier

The maximum load for the robot is about 500 kilograms, sufficient to transport completely different goods. Figure 3 shows one of the stages of the development of the robot. Nearby is a standard mesh carrier that is used in hospitals.

2.1 Distributed systems

A *distributed system* is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system. The components interact with one another in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components [1] .

There are many different types of implementations for the message passing mechanism, including pure HTTP, RPC-like connectors and message queues [2] . The system used in this work uses the message queuing mechanism, the principle of which is described in more detail in Section 4.2.

2.2 Robot Operating System

ROS is used as the software installed on the robot. Robot Operating System (ROS or ros) is an open source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server [3] . Processing takes place in nodes that can receive, publish, and multiplex data from sensors, control, state, scheduling, actuator, and other messages. In this project, for the most part, asynchronous data streaming over the topics was used, the principle of which you can see in Figure 4.

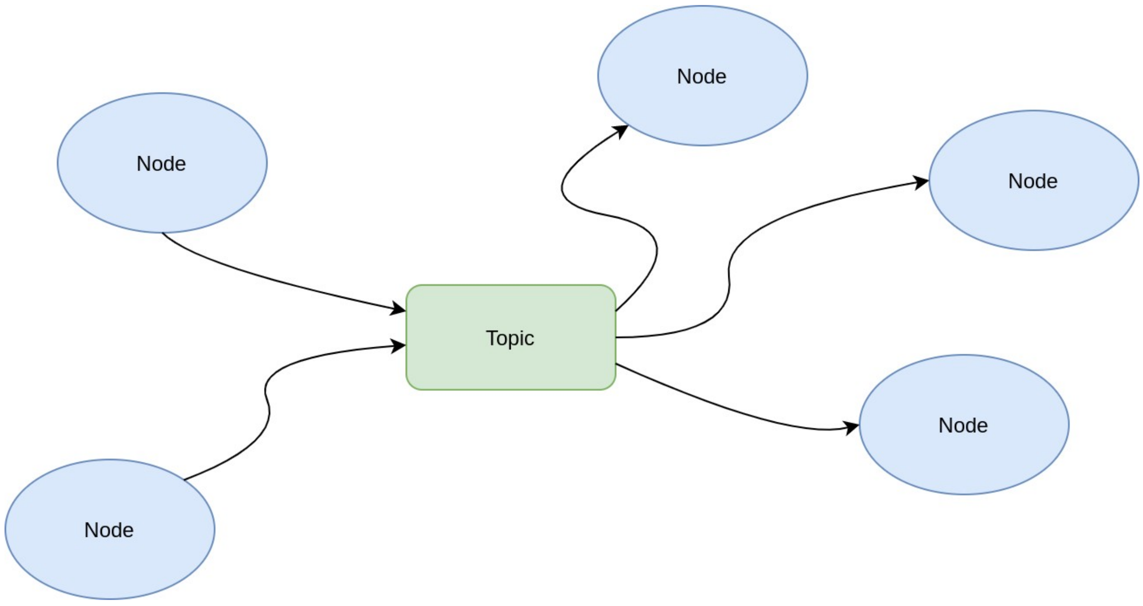


Figure 4: Message and topics in ROS

2.3 Hardware

In order for the robot to be able to move freely around the hospital, it needs to be able to communicate with different types of blocking obstacles, such as doors and elevators.

The base component for all electronics used as door and elevator controllers is the ESP32 [4] . It is a series of low cost, low power consumption microcontrollers. They are a system-on-a-chip with integrated Wi-Fi and Bluetooth connectivity.

In the hospital itself, there are doors of various types, some are of a new type and the ability to open / close is already present through the security system and is performed by a simple API request, and some are of the old type and for which you have to create your own controllers to achieve this functionality. With elevators, the situation is different, and such an opportunity exists for no one. See more detailed explanation in Section 2.3.1.

2.3.1 Elevator

In order for the robot to be able to ride the elevators, several modifications must be made. The first is to be able to control them - go to different floors, open / close doors. The second is the ability to track the actions of the robot to control and correct its movements, as well as track people in the elevator. To cope with this task, a suitable device has been designed. It consists of a thermal sensor and a passive infrared sensor (PIR sensor). The sensor enclosure, shown in Figure 5 uses custom design and is 3D printed with PLA material. The elevator controller communicates with the backend API via MQTT, which allows for fast information exchange.

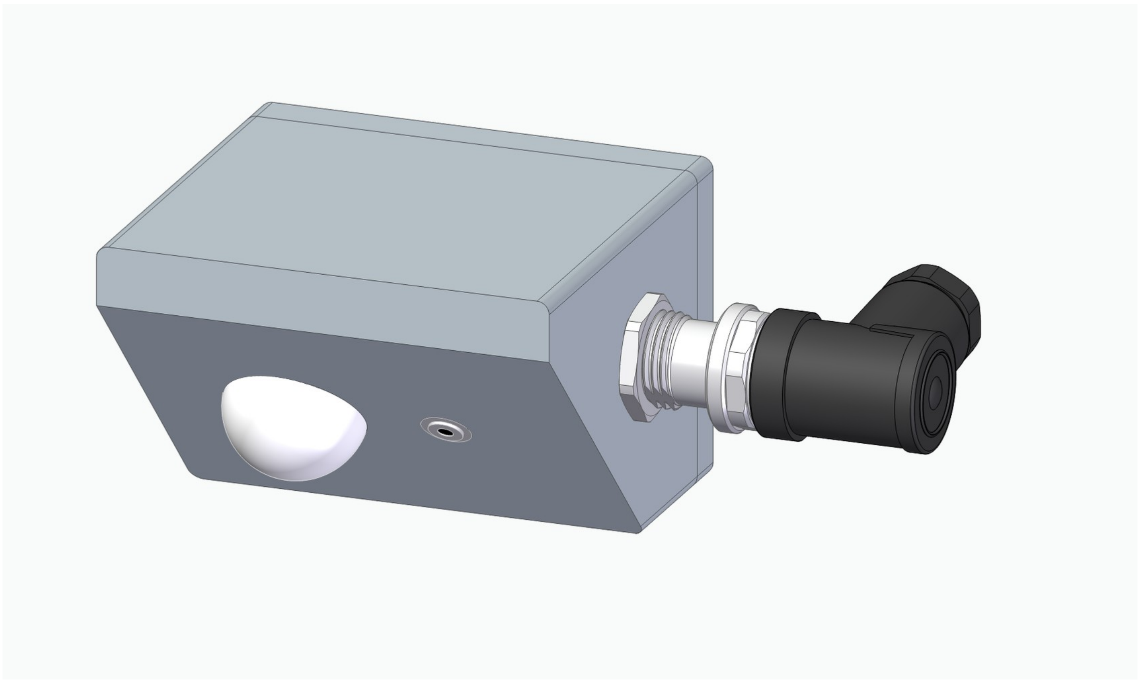


Figure 5: Elevator sensor isometric view

2.4 Requirements

When creating a control and monitoring system, many requirements must be met.

Business requirements:

- Develop a web-based user interface by using nowadays technologies
- Application interface must be built on reusable web components
- The designed system must be secure

Functional:

- There should be a page to track the status of one robot
- The ability to see the streaming image from the robot
- Ability to move the robot to its destination
- The ability to see the status of each robot in real-time

- Ability to see events from robots

Non-functional:

- The application interface should be simple and straightforward
- Workers should be able to easily control the robot

2.5 Technologies

Variability of the stack and an abundance of technology used in this project a little striking. For writing everything related to hardware, the C++ language was mainly used and ESP32 as the electronics platform. The backend server was written in Python. The robot itself contains ROS with written modules (nodelets) that also use the Python language. The frontend uses mainly Typescript, which in turn compiles to JavaScript. Interface styling is done in LESS language, which is also compiled, into CSS. *Stencil* was chosen among the frameworks for the frontend. Vue was also tried as an alternative, but it did not satisfy the requirement to use reusable web components, this is also described in Section 3.3. The project also slightly touched on the development of an Android application, a small part of which was written in Java. *Stencil* to Android conversion was achieved using the Capacitor library.

Various options were considered for transferring data between devices in a distributed system: XMPP, Kafka, AMQP, MQTT. The choice was made in favor of MQTT which is standard protocol for the Internet of Things (IoT). For communication between the user interface and the frontend server, HTTP was used as a channel for making requests to the server and WebSocket for sending data from the server. Two database management systems for the frontend - Redis and MySQL and one for the backend - SQLite. Containerization is achieved via Docker using docker-compose and HTTP / WebSocket traffic proxying is done with Nginx – detailed in Section 4.2. And of course some bash scripts for deployment and Node.js with Yarn for frontend development. As the main IDE - VSCodium was used, the rest of the things related to the Android application - Android Studio.

3 User interface development

The creation of the user interface took most of the development because it should be simple and user-friendly, and also based on modular and reusable components. This part goes through a lot of trial and error, ranging from using CSS frameworks such as Bootstrap and Tailwind, to functional frameworks for developing interfaces, like Vue and *Stencil*. In Section 3.4 describes the experience of migrating the entire application to the Android platform.

3.1 Design of the interface

When planning the interface, it was necessary to take into account the fact that it had to work both on a tablet and on a standard computer screen. It should be responsive as well as user friendly and understandable for the average user. There should also be a separate adapted page for the tablet.

The system design consisted of 4 main screens, which in turn split the applications into two separate applications: the robot user interface and the administrator user interface.

- Separate robot tablet page - Robot UI (see Figure 7)
- All robots overview (see Figure 16)
- Robot detail page (see Figure 17)
- Infrastructure overview (see Figure 20)

Since it would have taken a long time to work in Adobe Photoshop or other graphics editor alternative, it was easier to lay out the basics of HTML and CSS to coordinate and select the initial design. Bootstrap was chosen as the basis for UI and Font Awesome as the font and icon toolkit. Bootstrap is a free and open-source CSS framework directed at responsive, mobile-first frontend web development. It contains

CSS- and JavaScript-based design templates for typography, forms, buttons, navigation, and other interface components.

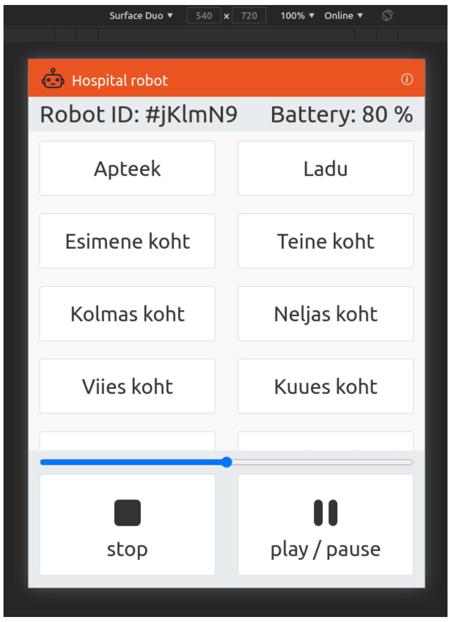
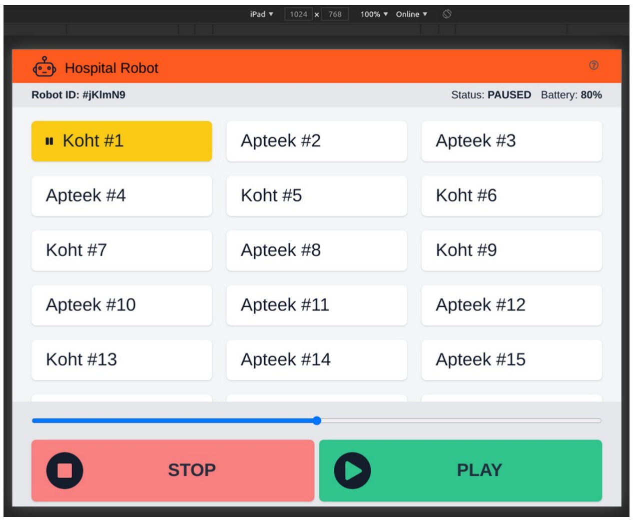


Figure 6: Robot UI designed using Bootstrap

After initial sketches and approval of the main components of the system, it was decided to move to using Tailwind, another more suitable framework. Tailwind [6] is a feature-oriented CSS framework containing useful classes that you can compose to create any design right in the markup. Utility classes help you work within the constraints of the system instead of cluttering your style sheets with arbitrary values. They make it easy to be consistent with color choices, typography, shadows, spacing, and everything else that makes up a well-engineered design system. Using this framework will save time and be ready for quick changes without much need to rewrite styles.

The interface for the Robot UI is pretty simple (see Figure 7). It consists of two fixed panels at the top and bottom, with content area in between. In the bottom panel there are two buttons for control and an indicator showing the state of completion of the route above them. The destination is highlighted in the content area.


iPad 1024 x 768 100% Online

Robot ID: #jKlmN9

Status: PAUSED Battery: 80%

Figure 7: Robot UI designed using Tailwind

3.2 First prototype

Vue [7] was taken as the main framework for the initial prototype. One of the key features of Vue is the ability to build into web components. However, Vue is *externalized*, which means that in order for the created components to work, it must be globally available on the host page. When creating this solution, Typescript and LESS were used. Also at this stage, a mock server was written with support for exchanging information with the client in near real-time, which is described in Section 4.1.

3.3 Using Web Components for future compatibility

Web Components is a suite of different technologies allowing you to create reusable custom elements - with their functionality encapsulated away from the rest of your code - and utilize them in your web apps [8] . The problem with using Vue in web components was that the components couldn't work without including this framework in the host page, meaning there was still a dependency on the framework. It was decided to try a new better alternative framework called *Stencil*.

Stencil is a compiler that generates Web Components (more specifically, Custom Elements) and builds high performance web apps. *Stencil* combines the best concepts of the most popular frameworks into a simple build-time tool. Since *Stencil* generates standards-compliant web components, they can work with many popular frameworks right out of the box, and can be used without a framework because they are just web components. APIs like Virtual DOM, JSX, and async rendering make fast, powerful components easy to create, while still maintaining 100% compatibility with Web Components [5] .

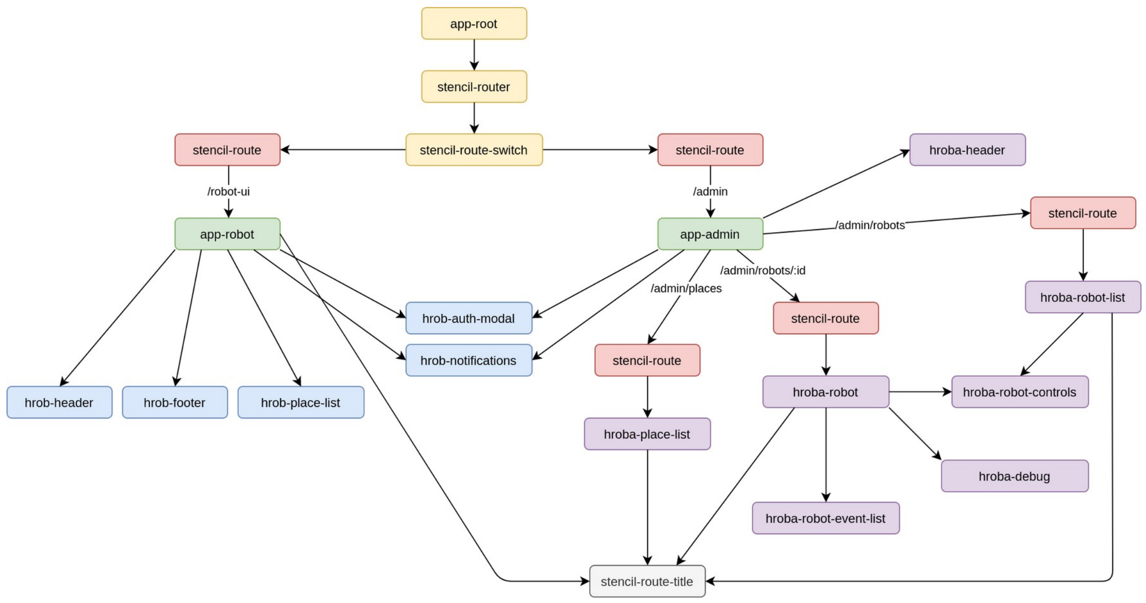


Figure 8: Component structure in Web UI application

Figure 8 describes architecture of the frontend application, where each node is a web component. Routes are defined in *app-root* using core components prefixed with *stencil* keyword. The components marked in green are essentially controllers in which all data manipulations take place, which are sent through properties to child components. In addition, these components are responsible for binding and listening for events from the event emitter. In addition to using *Stencil*'s standard event approaches, there was also a third-party library *event-emitter3* for more complex event operations. Every component uses Shadow DOM [11] which involves cutting the desired styles from Tailwind. Using the tailwind.config.js config file, it was quite easy to do this by specifying an array of files to look up the class names in the purge property. In addition, a lightweight shared state library called Store from *Stencil* core was used to display the notification queue.

The abbreviation for hospital robot - *hrob* was chosen as the prefix for the reusable web components used in Robot UI, *hroba* for Admin UI. How routing looks from the code side in the *app-root* component can be seen in Appendix 2. The total size of the entire application, including the admin part, which will be discussed in Section 4.3.5, is about 675 kilobytes.

3.4 Android application

Nowadays, in order to develop an Android application, it is not necessary to write it in such native languages as Java or Kotlin. You can simply convert an existing web application using the latest advances in this area. There are many solutions on the market to achieve these goals: React Native, NativeScript, Apache Cordova, Capacitor and so on. Our choice fell in favor of Capacitor. Capacitor is a cross-platform API and code execution layer that makes it easy to call Native SDKs from web code. The advantage of using it is that you can write custom native plugins that your application may need or use existing ones written specifically for Apache Cordova, which in turn expands the possibilities of using this tool.

In order for the application to be safely used within the walls of the hospital, several problems had to be solved:

- Should always be used by the only application without the ability to exit it or go to the settings
- Impossibility of switching applications
- The application should start when the device boots
- The device must always be in an active state so that the user does not have to look for the power button
- The application must run in full screen mode

By implementing them in the application, in fact, we will turn it into a COSU app (Corporate Owned Single Use application) [12] or a Single-Use device. To accomplish this task, Java code was written and some configurations were added to make our application a device administrator.

The final size of the compiled application on a tablet is about 9 megabytes.

4 Providing near real-time performance

In order to add the ability to transfer data from the web server about the robot to the client, it was decided to use WebSockets. WebSocket [8] is a computer communication protocol that provides full duplex communication channels over a single TCP connection. Using this technology creates less overhead than half-duplex alternatives such as HTTP polling, making it easier to transfer data in real-time to and from the server.

4.1 Mock server

To simulate the actions and more clearly imagine how the system will work as a whole, it was decided to use the mock server first. It was written in Node.js using JavaScript. Node.js is an cross-platform backend JavaScript runtime environment that executes JavaScript code outside a web browser.

To create the application, was used Express, a web server application framework for Node.js. The endpoints are described in more detail in the Section 4.3.

4.2 General architecture

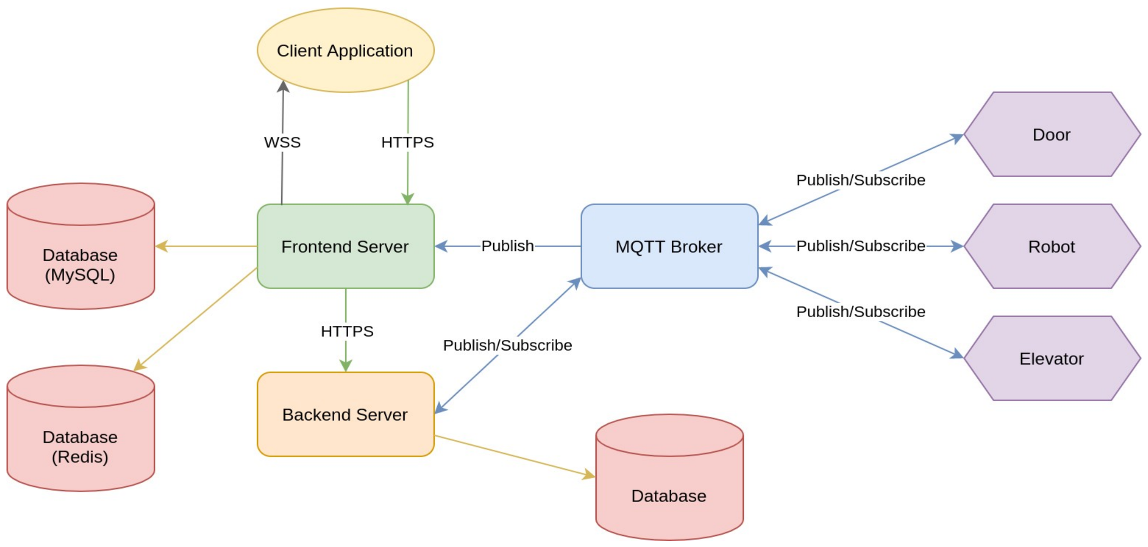


Figure 9: General architecture

In Figure 9, you can see the overall architecture of the project. MQTT connections are marked in blue and all IoT devices communicate through it. WebSocket Secure (WSS) and Hypertext Transfer Protocol Secure (HTTPS) are used to denote secure protocols.

The Message Queuing Telemetry Transport (MQTT) is a lightweight, publish-subscribe network protocol that transports messages between devices. The protocol usually runs over TCP/IP; however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT [9] . The *Backend Server* can subscribe and publish messages to the MQTT broker, however the *Frontend Server* can only subscribe to MQTT topics which is necessary to obtain a stream of images and other metrics of the robot arriving in real-time without having to proxy it from the *Backend Server*. With MQTT broker architecture, the devices and application becomes decoupled and more secure. *Frontend Server* topic restrictions is done in a Mosquitto access control list (ACL) file via mosquitto.conf [10] on the broker which prevents unauthorized publishing and subscribing to restricted topics.

The robot, as well as the control tablet attached to it, are connected to the local network separately from each other, via Wi-Fi. Using the Admin UI is possible from any device in this local network.

4.3 Server for communicating with the application

Several endpoints have been created for the API:

- **GET /robot/:id** - returns data about the robot
- **POST /robot/:id/move** - is used to launch the robot to a specific location and triggers the transmission of the robot's status via a WebSocket.
- **POST /robot/:id/status** - changing the status of the robot. Used for pause and stop
- **GET /robot/:id/stream** – returns default robot MJPEG image stream (see Section 4.3.4)
- **GET /place** - returns the places available for visiting by the robot
- **POST /auth** – authentication method, session cookie will be added after success

To identify the robot, the *Robot-Secret* header is sent with every HTTP request from the Robot UI. The value for this field will be contained in the robot control tablet.

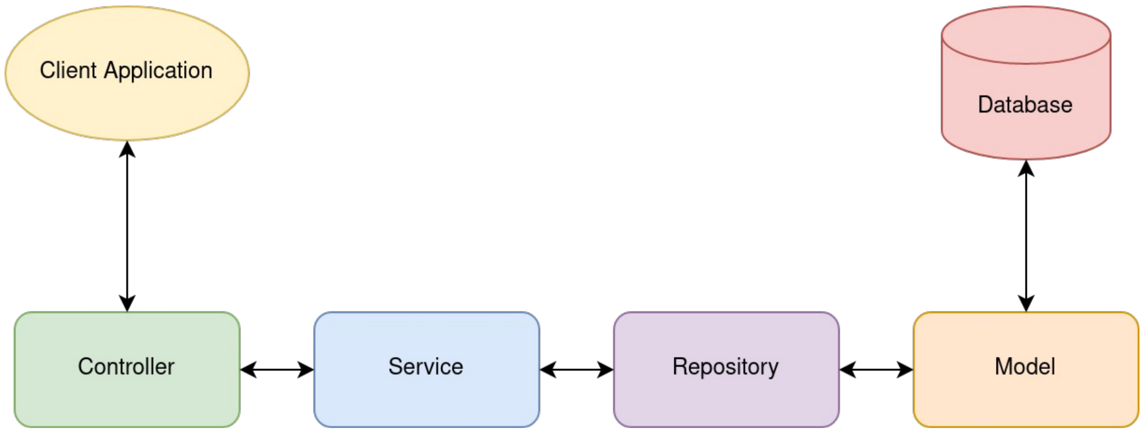


Figure 10: Controller-Service-Repository architecture

Controller-Service-Repository architecture was chosen to build the backend application. It is a great approach for breaking up the business layer of the app into two distinct layers. This pattern relies on dependency injection to work properly. Dependency Injection (DI) [13] is a practice where objects are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally. *TypeDI* package was chosen as the dependency injection tool.

4.3.1 Real-time communication

WebSockets are used to send real-time events from the frontend server to the frontend application. While the robot is moving, it gives a *moveRobot* message to the client. When stopped and finished, the corresponding *robotStop* and *robotFinish* events will be dispatched. The sequence diagram in Figure 11 describes the process of launching the robot to its destination. It shows how the entire system works, how data is transmitted and through which channels. In the figure, there is also a repeated event loop that occurs until the *Backend Server* checks for the condition that the robot has reached its destination, after which the event about the end of the movement is transmitted to the frontend.

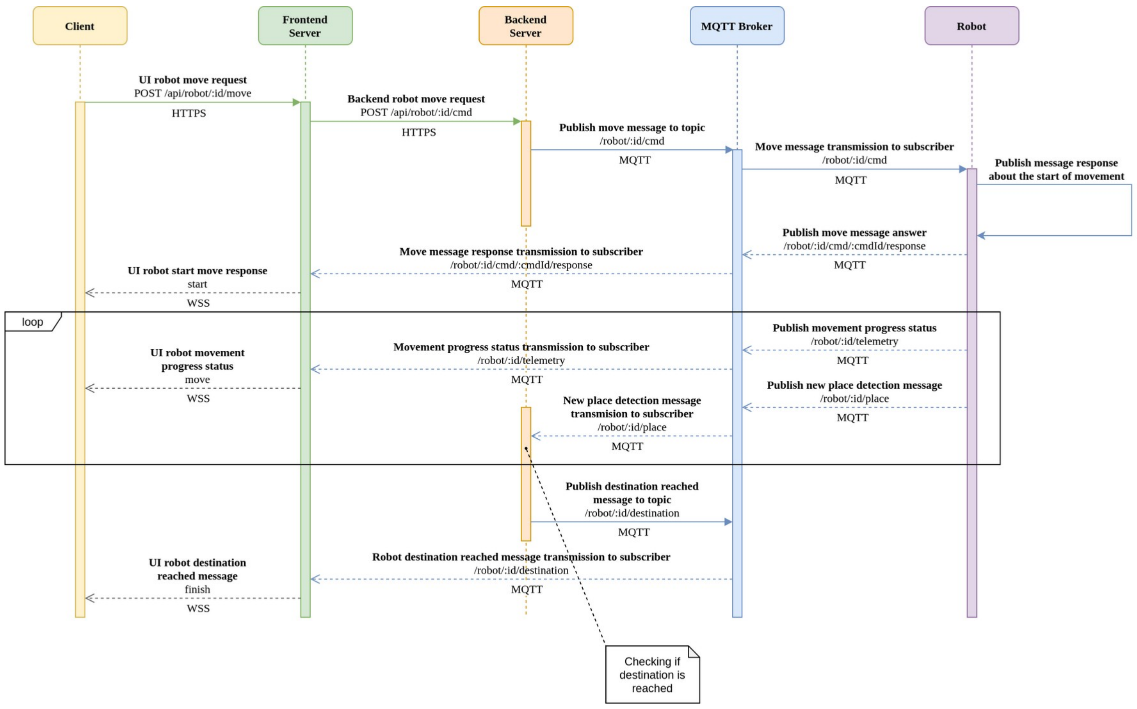

Client
Frontend Server
Backend Server
MQTT Broker
Robot
UI robot move request
POST /api/robot/:id/move
HTTPS
Backend robot move request
POST /api/robot/:id/cmd
HTTPS
Publish move message to topic
/robot/:id/cmd
MQTT
Move message transmission to subscriber
/robot/:id/cmd
MQTT
Publish message response about the start of movement
Publish move message answer
/robot/:id/cmd/:cmdId/response
MQTT
Move message response transmission to subscriber
/robot/:id/cmd/:cmdId/response
MQTT
UI robot start move response
start
WSS
loop
Publish movement progress status
/robot/:id/telemetry
MQTT
Movement progress status transmission to subscriber
/robot/:id/telemetry
MQTT
UI robot movement progress status
move
WSS
Publish new place detection message
/robot/:id/place
MQTT
New place detection message transmision to subscriber
/robot/:id/place
MQTT
Publish destination reached message to topic
/robot/:id/destination
MQTT
Robot destination reached message transmission to subscriber
/robot/:id/destination
MQTT
UI robot destination reached message
finish
WSS
Checking if destination is reached

Figure 11: Robot movement sequence diagram

4.3.2 Request flow

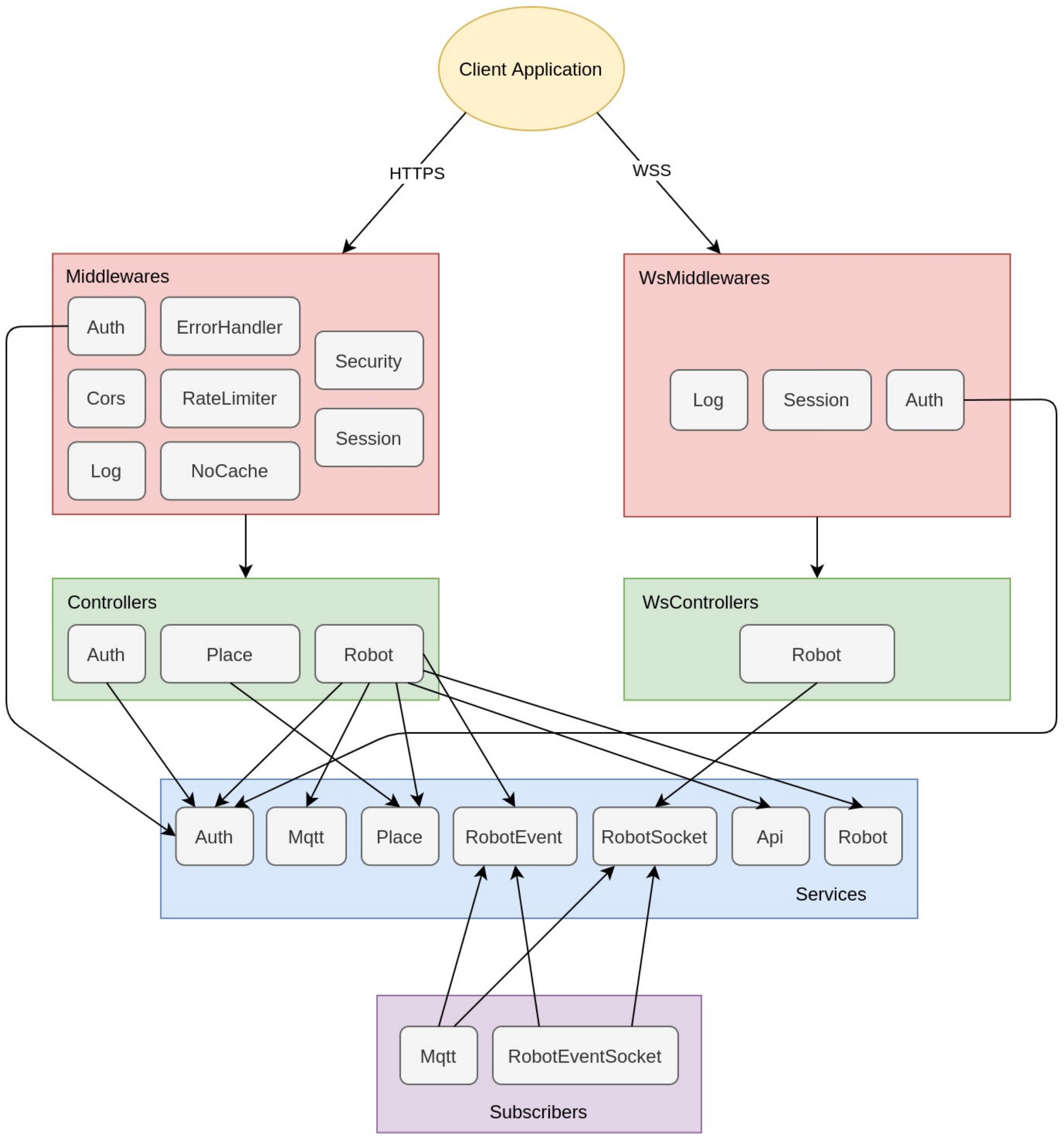


Figure 12: Application API requests flow

Middleware is used to handle requests before and after controllers. Sessions, logging, authorization and others are implemented through middleware. The WebSocket middleware works mostly on opening connections, the main logic for the messages themselves is in the *RobotSocket* service. Authentication middleware uses an authentication service to provide the required functionality, which is accomplished through dependency injection.

Subscribers are used for sending messages to WebSocket clients from anywhere in the application without the need to include service.

Sessions are used for all API requests. Since a solution based on *socket.io* was chosen for WebSocket connections, a session expiration check was added to each *ping* and *pong* request on the server side. Redis was used as a rate limiter and session storage. Redis (Remote Dictionary Server) is a fast in-memory key-value data store for use as a database [15] .

4.3.3 Database

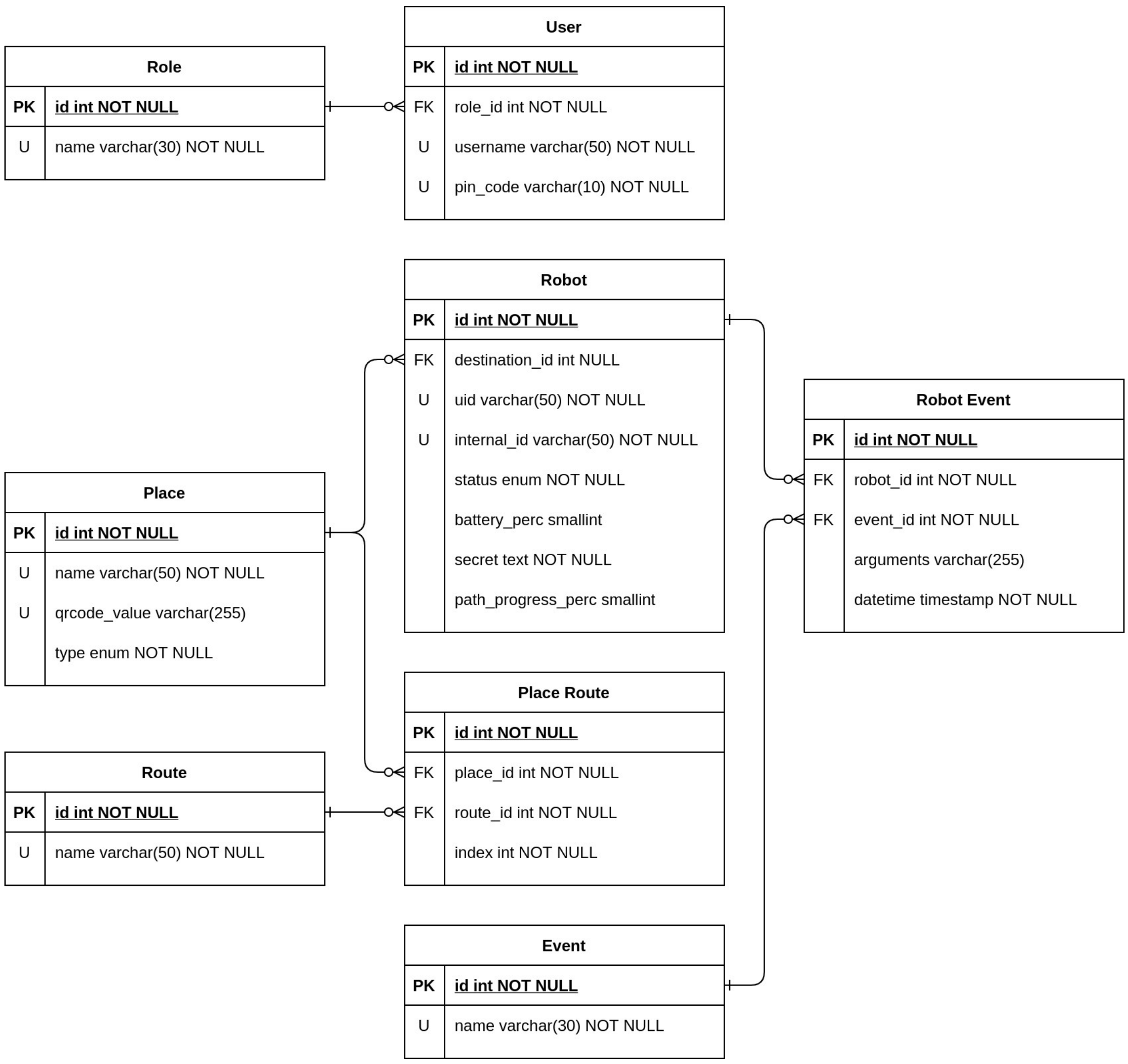


Figure 13: Database schema for frontend application

In Figure 13 the database schema for frontend application can be seen. FK - means *foreign key*, U - *unique key*. For clarity of writing, the names of the tables are written in an easy-to-read format, in fact, the snake case is used. A list of all places, elevators, doors are contained in the *Place* table, where the attribute *type* determines the type of place. Events for all robots can be found in the *Robot Event* table, the types of events used are in the *Event* table. These tables are needed to display the list of events on the detailed page of the robot (Figure 17). For the functionality of the routes and the output of the necessary checkpoints in the future, the *Route* and *Place Route* tables are used, where the *index* attribute is the ordinal number of the place to compose a complete route.

Also, the project uses data seeding. This is the process of filling the database with initial data, for example, roles for users, this in turn happens through factories using typeorm-seeding package. Since the project has an admin panel, the administrator role is needed for such cases. Database migrations are done with typeorm itself.

4.3.4 Streaming images from a camera

The application should have a image stream from the robot in an amicable way, so that you can observe in real time what it sees and how it behaves, catch problems if necessary and resolve them in a timely manner. To test the required functionality, I was lent an Intel RealSense Depth Camera D435 [16] , essentially the same that will be used on working robots.

In order for the application to receive frames from the camera, it was first necessary to transmit them from the robot itself. For this, a Python script was written along with a launch file to use it as a ROS nodelet. Nodelets are designed to provide a way to run multiple algorithms on a single machine, in a single process, without incurring copy costs when passing messages intraprocess. roscpp has optimizations to do zero copy pointer passing between publish and subscribe calls within the same node [17] . The script consist of class that listens to the desired ROS topics with a image streams and sends them to the MQTT topics as a Base64 [14] encoded string. The script itself also has some arguments: host, port, username, password, ca_file.

The Node.js app server then listens on the MQTT topics and decodes the Base64 string into an MJPEG image streams. The image stream is delivered until the session is terminated. How the data exchange takes place can be seen in more detail in Figure 14.

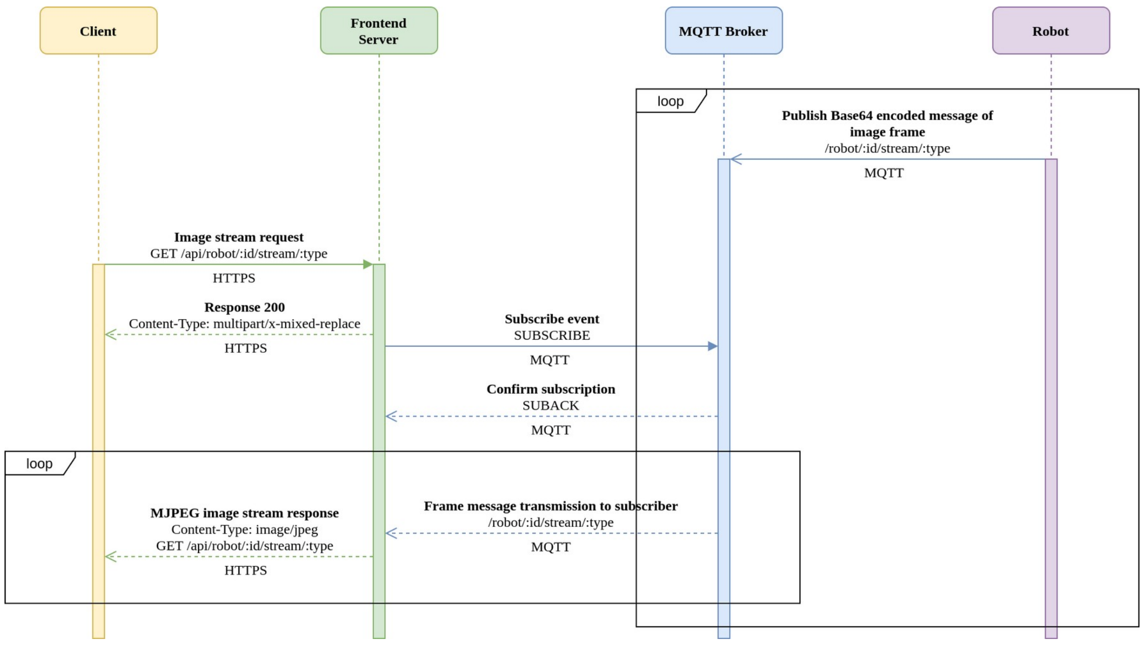


Figure 14: Image streaming sequence diagram

The application uses a link of the form /api/robot/{id}/stream/{type}, where *id* is the robot id and *type* is a stream type. For Robot UI *type* is not used, the default image stream will be used instead.

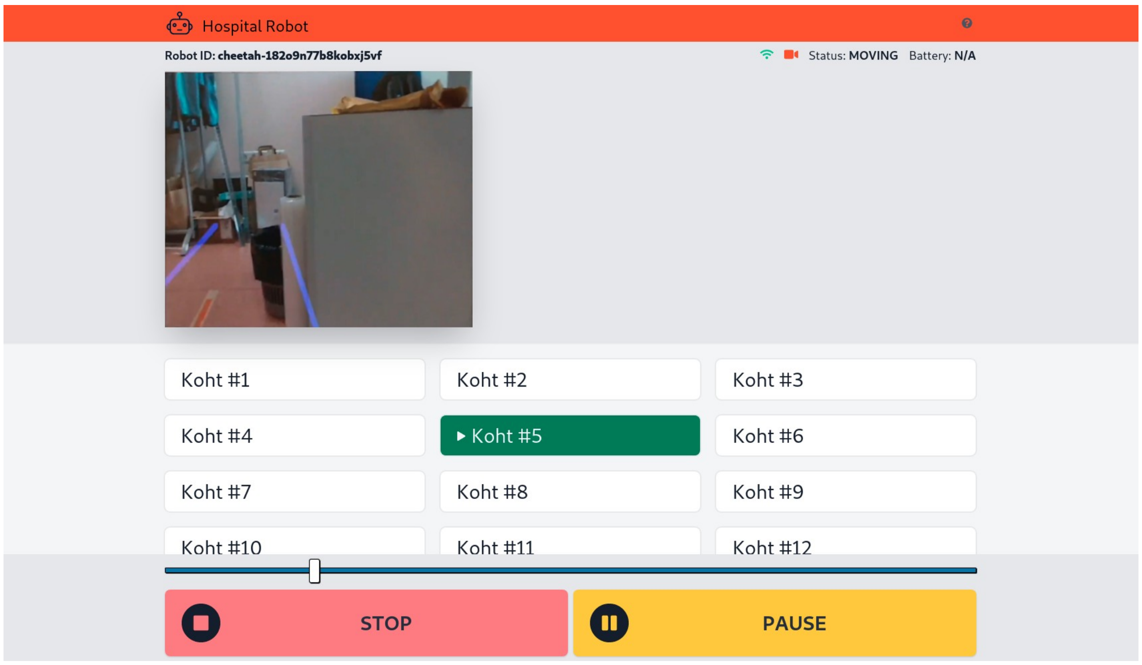


Figure 15: Robot UI image stream

In the robot's user interface (Figure 15), the image stream appears on the top fixed panel that appears over the content area when you click on the corresponding camera icon. What the streaming image looks like in the admin panel is shown in Figure 18.

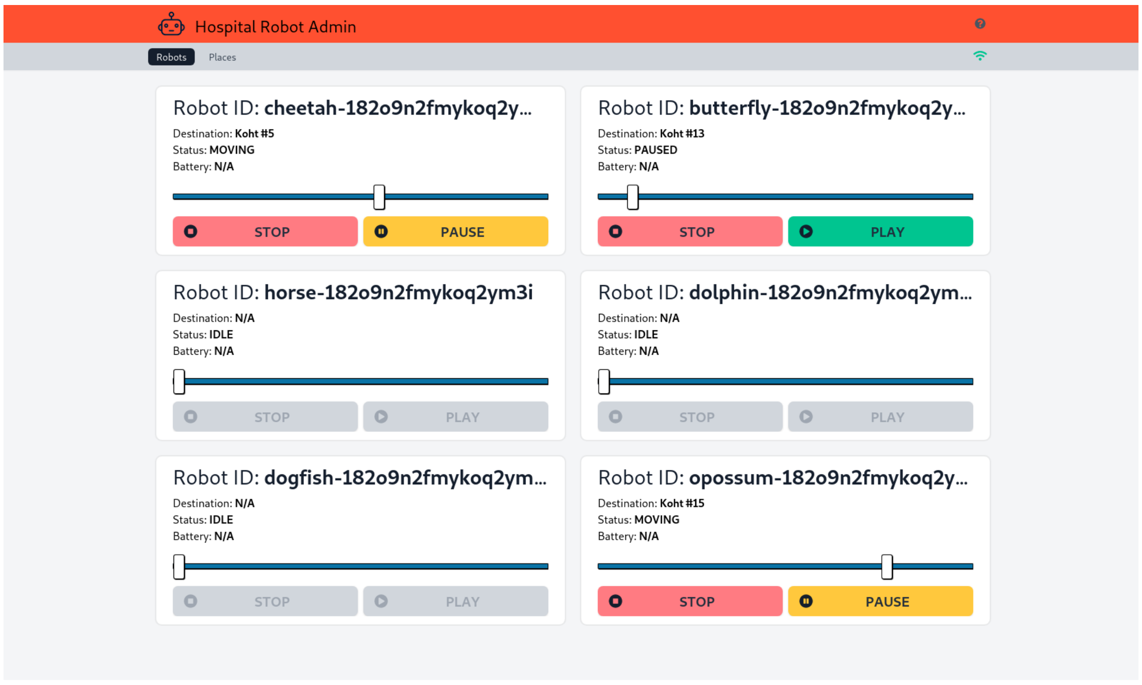
4.3.5 Administration

Another part of the application that also had to be worked on is the administration page.

In addition to the endpoints described in Section 4.3, a few more have been added:

- **GET /robot/:id/event** - returns data about robot events
- **GET /robot/:id/debug** - helper endpoint for debug console (see Figure 19)
- **GET /robot/:id/stream/:type** – returns the type-specified MJPEG robot image stream (see Section 4.3.4)

The first screen is an overview screen that shows the states of all robots, which can be seen in Figure 16. In admin mode, the WebSocket connection sends events from all robots at once, which allows you to see the picture in real time. Unlike Robot UI, there is navigation in the admin panel in the top fixed menu. Now there are only two main screens: *Robots* - a screen with all robots (Figure 16) and *Places* - an overview of the infrastructure (Figure 20).



Robot ID: cheetah-182o9n2fmykoq2y...
Destination: Koht #5
Status: MOVING
Battery: N/A

Robot ID: butterfly-182o9n2fmykoq2y...
Destination: Koht #13
Status: PAUSED
Battery: N/A

Robot ID: horse-182o9n2fmykoq2ym3i
Destination: N/A
Status: IDLE
Battery: N/A

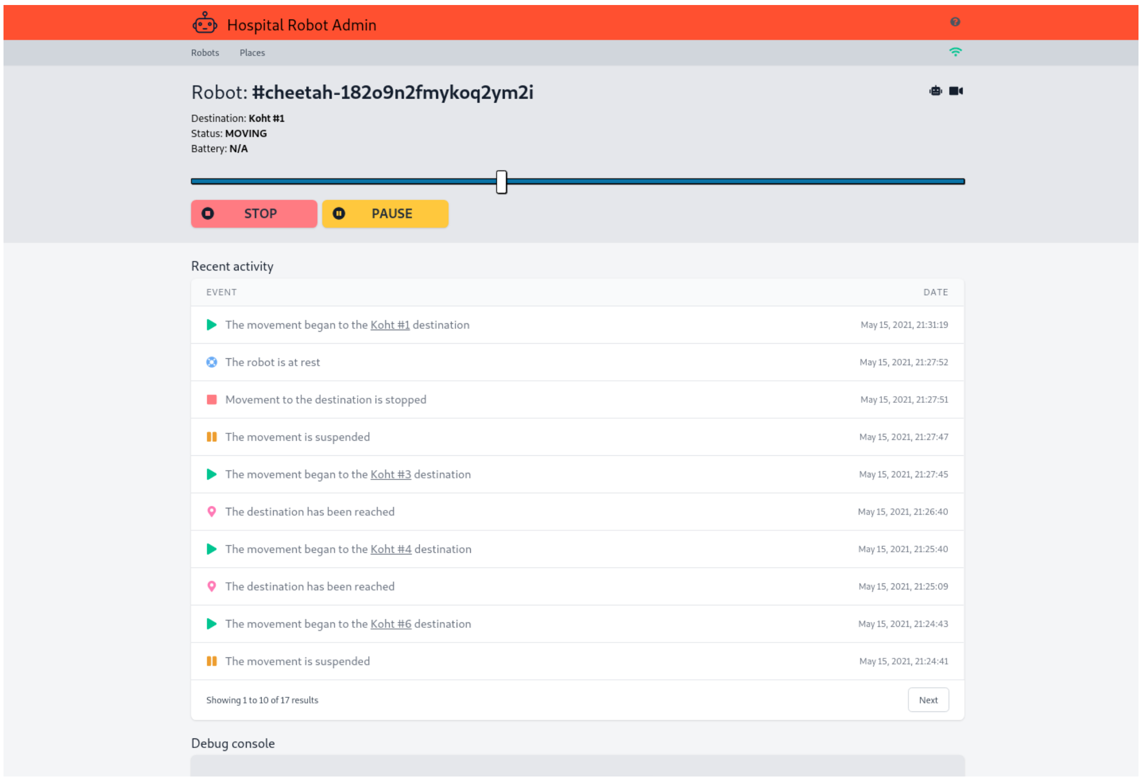
Robot ID: dolphin-182o9n2fmykoq2ym...
Destination: N/A
Status: IDLE
Battery: N/A

Robot ID: dogfish-182o9n2fmykoq2ym...
Destination: N/A
Status: IDLE
Battery: N/A

Robot ID: opossum-182o9n2fmykoq2y...
Destination: Koht #15
Status: MOVING
Battery: N/A

Figure 16: All robots overview page

To get to the detailed page of the robot, you need to click on the id of the robot. For better memorization, animal names are used as a prefix for each robot id.



Robot: #cheetah-182o9n2fmykoq2ym2i
Destination: Koht #1
Status: MOVING
Battery: N/A

Recent activity

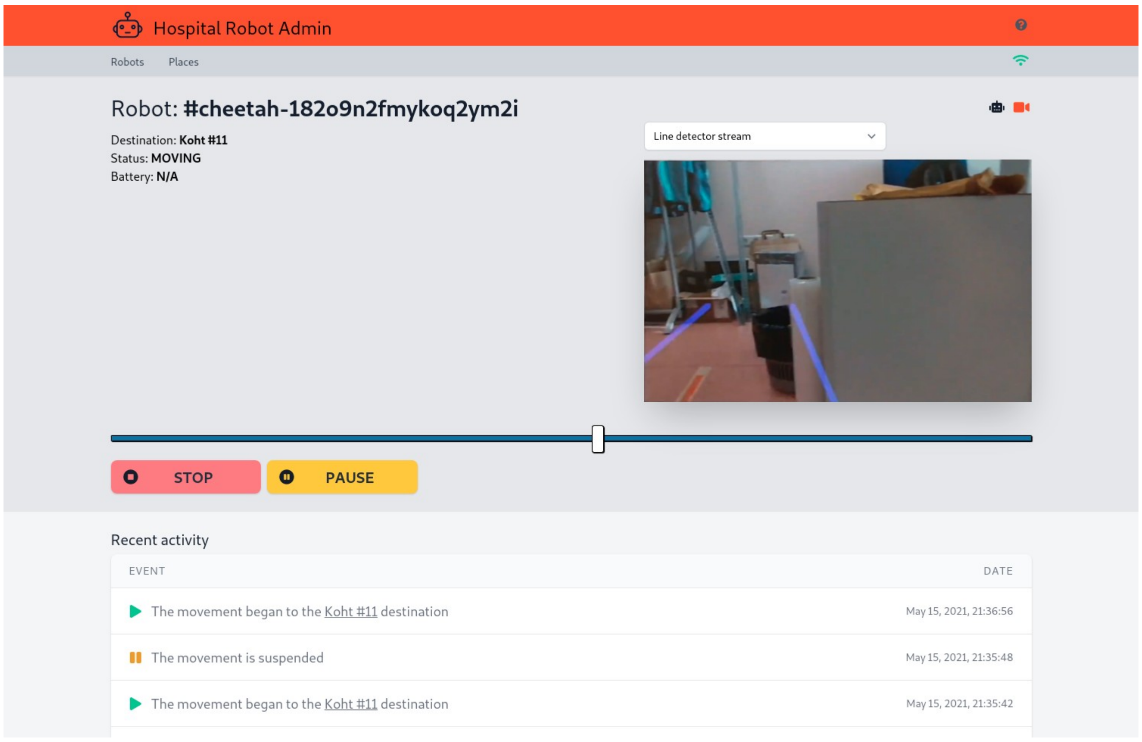
EVENT	DATE
▶ The movement began to the Koht #1 destination	May 15, 2021, 21:31:19
The robot is at rest	May 15, 2021, 21:27:52
■ Movement to the destination is stopped	May 15, 2021, 21:27:51
❚❚ The movement is suspended	May 15, 2021, 21:27:47
▶ The movement began to the Koht #3 destination	May 15, 2021, 21:27:45
The destination has been reached	May 15, 2021, 21:26:40
▶ The movement began to the Koht #4 destination	May 15, 2021, 21:25:40
The destination has been reached	May 15, 2021, 21:25:09
▶ The movement began to the Koht #6 destination	May 15, 2021, 21:24:43
❚❚ The movement is suspended	May 15, 2021, 21:24:41

Showing 1 to 10 of 17 results

Debug console

Figure 17: Robot detail page

The detailed page of the robot, which can be seen in Figure 17, contains a table with events. Clicking on the camera-shaped icon displays a streaming image with the option to select the desired image stream above (Figure 18).



Robot: **#cheetah-182o9n2fmykoq2ym2i**

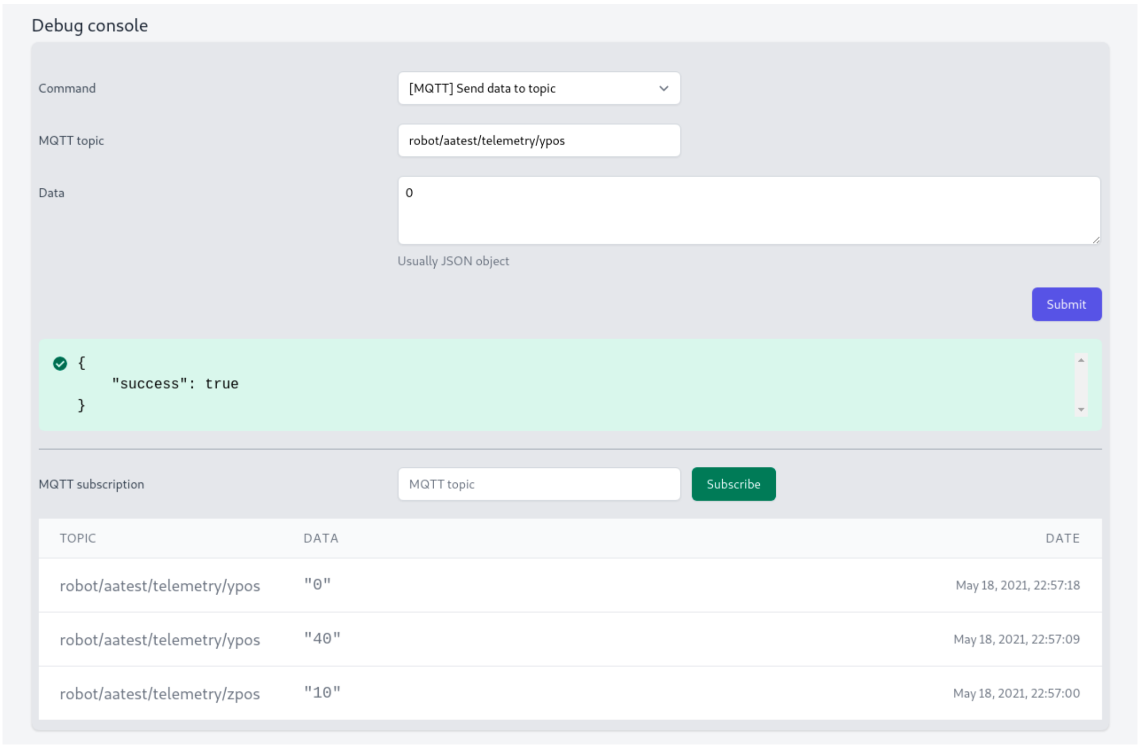
Destination: **Koht #11**
Status: **MOVING**
Battery: **N/A**

Recent activity

EVENT	DATE
The movement began to the Koht #11 destination	May 15, 2021, 21:36:56
The movement is suspended	May 15, 2021, 21:35:48
The movement began to the Koht #11 destination	May 15, 2021, 21:35:42

Figure 18: Robot detail page with image stream

The debug console is located at the bottom of the page. With its help, it becomes possible to send requests to the backend and directly to MQTT. It is also possible to subscribe to the desired MQTT topics and see events in real time. All of this can be done directly from the web page, which makes debugging very easy, see Figure 19.


Debug console

Command	[MQTT] Send data to topic
MQTT topic	robot/aatest/telemetry/ypos
Data	0

Usually JSON object

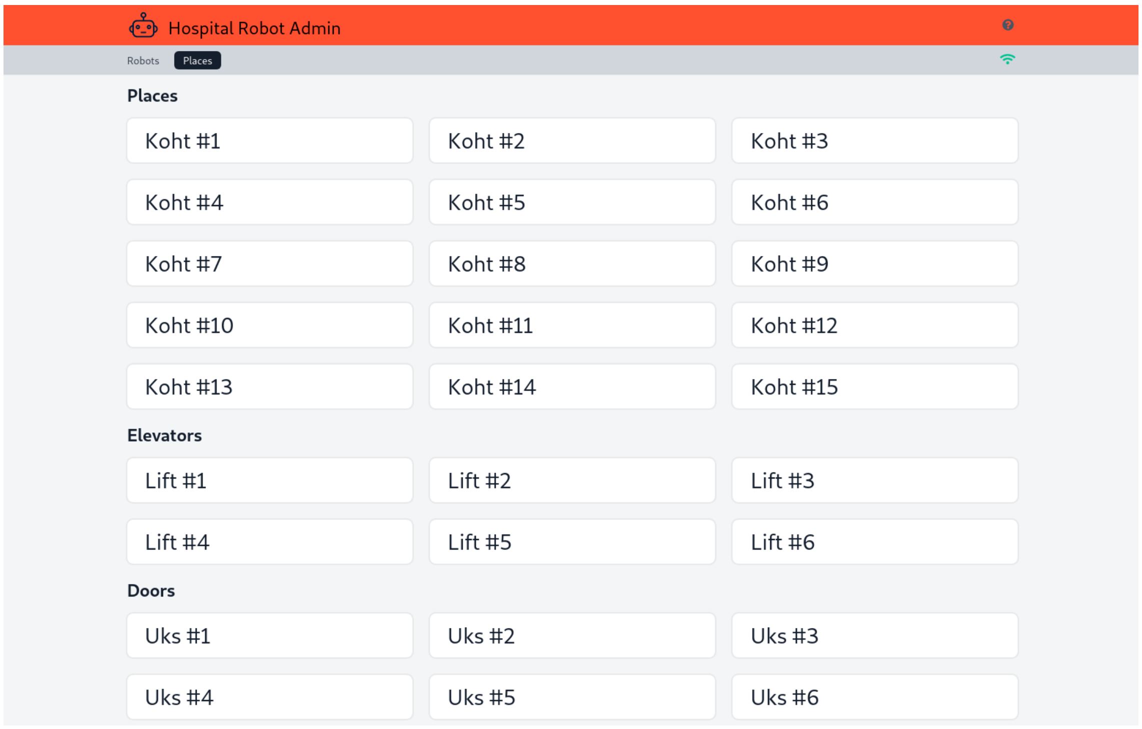
```
{
    "success": true
}
```

MQTT subscription

TOPIC	DATA	DATE
robot/aatest/telemetry/ypos	"0"	May 18, 2021, 22:57:18
robot/aatest/telemetry/ypos	"40"	May 18, 2021, 22:57:09
robot/aatest/telemetry/zpos	"10"	May 18, 2021, 22:57:00

Figure 19: Debug console on the robot detail page

By clicking on *Places* in the navigation menu, you will be taken to the infrastructure page. Figure 20 shows that there are main objects that robots interact with, divided into 3 main groups: places, elevators and doors.



Places

Koht #1	Koht #2	Koht #3
Koht #4	Koht #5	Koht #6
Koht #7	Koht #8	Koht #9
Koht #10	Koht #11	Koht #12
Koht #13	Koht #14	Koht #15

Elevators

Lift #1	Lift #2	Lift #3
Lift #4	Lift #5	Lift #6

Doors

Uks #1	Uks #2	Uks #3
Uks #4	Uks #5	Uks #6

Figure 20: Infrastructure overview screen

4.3.6 Deployment

As a containerization platform Docker was used. Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. Docker can package an application and its dependencies in a virtual container that can run on any Linux, Windows, or macOS computer [19] .

A multi-stage build was used to create a docker image with a minimal size. For builder stage node:slim image was used and node:alpine for production. This approach reduced the final size by 50% compared to using only the node:alpine image. Inside the docker container, the PM2 package was globally installed, which was used as a production process manager. PM2 is a daemon process manager that will help you manage and keep your application online 24/7. To put everything together, and indeed to keep all containers in one configuration file, docker-compose.yml was used.

A reverse proxy was required for traffic routing and HTTPS availability. Nginx [20] was the best choice for this. It is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. For debugging and quick database management, used phpMyAdmin on /pma endpoint with HTTP authentication for more security. How proxying is made can be seen in Figure 21.

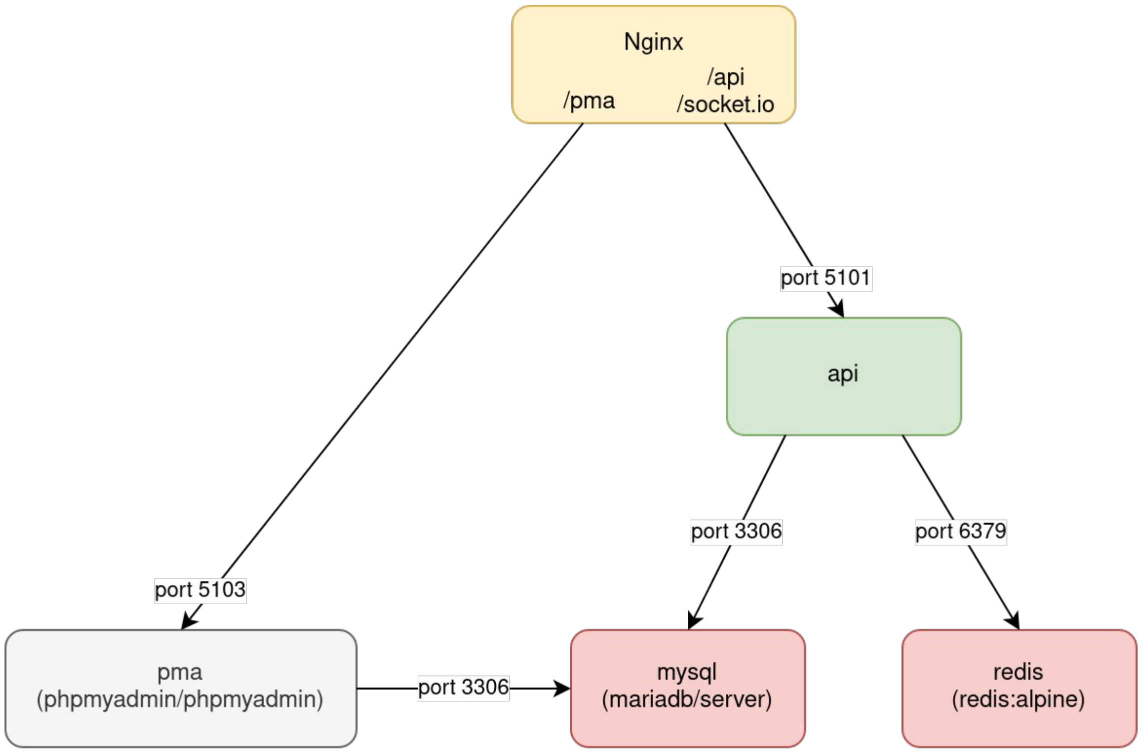


Figure 21: Production microservice arhitecture

All application file paths are configurable via environment variables:

- **CONTROLLERS** – controller files
- **WS_CONTROLLERS** – WebSocket controller files
- **MIDDLEWARES** – middleware files
- **WS_MIDDLEWARES** – WebSocket middleware files
- **SUBSCRIBERS** – subscriber files
- **TYPEORM_MIGRATIONS** – migration files
- **TYPEORM_MIGRATIONS_DIR** – migrations directory
- **TYPEORM_ENTITIES** – database model files

The use of such configurations is necessary for a containerized application, since the path changes along with the file extension after compilation from Typescript to JavaScript.

Since the project uses a lot of auxiliary scripts for development, build, linting, configurations, cleaning, working with the database and docker, a solution was needed that allows to keep package.json maintainable. nps is a package that solves this problem by allowing to move scripts to a package-scripts.js file. And because this file is a JavaScript file, you can do a lot more with project scripts, an example of which can be seen in Appendix 3.

5 Conclusion

My main goal in this project was to create a user-friendly software for controlling and monitoring robots in the distributed system. The requirements that had to be achieved in this work are described in more detail in Section 2.4.

In the process of working on this project, quite a lot of things have been done. It all started with discussing and planning the interface of how the user would interact with this robot. After that, the initial Vue framework for creating interfaces was chosen and the very first prototype was made. Realizing that this framework is not quite suitable for our tasks, it was decided to use a more suitable one, specifically for *Web Components – Stencil*. More details about usage were described in Section 3.3. After the initial skeleton of the application had already been sketched, a discussion began on how to do this for use on a tablet attached to a robot. In view of the fact that the user should be able to use only one application, it was decided to create an native Android application by converting the existing code base using the *Capacitor*.

After the work with the frontend was completed, the more serious part of writing the backend began. It all started with a simple mock server in pure JavaScript. After discussing some details with the team, I started writing an application that would be more production ready. Since TypeScript was chosen as the language, the complexity of this task was slightly lower than it would be written in the same Java, for example. To build the backend application, *Controller-Service-Repository* architecture was chosen using dependency injection. The admin area was one of the last tricky things to tinker with, see Section 4.3.5. It was also an interesting experience developing an application with the ability to view streaming images, especially when it comes to a decentralized system. The final part was system-wide integration, creating Docker containers, and configuring deployments using bash scripts. For more information about the technologies in favor of which solutions were chosen, see Section 2.5.

In general, for my part, I can say that I am very fortunate to have had such an opportunity to take part in such an interesting project and work in such a glorious team. I hope everything will work out for us and the project will be useful in the PERH hospital, alleviating the workload on the staff and reducing the risk of carrying any infections.

5.1 The future of the project

Since this work described the project only at the initial stage of development, there is certainly still something to work on and what to add / improve. First of all, I would wish to improve UX by adding dynamism when moving the robot, adding a more understandable status for the user that the robot is in motion. Next, the admin part should be completely cut out of the Android application so that there would be no mention of it. This, firstly, would reduce the size of the application, and secondly, it will make it significantly harder to inspect the device of the admin panel through reverse engineering.

As for what could be added, there are many options. A screen with the ability to administer destinations could be added to the admin panel. It would also be a great opportunity to see these places on the map. The next thing that would be a pretty good feature for understanding the state of the robot is showing intermediate points on the movement slider above the control buttons. This would make it possible to immediately fully see the entire route of the robot's movement.

Another interesting point is to start using a more convenient and at the same time secure authentication system. For tablets attached to the robot, it would be possible to add an employee card reader, so that by swiping this card, it would be easy to authorize employees who have the right to use this robot.

References

[1] Tanenbaum, Andrew S.; Steen, Maarten van. Distributed systems: principles and paradigms. Upper Saddle River, NJ: Pearson Prentice Hall, 2002.

[2] Magnoni, L. "Modern Messaging for Distributed Systems (sic)". Journal of Physics: Conference Series, 2015.

[3] "ROS-Introduction" [Online]. Available: http://wiki.ros.org/ROS/Introduction. [Accessed 12 05 2021].

[4] "ESP32 Wi-Fi & Bluetooth MCU I Espressif Systems" [Online]. Available: https://www.espressif.com/en/products/socs/esp32. [Accessed 18 05 2021].

[5] "Stencil - A Compiler for Web Components" [Online]. Available: https://stenciljs.com/docs/introduction. [Accessed 14 04 2021].

[6] "Tailwind CSS - Rapidly build modern websites without ever leaving your HTML." [Online]. Available: https://tailwindcss.com. [Accessed 18 05 2021].

[7] "Introduction — Vue.js" [Online]. Available: https://vuejs.org/v2/guide. [Accessed 18 05 2021].

[8] "Web Components" [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components. [Accessed 14 04 2021].

[9] "MQTT Version 5.0" [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. [Accessed 14 04 2021].

[10] "mosquitto.conf man page" [Online]. Available: https://mosquitto.org/man/mosquitto-conf-5.html. [Accessed 18 05 2021].

[11] "Shadow DOM" [Online]. Available: https://javascript.info/shadow-dom. [Accessed 18 05 2021].

[12] "What is COSU?" [Online]. Available: https://deviq.io/resources/articles/what-is-cosu. [Accessed 18 05 2021].

[13] "Dependency Injection" [Online]. Available: https://www.tutorialsteacher.com/ioc/dependency-injection. [Accessed 18 05 2021].

[14] "rfc4648" [Online]. Available: https://www.hjp.at/doc/rfc/rfc6455.html. [Accessed 18 05 2021].

[15] "Redis" [Online]. Available: https://redis.io. [Accessed 18 05 2021].

[16] "Depth Camera D435 — Intel® RealSense™ Depth and Tracking Cameras" [Online]. Available: https://www.intelrealsense.com/depth-camera-d435. [Accessed 18 05 2021].

[17] "nodelet" [Online]. Available: http://wiki.ros.org/nodelet. [Accessed 12 05 2021].

[18] "Redis" [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4648#section-4. [Accessed 18 05 2021].

[19] "Docker (software)" [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software). [Accessed 12 05 2021].

[20] "Welcome to NGINX Wiki!" [Online]. Available: https://www.nginx.com/resources/wiki [Accessed 18 05 2021].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Maksim Maksimov

1 Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Development of control and monitoring software for robots operating in a distributed system", supervised by Martin Rebane

   1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

   1.2 to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2 I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3 I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Source code of app-root component

```
import { Component, Element, h } from '@stencil/core';

@Component({
  tag: 'app-root',
  styleUrl: 'app-root.less',
  shadow: true,
})
export class AppRoot {
  @Element() el: HTMLElement;
  titleSuffix = ' - Hospital Robot Admin';

  render() {
    return (
      <main>
        <stencil-router>
          <stencil-route-switch scrollTopOffset={0}>
            <stencil-route url={['/', '/robot-ui']} component="app-robot" exact={true} />
            <stencil-route
              url="/admin"
              routeRender={() => (
                <app-admin>
                  <stencil-route url="/admin/robots/:robotId" exact={true} routeRender={data =>
<hroba-robot {...data} titleSuffix={this.titleSuffix} />} />
                  <stencil-route
                    url="/admin/robots"
                    exact={true}
                    routeRender={() => (
                      <hroba-robot-list>
                        <stencil-route-title pageTitle={`Robots${this.titleSuffix}`} />
                      </hroba-robot-list>
                    )}
                  />
                  <stencil-route
                    url="/admin/places"
                    exact={true}
                    routeRender={() => (
                      <hroba-place-list>
                        <stencil-route-title pageTitle={`Places${this.titleSuffix}`} />
                      </hroba-place-list>
                    )}
                  />
                </app-admin>
              )}
            />
          </stencil-route-switch>
        </stencil-router>
      </main>
    );
  }
}
```

Appendix 3 – Snippet of code from package.scripts.js file

```
const { series } = require('nps-utils');

function runFast(path) {
   return `ts-node -T -r tsconfig-paths/register ${path}`;
}

module.exports = {
   scripts: {
      ...
      /**
       * Building app into dist directory
       */
      build: {
         default: {
            script: series(
               'nps config',
               'nps lint',
               'nps clean.dist',
               'nps transpile',
               'nps copy.tmp',
               'nps clean.tmp',
            ),
            description: 'Builds the app into the dist directory'
         },

      },
      /**
       * Database scripts
       */
      db: {
         migrate: {
            script: runFast('./node_modules/typeorm/cli.js migration:run'),
            description: 'Migrates the database to newest version available'
         },
         seed: {
            script: runFast('./node_modules/typeorm-seeding/dist/cli.js seed'),
            description: 'Seeds generated records into the database'
         },
         drop: {
            script: runFast('./node_modules/typeorm/cli.js schema:drop'),
            description: 'Drops the schema of the database'
         },
         setup: {
            script: series(
               'nps db.drop',
               'nps db.migrate',
               'nps db.seed'
            ),
            description: 'Recreates the database with seeded data'
         }
      },
      ...
   },
};
```