



## Assignment of master's thesis

<b>Title:</b>	Progressive Web Application based on Microservice Architecture for monitoring of Babyboxes
<b>Student:</b>	Bc. Zbyněk Juřica
<b>Supervisor:</b>	Ing. Milan Dojčinovski, Ph.D.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Web Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2024/2025

### Instructions

Babybox Dashboard is an internal web application designed for remote monitoring and management of Babyboxes, further ensuring their safe and efficient operation across Czechia.

The goal of this work is to rewrite this outdated and hard to maintain application, transforming it into a more performant, manageable and future-ready system. The application is inspired by the previous version and will offer data visualizations, aggregations and implement further functionality for custom notifications and storing relevant data regarding each Babybox.

Main objectives:

- Analysis: assess use-cases and requirements and identify relevant technologies.
- Design: design a maintainable and modular microservice solution based on the previous monolithic application.
- Implementation: develop the application with a strong focus on data storage, visualization, notifications with a Progressive Web Apps (PWA) frontend solution.
- Testing and evaluation: test the application, evaluate its functionalities and performance compared to the older version.

Master's thesis

**PROGRESSIVE WEB  
APPLICATION BASED  
ON MICROSERVICE  
ARCHITECTURE FOR  
MONITORING OF  
BABYBOXES**

**Bc. Zbyněk Juřica**

Faculty of Information Technology  
Department of Software Engineering  
Supervisor: Ing. Milan Dojčinovski, Ph.D.  
May 9, 2024

Czech Technical University in Prague  
Faculty of Information Technology

© 2024 Bc. Zbyněk Juřica. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Juřica Zbyněk. *Progressive Web Application based on Microservice Architecture for monitoring of Babyboxes*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

## Contents

Acknowledgments	x
Declaration	xi
Abstract	xii
List of abbreviations	xiv
Introduction	1
<b>1 Background and Related Work</b>	<b>3</b>
1.1 Context	3
1.1.1 Are Babyboxes Necessary?	4
1.1.2 Technological Description	5
1.2 Related Work	7
1.2.1 Similar Solutions for Monitoring Applications	7
1.2.2 Relevant Alternatives	10
1.2.2.1 Grafana	10
1.2.2.2 TICK Stack	11
1.3 Technological Concepts	11
1.3.1 Client/Server	12
1.3.2 Microservice Architecture	12
1.3.3 Message Broker	13
1.3.3.1 Concepts	13
1.3.3.2 Strategies	14
1.3.4 API Gateway	15
1.3.5 REST API	15
1.3.5.1 Concepts	16
1.3.6 Server Side and Client Side Rendering	17
1.3.6.1 Server-Side Rendering	17
1.3.6.2 Client-Side Rendering	18
1.3.6.3 When to Choose Which?	18
1.3.7 PWA	19
1.3.7.1 Service Workers	19
1.3.7.2 Web App Manifest	19
1.3.7.3 HTTPS	19
1.3.8 SWR	19

1.3.9	Containers	20
1.3.9.1	Docker	20
1.3.9.2	Docker Compose	21
1.3.9.3	Benefits	22
1.3.10	NoSQL Databases	22
1.3.10.1	MongoDB	22
1.3.10.2	InfluxDB	23
1.3.11	JWT Authenticaiton Mechanism	24
<b>2</b>	<b>Analysis and Design</b>	<b>26</b>
2.1	Methodology	26
2.1.1	Software Development Process	26
2.1.2	Technological Freedom in Microservice Architecture	27
2.1.3	Guiding Principles	27
2.1.3.1	Design Principles and Patterns	28
2.2	Requirements Analysis	30
2.2.1	Identification and Analysis of User Requirements	31
2.2.2	User Personas	32
2.2.3	Use Cases	32
2.2.4	Functional Requirements	34
2.2.5	Non-functional Requirements	36
2.3	Choosing technologies for This Project	36
2.3.1	Programming Languages and Frameworks	36
2.3.1.1	Next.js	37
2.3.1.2	JavaScript Runtime	38
2.3.1.3	Go	38
2.3.1.4	Python	39
2.3.2	Web Frameworks	40
2.3.3	Charting library	41
2.3.4	Databases	43
2.3.4.1	Time-series Database	43
2.3.4.2	General-purpose Database	45
2.3.5	Middleware	46
2.3.5.1	Message Broker	46
2.3.5.2	API Gateway	48
2.3.5.3	Docker and Docker Compose	48
2.4	System Design	49
2.4.1	Previous Solution	50
2.4.2	Proposed Solution	52
2.4.2.1	Transition to a Microservice Architecture	53
2.4.2.2	Architectural Design	55
2.4.2.3	Design of Individual Components	56
2.4.2.4	Domain Model	58

<b>3</b>	<b>Implementation</b>	<b>61</b>
3.1	Implementation Process	61
3.2	Infrastructure	62
3.2.1	Code Structure	63
3.2.1.1	Nx	63
3.2.1.2	Current Setup	64
3.2.2	Containerization	64
3.2.2.1	Docker	65
3.2.2.2	Docker Compose	66
3.2.2.3	Environment Variable Management	66
3.2.2.4	Container Dependencies	67
3.2.3	API Gateway	67
3.2.3.1	Development Environment	67
3.2.3.2	Production Environment	68
3.2.4	RabbitMQ	69
3.2.4.1	Docker Compose Configuration	69
3.2.4.2	Naming Conventions	69
3.2.4.3	Communicating through the Message Broker	69
3.2.4.4	Types of Data	70
3.2.5	MongoDB	70
3.2.5.1	Docker Compose Configuration	71
3.2.5.2	Practical Utilization of Schemaless Design	71
3.2.5.3	Efficient Data Retrieval	71
3.2.5.4	Indexing Considerations	72
3.2.5.5	Tooling and Development Support	72
3.2.6	InfluxDB	72
3.2.6.1	Docker Compose Configuration	73
3.2.6.2	Data Storage	73
3.2.6.3	Development Tools and Visualization	73
3.3	Microservices	74
3.3.1	Snapshot Handler Microservice	74
3.3.1.1	New Firmware API Improvements	74
3.3.1.2	Slug Conversion and Usage	75
3.3.1.3	Data Handling and Storage	75
3.3.1.4	API Endpoints for Data Retrieval	75
3.3.1.5	API Versioning and Envelope Pattern	76
3.3.1.6	Authentication	76
3.3.1.7	Health Check Endpoint	77
3.3.2	Querying Data from InfluxDB	77
3.3.2.1	Gap Filling Strategies	79
3.3.3	Babybox Microservice	81
3.3.3.1	Core Functionality	81
3.3.3.2	API Design and Endpoint Functionality	82
3.3.4	Notification Microservice	83

3.3.4.1	REST API . . . . .	83
3.3.4.2	Template Configuration and Snapshot Processing	84
3.3.4.3	Strategies to Refining Notification Frequency .	84
3.3.4.4	Chain of Checkers . . . . .	84
3.3.4.5	Email Notification Mechanism . . . . .	87
3.3.5	User Microservice . . . . .	88
3.3.5.1	REST API . . . . .	88
3.3.5.2	Registration Process . . . . .	88
3.3.5.3	Login Process . . . . .	89
3.3.5.4	Integration with Other Services . . . . .	90
3.3.6	Battery Analyzer Microservice . . . . .	92
3.3.6.1	REST API . . . . .	92
3.3.6.2	Taking a Measurement . . . . .	93
3.3.6.3	Battery Quality Assessment Idea . . . . .	93
3.3.6.4	Empirical Research and Methodology . . . . .	94
3.4	Front-end . . . . .	95
3.4.1	Application Flow . . . . .	96
3.4.2	User Experience . . . . .	97
3.4.3	Implementing PWA . . . . .	98
3.4.4	Login and Logout Functionality . . . . .	99
3.4.5	Starting Page . . . . .	100
3.4.5.1	Data Tables . . . . .	102
3.4.6	Babybox Page . . . . .	103
3.4.6.1	Grouping notification . . . . .	105
3.4.7	Chart Page . . . . .	106
3.4.7.1	Chart Component . . . . .	106
3.4.7.2	Consistent Color Coding . . . . .	108
3.4.7.3	SearchParams as State . . . . .	108
3.4.7.4	ApexCharts Performance . . . . .	109
3.4.7.5	Displaying Event Data . . . . .	110
3.4.7.6	Gap Filling Algorithms . . . . .	112
3.4.8	Battery Analysis Page . . . . .	113
3.4.8.1	Babybox Information Page . . . . .	113
3.4.9	Notification and User Pages . . . . .	114
<b>4</b>	<b>Testing and Evaluation</b>	<b>115</b>
4.1	Testing . . . . .	115
4.1.1	Unit testing . . . . .	115
4.1.2	Testing the API interface . . . . .	116
4.2	CI/CD Pipelines . . . . .	118
4.2.1	Continuous Integration . . . . .	118
4.2.1.1	Additional Workflows . . . . .	119
4.2.2	Continuous Deployment . . . . .	119
4.3	Documentation . . . . .	120

4.4	Evaluation and User Feedback . . . . .	120
4.4.1	Requirements Fulfillment . . . . .	120
4.4.2	User Feedback . . . . .	122
<b>5</b>	<b>Conclusion</b>	<b>123</b>
5.1	Future Improvements . . . . .	124
5.1.1	Kubernetes . . . . .	124
5.1.2	Analysis Services . . . . .	124
5.1.3	Maintenance Microservice . . . . .	125
5.1.4	Improving Current Components and Services . . . . .	125
<b>A</b>	<b>Previous Solution Screenshots</b>	<b>127</b>
A.1	Babybox Detail Page . . . . .	127
A.2	Babybox Page . . . . .	128
A.3	Time Filtering on Chart Page . . . . .	129
<b>B</b>	<b>New Solution Screenshots</b>	<b>130</b>
B.1	Babybox Detail Page . . . . .	130
B.2	Babybox Table with Tooltips . . . . .	131
B.3	Time Filtering on Chart Page . . . . .	132
B.4	Statistics and Tabular Data under the Chart . . . . .	133
	<b>Contents of the Attached Media</b>	<b>139</b>



## List of Figures

2.1	Performance benchmarks of the web frameworks we considered. Showing the average latency, P90 and P99 values for 64, 256 and 512 concurrency. . . . .	40
2.2	Performance benchmarks of the same web frameworks showing a score of performance.[53] . . . . .	41
2.3	Example of range, x-axis, y-axis and point annotations in a line chart.[54] . . . . .	42
2.4	Benchmark results showing ingestion speeds (top image), query speeds (left image) and aggregation speeds (right image). . . .	44
2.5	Data flow diagram of the previous solution, where the blue Backend component is the one big monolith. . . . .	51
2.6	Data flow diagram of the new microservice solution. . . . .	55
2.7	Domain model . . . . .	58
3.1	Visualization of distributing snapshots among consumers. . . .	70
3.2	Visualization of the filling algorithms. . . . .	80
3.3	Visualization of the notification checking chain. . . . .	85
3.4	Visualization of the notification checking in action (streak set to 2, new error set to <code>true</code> , delay set to 35 minutes). . . . .	86
3.5	Normalized battery voltage decrease over time (y-axis starts at 12.5V and ends at 13.8V; x-axis is from 0 seconds to 25 minutes). . . . .	95
3.6	Diagram showing the flow through the application. . . . .	96
3.7	Skeleton components indicating that the overview widgets are loading/fetching data. . . . .	97
3.8	Screenshots of the PWA application in standalone mode looking like a native application. . . . .	99
3.9	New table containing all the babyboxes on the starting page. . .	101
3.10	Table showing a list of all the babyboxes in the previous version of the application. . . . .	101
3.11	Partial screenshot of variable overview widgets showing the first 3 variables - chart and minimum, maximum and average statistics over the last week, 3 days and 1 day. . . . .	103
3.12	Screenshot of the table showing the latest snapshots. . . . .	104
3.13	Notifications displayed using the accordion component grouped by their <code>template_id</code> and further grouped by the day on which they occurred. . . . .	106

3.14	The main line chart displaying the temperatures over one week time. . . . .	107
3.15	The main chart displaying the color coded events with labels over one week of time. . . . .	111
3.16	Visualization of the intervals algorithm creating and then combining the intervals together. . . . .	112
3.17	User interface of the battery measurement. . . . .	113
4.1	Performance testing in Postman. . . . .	121

## List of code listings

3.1	Pseudo-Dockerfile showcasing the basic structure of the Dockerfiles in this application. . . . .	65
3.2	Configuration a service through environment variables in a Docker Compose file. . . . .	66
3.3	Configuring snapshot-handler service for Traefik using labels. . . . .	68
3.4	Configuring snapshot-handler service in Caddyfile. . . . .	68
3.5	Query template for a snapshot range query based on slug. . . . .	77
3.6	Query creation for an aggregated query. . . . .	78
3.7	Using <code>aggregateWindow</code> for filling gaps in time-series data. . . . .	80
3.8	Using a pre-hook in Mongoose for hashing passwords before saving to MongoDB. . . . .	89
3.9	Generating a new JWT in Bun. . . . .	89
3.10	Protecting a group of endpoints in Go's Echo by checking the JWT validity. . . . .	90
3.11	Protecting endpoints in Python's FastAPI by checking the JWT validity. . . . .	91
3.12	Protecting a group of endpoints in Bun's Elysia by checking the JWT validity. . . . .	92
3.13	Adjusting CSS styling for PWA users only. . . . .	98
3.14	Structure of the auth-context that can be used throughout the application. . . . .	100
3.15	Using the <code>DataTable</code> component. . . . .	102
3.16	Column definition (for the temperature inside column) to provide icons indicating a sudden change in the data. . . . .	104
3.17	CSS and Tailwind configurations working together to add custom colors. . . . .	108
3.18	Updateing search <code>from</code> and <code>to</code> parameters in Next.js. . . . .	109

3.19	Code for delaying the render of the line chart component. . . .	110
4.1	Example of a test suite for the notification pipeline of grouping them by template ID, day of year and then merging together. .	116
4.2	Examples of API testing using Postman to check the status code, headers and data payload. . . . .	117
4.3	Snippet of the Continuous Integrity part of the pipeline for running automated tests. . . . .	118
4.4	Snippet of the Continuous Delivery part of the pipeline for automatically deploying to a remote server. . . . .	119

*I would like to express my deepest and warmest gratitude to my supervisor for his invaluable guidance, and support that he provided, which carried me through all the stages of writing this thesis. His insights and expertise were instrumental in shaping this thesis.*

*I am also immensely grateful to my family for their unwavering support, encouragement, and advice. To my girlfriend, thank you for your patience, understanding, and constant motivation, it has meant a lot to me.*

*Finally, a special thanks to my dog, an old lady who watched me tirelessly work on this thesis. Your quiet companionship made the long hours much more bearable.*

*Thank you all for being my pillars of support during this endeavor.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2024

## Abstract

This thesis presents the design and implementation of a monitoring system based on microservice architecture for managing and analyzing data from babyboxes across the Czech Republic. The work involved transitioning from an outdated monolithic architecture to a more flexible and maintainable microservices architecture, aiming to empower staff working as operators and maintenance technicians.

The system includes several microservices handling data ingestion, user management, notifications, and battery analysis. Built using Go, TypeScript, Python, MongoDB, InfluxDB, and RabbitMQ, the backend provides a scalable and modular structure. The front-end, developed with Next.js and React, offers comprehensive data visualization, aggregations, notifications, and analysis features. The application was continuously improved based on user feedback, laying a strong foundation for future enhancements and integrations.

**Keywords** Microservice architecture, Progressive Web Application, Babybox, Baby Hatch, Internet of Things, Go, Next.js, TypeScript, Bun, InfluxDB, MongoDB, Docker, Monitoring System

## Abstrakt

Tato diplomová práce představuje návrh a implementaci monitorovacího systému založeného na architektuře mikroslužeb pro správu a analýzu dat z babyboxů po celé České republice. Práce zahrnovala přechod ze zastaralé monolitické architektury na flexibilnější a lépe udržitelnou architekturu mikroslužeb s cílem poskytnout lepší nástroje pracovníkům, kteří působí jako operátoři a servisní technici.

Systém zahrnuje několik mikroservis, které zajišťují příjem dat, správu uživatelů, notifikace a analýzu stavu akumulátorů. Back-end je vytvořen pomocí technologií Go, TypeScript, Python, MongoDB, InfluxDB a RabbitMQ a poskytuje škálovatelnou a modulární strukturu. Frontend, vyvinutý pomocí

Next.js a React, nabízí komplexní vizualizaci dat, agregace, notifikace a analytické funkce. Aplikace byla průběžně vylepšována na základě zpětné vazby uživatelů, což položilo pevný základ pro budoucí vylepšení a integrace.

**Klíčová slova** Architektura mikroslužeb, Progresivní Webová Aplikace, Babybox, Baby Hatch, Internet Věcí, Go, Next.js, TypeScript, Bun, InfluxDB, MongoDB, Docker, Monitorovací Systém

## List of abbreviations

API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
CSR	Client-Side Rendering
DB	Database
DRY	Don't Repeat Yourself
GSM	Global System for Mobile communications
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
IP	Internet Protocol
JWT	JSON Web Token
JSON	JavaScript Object Notation
MQ	Message Queue
MQTT	Message Queuing Telemetry Transport
NoSQL	Not Only SQL / No SQL
PDF	Portable Document Format
PWA	Progressive Web Application
REST	Representational State Transfer
SEO	Search Engine Optimization
SMS	Short Message Service
SQL	Structured Query Language
SSR	Server-Side Rendering
SWR	Stale-While-Revalidate
TDD	Test Driven Development
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language



# Introduction

Babyboxes, also known as baby hatches, provide a secure and anonymous method for individuals to leave their newborns in a location where they will be promptly discovered and cared for. This concept, deeply rooted in ancient history, has been prevalent for centuries.

The inception of babyboxes dates back to the 12th century in Italy. Strategically placed in accessible locations, often near churches or monasteries, these facilities offered a safe alternative for individuals unable to care for their infants, ensuring the well-being of these children.[1]

Over time, this idea spread to other countries and has now reached different parts of the world. Today they can be found in many different countries including Germany, Poland, Italy, and even countries like Japan, Malaysia, and places like Vancouver. Despite regional variations, the core objective of these babyboxes remains consistent: ensuring the safety of infants.[1]

The enduring relevance of babyboxes illustrates their importance and highlights the need for innovative solutions to enhance the safety and responsiveness of these devices. This is especially evident in the Czech Republic, where there is a notable number of these devices further enhanced by modern technologies featuring mechanisms to ensure improved safety but also availability with immediate multi-layer alerting.

Integral to the efficient operation of these babyboxes is Babybox Dashboard, an internal tool for operations staff to monitor the network of babyboxes. This application receives data, such as temperatures and voltages, from babyboxes periodically and enables its users to monitor the status and fix any potential issues.

The goal of this thesis is to rethink, improve, and rewrite the current application to support future extensibility and better maintainability while enhancing its scalability and performance. Inspiration will be drawn from the previous solution to identify and improve areas that were lacking, particularly those involving outdated technologies, performance issues affecting the front-end user experience, and notification fatigue caused by an excessive number of notifications generated. The implementation will leverage the microservice

architecture to build a progressive web application that provides an intuitive overview of the status of each babybox while accommodating this functionality with custom notifications and analysis.

This thesis will commence by [Chapter 1](#) explaining the background, describing similar works and also giving an overview of the relevant technologies. The following chapter, [Chapter 2](#), goes over the analysis, focusing on the preceding solution, followed by an outline of the new design and the transition from a monolithic to a microservice architecture. The main chapter is [Chapter 3](#), which describes the practical implementation of this work: the application infrastructure and details the components and services that constitute the entire application. Finally, [Chapter 4](#) will go over testing and evaluating the final solution. The [Conclusion](#) summarizes this work and its results and sets future goals and possible expansions of this application.

## Chapter 1

# Background and Related Work

*This chapter delves into the background and related work that provide essential context for this thesis. Initially, it introduces the concept of babyboxes, exploring their purpose and technical specifics, thereby setting the stage for the application’s focus. Subsequently, the discussion extends to similar projects in various domains and also examining alternative solutions and approaches. Additionally, this chapter explicates key technical concepts employed throughout this thesis. The comprehensive examination presented here is crucial for understanding the domain of operation and the rationale behind the decisions made during the project’s development phases.*

## 1.1 Context

The concept of “babyboxes” or “baby hatches”<sup>1</sup> provides a secure and anonymous method for parents, especially mothers, to safely leave their newborns. In the Czech Republic, these devices are equipped with temperature regulation features and are typically located near healthcare facilities to ensure immediate care for the infants. Spearheaded by the Nadační fond pro odložené děti Statim<sup>2</sup> and the registered association Babybox pro odložené děti–Statim<sup>3</sup>, the initiative has seen significant adoption across the country.

As of April 2024, 88 babyboxes have been installed in the Czech Republic, which have collectively safeguarded 253 children. These installations are mostly found in hospitals, with a few exceptions in other types of buildings,

---

<sup>1</sup>In this thesis, we will use the term “babybox” instead of “baby hatch”, as that is the name commonly recognized in the Czech Republic.

<sup>2</sup><https://www.babybox.cz/?p=fond>

<sup>3</sup><https://www.babybox.cz/?p=sdruzeni>

reflecting their integration into healthcare and community infrastructure. The design of babyboxes has evolved over time to where it is today: Babyboxes are made of stainless steel with automated doors and safety features like infrared sensors to prevent accidents. [2][3]

The initiative's growth in the Czech Republic has been driven by key figures such as Ludvík Hess, with support from various sponsors and medical professionals. Despite some legislative and administrative challenges, the commitment to expanding this service highlights its importance.[2][3]

### 1.1.1 Are Babyboxes Necessary?

In examining the role and necessity of baby hatches in our society, we looked at the insights provided by research from Japan, Malaysia, and Poland, each offering a unique perspective on this complex issue.

The Japanese perspective, as discussed in the first paper, emphasizes that baby hatches serve as crucial intervention points for infants who might otherwise be abandoned in dangerous situations. This analysis supports the view that while baby hatches do not address the underlying social challenges that lead to abandonment, they fulfill an essential role by offering a safe and anonymous option for parents in crisis. The paper advocates for the preservation of these hatches as they ensure the survival and safety of infants, presenting a pragmatic solution to a distressing social dilemma.[4]

The Malaysian study shifts the focus slightly to the societal implications of baby hatches, particularly in contexts where there is significant social stigma against unwed mothers and victims of sexual assault. It outlines how baby hatches in Malaysia act as compassionate alternatives, providing a secure environment for abandoned infants while maintaining the anonymity of the parents. This approach helps to mitigate infant mortality and supports mothers in desperate situations without subjecting them to further societal judgment or exclusion.[5]

Poland's case, explored in the third paper, delves into the ethical, cultural, and societal debates surrounding baby hatches. It critically examines the tension between the child's right to know their origins and the societal benefits of providing a secure abandonment point. Despite various ethical concerns, the conclusion drawn is that baby hatches, albeit controversial, are necessary. They provide an indispensable safety option for individuals facing extreme circumstances, thus serving as a vital last-resort measure in safeguarding the welfare of children.[6]

Bringing these analyses together, it is clear that baby hatches, regardless of the cultural or ethical controversies they may spark, serve as indispensable safety measures. They are established not as solutions for the root cause but as necessary interventions to ensure the immediate safety of infants. Each paper, while recognizing that baby hatches do not tackle the root causes of abandonment, reiterates their importance in providing a safe alternative for

ensuring the survival and well-being of infants when parents find themselves in untenable situations.

In conclusion, the necessity of baby hatches as a last-resort measure is evident across different social and cultural settings. They provide a critical safety net for newborns, ensuring that infants have a safe place to go when all other options might fail. This makes them a crucial part of social safety measures, needed to protect the most vulnerable in society—the newborn infants.[4][5][6]

### 1.1.2 Technological Description

Bayboxes, for which we will be developing the platform, are engineered with technological components designed to ensure safety and security for the infant while facilitating easy and discreet operation for the parent. As we are the ones manufacturing them, we have a direct influence on which hardware and software is going to be used. Babyboxes are usually built into a wall of the hospital, leading to a separation to the outside side, accessible publicly by the individual leaving the child, and the hospital side, where the medical staff retrieves the child. Each babybox is composed of several key components:

#### 1. Engine Unit:

- **Functionality:** The engine unit controls the automatic doors on the outside side of the babybox. It integrates sensors to detect the weight of the object placed inside, activating the doors to open or close automatically. This unit is designed to operate seamlessly to ensure that the door mechanism does not pose any risk during its operation.

#### 2. Thermal Unit:

- **Heating and Cooling:** The thermal unit manages the internal environment of the babybox through two types of heating mechanisms—casing heating and fan heating—to maintain a stable temperature regardless of external weather conditions. Cooling is facilitated through fans, which are activated as needed to keep the internal temperature within a safe range.
- **Temperature Monitoring:** Multiple sensors are deployed to monitor various temperatures:
- **Inside Temperature:** Ensures the internal environment remains within a safe range for the infant.
- **Outside Temperature:** Monitors external conditions to adjust internal heating or cooling.
- **Casing Temperature:** Oversees the temperature of the babybox casing heating.

- Top and Bottom Temperatures: These are specifically placed on the top and bottom of the heat exchanger to monitor the state and functioning of the heating/cooling system.
- Voltage Monitoring: The unit monitors various voltages, most importantly the input voltage and the voltage of the battery in case of a power outage.

### 3. Monitoring and Communication:

- Camera: A camera installed inside the babybox provides real-time visual monitoring, allowing hospital staff to observe the babybox interior continuously. This is important for ensuring the safety and security of the child until retrieval.
- Router: Connects the babybox to the hospital's network, enabling the transmission of real-time data and video feeds to the monitoring computer accessed by hospital staff. The router also facilitates an internet connection for external communications.
- GSM Communicator: An additional layer of communication is provided by a GSM communicator, which sends SMS notifications to predefined recipients, enhancing the alert system in case of emergencies.

### 4. Connectivity and Alerts:

- Monitoring Panel: A computer installed in the hospital, in a place where staff is present 24/7, displaying real-time data and video from the babybox as well as alerting the staff about problems and activation.
- Emails: Sent by both the engine and thermal units, and the camera, to notify staff of events or problems.
- SMS Alerts: Generated by the GSM communicator, sending text messages to designated recipients about critical events.

The Babybox system employs a structured approach to data collection, which is essential for monitoring the operational integrity of each unit. The thermal unit within each Babybox is tasked with gathering data on temperatures and voltage measurements at regular intervals. Specifically, data is transmitted every 10 minutes (e.g., XX:00, XX:10, XX:20, etc.), ensuring that any significant changes in environmental conditions or system performance are promptly captured and addressed.

Data is sent to a designated HTTP endpoint: `GET /BB.{BABYBOX_NAME}.data?BB={BABYBOX_NAME}&T0=value&T1=value,...`, where `{BABYBOX_NAME}` is an identifier such as `BRNO` or `OSTRAVA`, and `T0` to `TX` represent readings from various sensors. This method of transmission has been a historical choice for the system, supporting current operational needs effectively. However, our goal is to update and enhance this data transmission process in future versions of the system.

Currently, the system does not automatically collect event data, which is seen as a significant area for improvement. Adding automatic event data collection would provide a more comprehensive overview of the Babybox's operational status and help pinpoint specific incidents requiring immediate attention.

Looking forward, we intend to reduce the data transmission interval to every 5 minutes to allow more frequent updates and closer monitoring. Alongside this change, the legacy endpoint format is under review to potentially incorporate more advanced data handling capabilities, which would improve efficiency and the overall functionality of Babybox Dashboard. These planned updates are part of a broader strategy to enhance the system's responsiveness and the safety of the infants cared for within the Babybox network.

## 1.2 Related Work

We looked at the landscape of existing solutions and technologies in this domain. This section explores various monitoring applications that utilize microservice architectures, providing insights into how similar systems are structured and operate. Additionally, we will look into established tools such as Grafana and the TICK stack, which are prominent in the fields of monitoring and analytics. Analyzing these solutions allows us to draw on industry experiences and identify effective practices that could be adapted to enhance our system. This review not only broadens our perspective on possible design and functional approaches but also ensures that our development is informed by a wide range of existing technological advancements.

### 1.2.1 Similar Solutions for Monitoring Applications

#### **A Microservices-based IoT Monitoring System to improve the Safety in Public Buildings**

This study introduces a microservices-based IoT monitoring system designed to enhance safety in public buildings by continuously tracking environmental conditions such as temperature, or smoke levels. The microservice architecture allows for a pattern-based specification of system components that are adapted to the specific context, such as user's needs and requirements.

The microservice architecture was chosen to best satisfy the non-functional requirements. The implementation demonstrates the effectiveness of this architecture for safety applications where system failure can have severe consequences. The microservice architecture was chosen to improve availability and system failure tolerance, performance, and scalability.[7]

#### **Design and implementation of a smart beehive and its monitoring**

**system using microservices in the context of IoT and open data**

The smart beehive monitoring system utilizes microservices to manage and analyze data collected from various sensors within beehives, focusing on parameters such as temperature and humidity which are critical for bee health. This system uses Python for microservices development, integrating with both wireless sensor networks (WSN) and IoT technology to gather and process environmental data, which they also provide for further research in the formats of XML, JSON, RDF, Turtle and other open data formats.

This approach highlights the benefits of microservices in managing complex data structures and ensuring the scalability of IoT applications. The study is satisfied with the results of the microservice architecture, however, more work needs to be done in properly handling and utilizing the data in a meaningful way.[8]

**A Microservices Platform for Monitoring and Analysis of IoT Traffic Data in Smart Cities**

The PROMENADE project developed primarily for real-time monitoring and analysis of traffic data from smart cities, employing a microservices architecture to handle the vast amounts of spatio-temporal IoT traffic data with fine resolutions. The platforms utilizes IoT/Fog/Cloud paradigms, microservice architecture and DevOps to run the application using OpenShift with Kubernetes and Apache ActiveMQ as a message broker along side with Kafka for receiving data.

The successful deployment of this system underscores the adaptability and efficiency of microservices in handling real-time data and large-scale applications within smart cities. The use of microservices enables the system to develop robust and reliable services and to improve the ability of satisfying strict non-functional requirements.[9]

**Design of a Cattle-Health-Monitoring System Using Microservices and IoT Devices**

In the domain of agriculture, a microservices-based system was developed to monitor cattle health by collecting data on various physiological and environmental parameters. The system efficiently manages data across a distributed network, facilitating scalability at the service level and robust data processing capabilities. This enables farmers to receive real-time insights into the health status of their cattle, helping them make informed decisions about animal care and management.

The study demonstrates the practical applications of microservices in agricultural settings, where the ability to process large volumes of data reliably is essential for effective livestock management. The nature of microservices allow for better fault tolerance. This research highlights the flexibility of microservices in enhancing data-driven decision-making in agriculture, in the context



of using machine learning techniques for further analysis.[10]

### **Microservices-based IoT Monitoring Application with a Domain-driven Design Approach**

This paper discusses an IoT-based environmental monitoring system within a Domain-driven Design (DDD) framework, using microservices to manage complex interactions between various data points and services. By employing microservices, the services are more independent during the development process and their performance does not affect each other that much; the deployment process has also been more effective.

The use of DDD in conjunction with microservices architecture goes hand-in-hand to produce adaptive software to changes during the development process. In the future, the authors would like to explore the usage of MQTT protocol more for improved independence and greater scalability.[11]

### **Multiple time series database on microservice architecture for IoT based sleep monitoring system**

The paper explores an IoT-based sleep monitoring system utilizing a microservices architecture to manage and analyze sleep data captured through wearable devices. This system employs multiple time series databases to store and process large volumes of health-related data efficiently, ensuring high throughput and minimal latency. The use of the MQTT protocol facilitates the real-time transmission of data. The microservices allow for the modular integration of various components, each handling specific functionalities such as data ingestion, processing, and analysis.

The conclusion drawn from this study emphasizes that microservices architecture is better for developing scalable, reliable, and flexible healthcare monitoring systems that can adapt to increasing data volumes and complex processing needs. Introducing InfluxDB for data ingestion has increased write performance by about 20 times faster. Using MQTT as the sensor gateway increased throughput by about 2 times. Using a separate database for each service has further increased scalability, resilience, and independence.[12]

### **Summary**

The related work provides inspiration not only through the technologies commonly employed in this space, such as message brokers (using MQTT or Kafka) and containerization tools (Docker or Kubernetes), but also by highlighting the advantages of flexibility, fault tolerance, independence, and improved availability, performance, scalability, and maintainability.

## 1.2.2 Relevant Alternatives

We have also reviewed and rejected some other solutions that could have been potentially easier to use, but do not actually fully satisfy the needs and requirements of our application.

### 1.2.2.1 Grafana

Grafana is an open-source platform renowned for its robust capabilities in monitoring and visualizing metrics from a vast array of data sources such as InfluxDB, Prometheus, and Elasticsearch. It enables users to create detailed, interactive dashboards that showcase data through various visual representations including graphs, charts, and alerts. This platform excels particularly in handling time series data, which is essential for applications that require continuous monitoring and performance analytics. Grafana's customization options are extensive, allowing users to tailor dashboards to specific needs, and its alerting capabilities are invaluable for maintaining operational awareness and responding promptly to critical conditions.[13]

Each dashboard can be fine-tuned to display real-time information, making Grafana an indispensable tool in domains where timely data visualization and analysis are critical. Its ability to integrate seamlessly with many modern data sources and its user-friendly interface make it an attractive option for system administrators and data analysts alike.

Despite these strengths, we ultimately decided against employing Grafana for the Babybox monitoring system. Our project demanded a more customized solution to align closely with the specific operational workflows unique to Baby-Box monitoring. Grafana, while powerful and flexible, is a general-purpose tool designed to cater to a wide range of monitoring scenarios. The level of customization required to adapt Grafana to our particular needs could potentially lead to a complex and cumbersome setup.

Furthermore, our vision for the Babybox monitoring system includes not only meeting current requirements but also accommodating future expansions and functionalities. Although Grafana is scalable and capable of handling significant amounts of data, tailoring it to support future-specific enhancements and features could limit its effectiveness and efficiency. We needed a system that could evolve without the constraints imposed by a predefined platform, no matter how versatile it might be.

In conclusion, while Grafana offers a compelling set of features for generic monitoring tasks, the unique challenges and specific goals of our Babybox monitoring project necessitated the development of a custom solution. This approach ensures that the monitoring system is perfectly suited to our immediate needs and capable of adapting to future demands, providing a seamless, integrated experience tailored specifically for the context of Babybox operations.[13]

### 1.2.2.2 TICK Stack

The TICK stack is an integrated collection of open-source tools designed to handle time-series data efficiently across various stages of data management. Each component of the TICK stack serves a unique purpose within the ecosystem:[14][15]

- 1. Telegraf:** This is a plugin-driven server agent responsible for collecting metrics and data from a variety of sources and writing them to InfluxDB. Telegraf supports a vast array of data collection plugins, making it versatile for collecting data from virtually any source.
- 2. InfluxDB:** At the heart of the stack, InfluxDB is a high-performance time-series database designed to handle high write and query loads. It is particularly optimized for fast, time-stamped data storage and retrieval, making it ideal for applications that require real-time analytics and monitoring.
- 3. Chronograf:** This component provides a user-friendly graphical interface for visualizing the data stored in InfluxDB. It offers tools for building dashboards and graphs that help users interpret vast amounts of data intuitively and quickly.
- 4. Kapacitor:** A real-time streaming data processing engine, Kapacitor is used for creating alerts, running ETL jobs, and detecting anomalies in the data. It can process both batch and stream data from InfluxDB, enabling complex data processing tasks.

Despite the comprehensive capabilities of the TICK stack, we ultimately decided against implementing it for the Babybox monitoring system for several reasons:

The specific requirements of our project necessitated a more tailored solution that could deeply integrate with the unique operational workflows unique to Babybox monitoring. The TICK stack, while powerful, is a general-purpose suite of tools that would require extensive customization to meet these specialized needs. Such significant modifications could introduce a level of complexity and maintenance that might not be sustainable in the long term.

Furthermore, there were concerns about future expandability and vendor lock-in. Relying on the TICK stack could limit our ability to seamlessly integrate other technologies or adapt to new requirements as the project evolves. Additionally, the absence of Czech language support in the user interfaces of the TICK tools is not compatible with our setting.[14][15]

## 1.3 Technological Concepts

In this section, we will explore several foundational theoretical concepts that are essential for understanding the technological topics discussed in this work.

These concepts encompass a range of topics integral for modern web development and software architecture, including microservices, data storage, or communication models.

### 1.3.1 Client/Server

The client/server architecture is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server hosts, delivers, and manages most of the resources and services to be consumed by the client. This model can be utilized in applications ranging from email exchanges to web browsing, where web servers serve web pages to client browsers.[16]

In this architecture, the client initiates communication sessions, which the server awaits for requests to respond to. This setup allows clients to access a range of network services that are continuously available on servers. The beauty of this architecture lies in its scalability and efficiency; new clients can be easily added, and they can communicate with the server as long as the network is reachable. Servers can also be scaled up to handle increases in workload, making this model adaptable to varying demand levels.[16]

### 1.3.2 Microservice Architecture

Microservice architecture is an approach to building a single application as a suite of small, modular services. Each module supports a specific goal and uses a simple, well-defined interface to communicate with other sets of services. This architecture is a development of the client/server model but breaks down the functionality into smaller, more manageable pieces that can be developed, deployed, and scaled independently. The microservice architecture has several key characteristics: [17][18][19][20]

- **Decentralized Control:** Unlike monolithic architectures, where a single database or a central unit of operation manages the entire application's processes, microservices decentralize control and data management. Each service in a microservice architecture operates independently with its own database and state, reducing dependency on a central management system.
- **Flexibility in Technology:** Microservices can be written using different programming languages, databases, hardware, and software environments depending on what fits best for their defined responsibilities. This polyglot approach allows developers to use the right tool for the right job rather than being constrained to the choices made at the start of a project.

- **Resilience:** By segregating services, the system ensures that even if one service fails, the others continue to function without disruption. This isolation helps in containing faults within the system and makes it easier to restart or replicate services without affecting the entire application.
- **Scalability:** Services can be scaled independently, allowing more precise handling of load changes. For instance, a component that handles payment can be scaled up during high-demand periods without having to scale the entire application.
- **Ease of Deployment:** Due to their smaller size, microservices can be quickly deployed and updated, allowing for frequent updates and faster release cycles. This agility supports continuous integration and continuous deployment practices that are integral to modern software development cycles.

In microservice architecture, services communicate with each other through well-defined APIs. They often employ lightweight protocols. Communication can be either synchronous or asynchronous. Asynchronous communication, favored in microservice architectures, is typically handled through a message broker.

The use of microservices brings several benefits but also introduces complexities in service integration, data consistency, and management. Therefore, organizations must weigh these factors based on their specific needs, considering both the size and scope of the project.

This architecture not only changes how applications are built but also affects how teams function, necessitating a culture of collaboration and continuous learning among teams.[17][18][19][20]

### 1.3.3 Message Broker

A message broker is a key intermediary software module that translates messages between disparate telecommunication, data communication, or computing systems. This module enables software applications, which may be built with different programming languages and on different platforms, to communicate seamlessly.[21][22]

#### 1.3.3.1 Concepts

- **Producer (Publisher):** A service that sends messages. It creates messages and sends them to an exchange, initiating the message delivery workflow.
- **Consumer (Subscriber):** A service that retrieves messages from queues. It processes messages that have been added to a queue by a producer.
- **Exchange:** A component that receives messages from producers. It decides how to route these messages to one or more queues based on the message

attributes, bindings, and the type of exchange used. The exchange plays a pivotal role in determining the flow of messages within the system.

- **Queue:** A component stores messages until they are processed by a consumer. It acts as a buffer that allows asynchronous processing—producers can continue adding messages to the queue without waiting for them to be processed.
- **Binding:** A rule that links a queue to an exchange. Bindings can have optional routing keys that direct the exchange on how to route messages to the correct queues.
- **Routing Key:** A specific attribute or identifier associated with a message that helps the exchange to route the message to the appropriate queue based on the binding rules.

### 1.3.3.2 Strategies

Message brokers support various messaging strategies and patterns that cater to different communication needs:

- **Point-to-Point:** In this simplest form, a producer sends a message and one consumer receives it directly. It's effective for direct and uncomplicated communication.
- **Work Queues:** This pattern distributes tasks among multiple workers. It's used to scale processing and handle multiple jobs by distributing them among several consumers, typically to balance load.
- **Routing Exchange:** Utilizes routing keys to direct messages to specific queues based on certain criteria. This allows for more directed and filtered communication.
- **Topic Exchange:** An extension of routing exchange that supports pattern matching against multiple criteria. It enables more flexible and dynamic routing scenarios.
- **Publish/Subscribe (Fanout):** This strategy broadcasts every message to all of the available queues connected to an exchange. It is used when the same message needs to be delivered to multiple consumers.

Using these components and strategies, message brokers facilitate diverse and flexible communication options for distributed systems. They enable effective decoupling of application components, which can improve scalability, fault tolerance, and the overall flexibility of system architecture.[21][22]

### 1.3.4 API Gateway

An API Gateway is a fundamental component in modern application architectures, especially those utilizing microservices. It acts as a reverse proxy that routes incoming requests from clients (such as web browsers or mobile apps) to various backend services. By centralizing common functionalities such as request routing, authentication, and rate limiting, an API Gateway simplifies the complexities involved in managing microservice infrastructures. The core functionalities API gateway can handle are:[23][24]

- **Request Routing:** The primary role of an API Gateway is to accept incoming API requests and route them to the appropriate microservice based on the URL path, method, and possibly other headers. This routing capability allows developers to decompose backend services according to business functionalities while presenting a unified API endpoint to clients.
- **Authentication and Authorization:** An API Gateway can centralize common security tasks like authentication and authorization. It can validate access tokens, ensuring that requests are allowed to access particular services and operations. This removes the necessity for each microservice to implement its authentication logic, promoting a DRY (Don't Repeat Yourself) approach and reducing the chance of security breaches due to inconsistent implementations across services.
- **Rate Limiting and Throttling:** To protect backend services from being overwhelmed by too many requests, an API Gateway can enforce rate limiting and request throttling. This ensures that services remain responsive and available even under high load, preventing resource exhaustion and potential service downtime.
- **Load Balancing:** By distributing incoming requests evenly across a pool of instances for each service, an API Gateway optimizes resource utilization and maximizes throughput. Load balancing improves the overall performance of the application by ensuring no single service instance becomes a bottleneck.
- **Other concerns:** Besides routing and security, API Gateways can handle other concerns such as logging, monitoring, and response transformation. For instance, an API Gateway can aggregate responses from multiple services and transform them into a format expected by the client, simplifying client-side logic.[23][24]

### 1.3.5 REST API

A REST API (Representational State Transfer Application Programming Interface) defines a set of rules for how web-based applications should communicate with each other using stateless operations over HTTP. Its design is guided

by a few key principles that make web interactions more efficient and flexible. [25][26][27]

### 1.3.5.1 Concepts

- **Resources:** In REST, a resource refers to any content or information that can be named and is the concept around which REST revolves. Each resource is uniquely identified by a URI (Uniform Resource Identifier). Resources represent objects or data entities and can be a document, an image, or a collection of other resources.
- **HTTP Methods:** REST uses standard HTTP methods to perform actions on resources, each method specifying a different type of operation:
  - **GET:** Retrieves data from a server; it should only retrieve data and have no other effect (idempotent).
  - **POST:** Sends data to the server for a new entity. It is often used when creating a new resource.
  - **PUT:** Updates existing data or creates a new resource at a specific URI if it does not exist, and it is idempotent.
  - **DELETE:** Removes data from the server, and it is idempotent.
- **Statelessness:** Every HTTP request from a client to server must contain all the information needed to understand the request. The server does not store any state about the client's session. This makes the REST service scalable and visible, as the lack of a required connection between a series of requests benefits load balancing and fault tolerance.
- **Headers:** HTTP headers let the client and server pass additional information with an HTTP request or response. Headers control the behavior of request/response or provide information such as formats a client can understand (**Accept** or **Content-Type** headers). Another important header is the **Authorization** header, which can be used by the client to authorize itself using a token.
- **Query Parameters:** These are optional key-value pairs that appear after the question mark (?) and further delimited by ampersands (&) in the URL. They are used to further refine HTTP requests. For example, filtering results in a GET request.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format that's easy to read and write for humans and easy to parse and generate for machines. JSON is commonly used in REST APIs to format data sent between clients and servers but other formats (such as XML can be used). JSON can exchange a few different types of data that can be nested inside each other:



- Numbers: JSON can represent double-precision floating-point format numbers. It does not distinguish between integers and floating-point numbers.
- Strings: Strings in JSON must be written in double quotes. They can contain a sequence of zero or more Unicode characters.
- Booleans: JSON supports the `true` and `false` values typical of boolean data types.
- Null: JSON can represent a null value with the keyword `null`.
- Arrays: An array is an ordered collection of zero or more values, enclosed in square brackets (e.g., `[1, "hello", true]`). Arrays in JSON can contain items of different types, including numbers, strings, objects, arrays, booleans, and null.
- Objects: An object is an unordered collection of key:value pairs, with the keys as strings and the values as any other JSON data type. Objects are enclosed in curly braces (e.g., `{"name": "John", "age": 30}`).

These principles and components enable REST APIs to facilitate clear, reliable, and efficient communication between distributed systems over the web.[25][26][27]

### 1.3.6 Server Side and Client Side Rendering

When choosing between server-side rendering (SSR) and client-side rendering (CSR), it's important to understand not just the basic mechanics of each approach, but also when rendering occurs and the implications of when it happens for performance, SEO, and application architecture.[28][29]

#### 1.3.6.1 Server-Side Rendering

In SSR, the HTML content of a web page is generated on the server in response to a user's request. This happens before the page reaches the browser. The server executes all the logic required to compile the page content, including fetching data from databases or APIs, and renders the final HTML to send to the client. An extension of this approach is Server Side Generation is when the page is generated during build time of the application (mostly for static pages).

SSR is often times combined with caching on the server, where the server caches the rendered page to improve response times for the same requests. There are many different strategies for caching and when to rerender the page.

The implications of this approach are that the browser receives a fully-rendered HTML page, which means the content is immediately available for display and interaction. This reduces the perceived load time and can enhance user satisfaction while also improving SEO as search engines can easily crawl

and index the fully rendered HTML, improving the visibility of the site in search results. This is particularly important for content-heavy sites where organic search traffic is a priority.

Since SSR occurs on the server, it has direct access to backend resources, potentially on the same local network or within a Docker network when used in a containerized environment like Docker Compose. This proximity can lead to faster data retrieval and integration, as it avoids the latency that might be involved with external API calls from the client side.[28][29]

### 1.3.6.2 Client-Side Rendering

CSR occurs entirely in the browser. Upon the initial request, the server sends a minimal HTML document with links to JavaScript files. The browser executes the JavaScript, which typically makes API calls back to the server to fetch data and then dynamically generates the HTML content on the client side.

The initial load time includes downloading, parsing, and executing the JavaScript, which means there can be a significant delay before any content is rendered on the screen. This can impact user experience negatively, particularly on slower networks or devices.

Furthermore, dynamic content rendering poses challenges for SEO as search engine crawlers may not effectively process JavaScript that loads content after the initial page load. This can reduce the effectiveness of SEO strategies for sites reliant on search engine traffic.

Unlike SSR, CSR typically cannot directly access backend resources on a local or Docker network. It must make separate API calls over the internet, which can introduce additional latency and complexity, especially when handling secure or sensitive data.

Although CSR has many negatives, it has one crucial advantage over SSR - it can, by definition, run code on the client side, therefore, unlike SSR it allows for user interaction which is crucial on websites that are more dynamic. Any component with a user interaction, such as a button, popup, modal and many others, need to be rendered using CSR to allow for running the code on the client side, when user interacts with the application.[28][29]

### 1.3.6.3 When to Choose Which?

Deciding between SSR and CSR involves weighing these factors against the specific needs of the application. SSR offers immediate content availability and excellent SEO benefits but can put a heavier load on the server. CSR provides an interactive user experience suitable for applications where user engagement is more critical than immediate content availability. Modern web development often involves a blend of both techniques to leverage the strengths of each, particularly with frameworks like Next.js that facilitate hybrid rendering strategies with granularity per each component.[28][29]

### 1.3.7 PWA

Progressive Web Applications (PWAs) utilize several integral technologies to provide a seamless and robust user experience akin to native apps, but through a web browser. All of these things need to be configured for PWA to be enabled.[30][31]

#### 1.3.7.1 Service Workers

Service Workers are pivotal to the functionality of PWAs. These scripts operate in the background, independent of the web page, and handle key operations like network requests and data management in offline conditions. This capability allows PWAs to load content, perform actions, and maintain a responsive interface even when network connectivity is limited or absent. Service Workers also enable background data synchronization, ensuring that any changes made offline are seamlessly updated once connectivity is restored. Additionally, they facilitate push notifications, which are important for keeping users engaged by delivering timely and relevant updates directly to their device.[30][31]

#### 1.3.7.2 Web App Manifest

The Web App Manifest is a configuration file in JSON format that tells the browser about the PWA and how it should behave when “installed” on a user’s device. It includes details such as the app’s name, icons, and start URL, which collectively influence how the app appears on the home screen and task switcher. The manifest also configures the PWA’s display settings, such as whether it should run in full screen or in a standalone window, essentially guiding the browser to treat the web app more like a native app. This setup is crucial for enhancing the user’s perception of the PWA as a real application rather than just another webpage.[30][31]

#### 1.3.7.3 HTTPS

Security in PWAs is enforced through HTTPS, ensuring that all communications between the user’s device and the server are encrypted. This encryption is essential not only for safeguarding user data from interception and tampering but also for maintaining user trust, particularly in applications that handle sensitive transactions or personal data. HTTPS is a fundamental requirement for the use of Service Workers, as the security risks of intercepting service worker traffic could otherwise compromise the entire application.[30][31]

### 1.3.8 SWR

SWR (Stale-While-Revalidate) is a strategy and React hook implemented by Next.js for efficient data fetching. This approach optimizes the way data is

loaded and updated in web applications, enhancing user experience by providing immediate access to data ("stale") while simultaneously updating the cache in the background ("revalidate").[32]

SWR works on the principle of returning the cached data first (stale), then sending the fetch request (revalidate), and finally coming back with the latest data. This process allows the interface to remain responsive and appear fast, as users see data immediately while the newest data fetches in the background.

- **Immediate Data:** When a component mounts or a request is made, SWR first provides data from cache, reducing the initial load time and perceived latency.
- **Revalidation:** After serving the stale data, SWR re-fetches the data from the server to get the most current version. If the new data differs from what's in the cache, it updates the cache and re-renders the UI with the updated data.
- **Efficient Updates:** SWR uses several smart features like deduplication of requests, on-focus revalidation (revalidating data when a window regains focus), and interval-based polling to keep the data fresh without overwhelming the server or the network.

By leveraging SWR, developers can build fast, reliable web applications with Next.js that handle data fetching elegantly, enhancing both the performance and the user experience.[32]

### 1.3.9 Containers

Containers offer a lightweight, efficient method to ensure that software runs reliably when moved from one computing environment to another. Docker, a leading containerization platform, uses containers to encapsulate an application's software environment, simplifying deployments and scaling across diverse systems.[33][34]

#### 1.3.9.1 Docker

Docker provides a platform for developers to package applications along with their dependencies into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.[33][34]

- **Image:** A Docker image is a static snapshot of the container's environment, capturing the application and its environment at a specific point in time. Images serve as the building blocks of Docker containers, defining what the environment should look like and which software it contains.

- **Container:** A container is a runtime instance of an image, where the image's static file becomes a live environment when the container is running. Containers isolate and secure the application ensuring it works uniformly despite differences, for example, between development and staging environments.
- **Dockerfile:** This is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

### 1.3.9.2 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With a simple YAML file, developers can configure application services and manage them as a single service when deployed.[35]

- **Compose file:** The `docker-compose.yml` file is where one defines Docker applications using various services, networks, and volumes. Here, each service can run in its container and interact with others.
- **Services:** Defined in the Docker Compose file, services are the actual containers in operation. Each service can have its configuration concerning how it is built, ports that are exposed, and more.
- **Volumes:** These are used to persist data generated by and used by Docker containers. When containers are deleted, their volume remains active, allowing data to persist.
- **Networks:** Docker networks allow containers to communicate with each other. Compose networks are isolated by default, providing each service with access to a default network to facilitate communication.
- **Environment Variables:** Used to pass configuration to the services running inside containers. They can dynamically set service parameters without hard coding them into the image.
- **Labels:** Key-value pairs attached to objects such as containers, images, networks, or volumes, useful for organizing images, managing container lifecycles, and configuring policies.
- **Restart Policies:** Determine how Docker should handle container exits. Policies like `always`, `on-failure`, or `unless-stopped` specify whether containers start automatically under certain conditions

### 1.3.9.3 Benefits

- **Consistency and Isolation:** Containers are fully portable and provide consistent operations across any platform. Docker encapsulates the application's environment, and its dependencies into a Docker container, which can be moved across systems and executed without change.
- **Developer Productivity:** Docker simplifies the development process by allowing developers to create predictable environments that are isolated from other applications. Docker Compose further enhances productivity by enabling developers to define and coordinate the operation of multi-container applications.
- **Operational Efficiency:** Docker containers can be started almost instantly, which means that scaling up to handle load spikes is straightforward. Docker Compose manages the entire lifecycle of application services collectively, simplifying deployment and scaling.
- **Docker and Docker Compose represent vital tools in modern software development, promoting more efficient application deployment and management through containerization technology. These tools ensure applications perform as expected in different environments by standardizing the software distribution process.**[33][34][35][36]

## 1.3.10 NoSQL Databases

NoSQL databases are designed to provide flexible schema, scalability, and high performance for various types of data models. Unlike relational databases, which use tables and a fixed schema, NoSQL databases use a variety of data models, including document, key-value, wide-column, and graph. These models are designed to handle large volumes of data distributed across many machines. NoSQL is particularly effective for applications requiring large data storage, rapid development, or agile sprints that frequently adjust data models.

### 1.3.10.1 MongoDB

MongoDB is a document-oriented NoSQL database known for its high flexibility, which stores data in JSON-like documents with dynamic schemas. This model allows applications to store data in a way that is closer to how data is represented in the application code, making it easier to work with and improving developer productivity.[37][38]

- **Collections and Documents:** Data in MongoDB is organized into collections, which are analogous to tables in relational databases. Each collection holds documents, which are sets of key-value pairs. Documents can

contain nested documents and arrays, allowing for a rich data structure that can easily adapt to changes.

- **Basic Queries:** MongoDB allows for performing basic queries by specifying a document containing the field values in question. For example, to find all documents in a collection where the "name" field equals "John", one would use `{name:"John"}` as the query.
- **Range Queries:** It is possible to query documents based on range conditions using operators such as `$gt` (greater than), `$lt` (less than), `$gte` (greater than or equal to), and `$lte` (less than or equal to). For instance, `{age:{$gt:30}}` would match documents where the age field is greater than 30.
- **Regular Expressions:** MongoDB supports using regex, which allows for flexible searching of strings. For example, finding documents where the "name" starts with 'J' can be performed with `{name:/^J/}`.
- **Aggregation Pipelines:** the aggregation framework is a powerful feature in MongoDB designed to process data records and return computed results. It groups values from multiple documents together and can perform a variety of operations on the grouped data to return a single result. MongoDB's aggregation framework operates in a pipeline, with each stage transforming the documents as they pass through.
- **JOIN operation:** Although MongoDB does not natively support joins as in SQL databases, it can achieve similar results using a `$lookup` stage in an aggregation pipeline, which can be used to perform a left-outer-join-like operation between documents from different collections. It essentially adds new fields to documents from the joined collection based on a specified matching condition.

#### 1.3.10.2 InfluxDB

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data, events, and metrics. It is designed to handle massive volumes of time-stamped information, making it ideal for applications such as monitoring real-time analytics, IoT, and sensor data.[39]

- **Buckets:** In InfluxDB, data is stored in buckets, which are organized by time and retention policies. This structure helps manage data lifecycle effectively by specifying how long data should be stored before it is automatically deleted.
- **Measurements and Fields:** Data points in InfluxDB are organized around measurements, tags, and fields. Measurements act as a data descriptor

similar to a table name in a relational database, tags provide indexed meta-data, and fields are non-indexed data. This structure allows for efficient data organization and retrieval.

- Flux: InfluxDB uses Flux, a powerful functional data scripting and query language that supports complex data processing, joining of data streams, and alert management.
- Line Protocol: The primary way to write data to InfluxDB is through the Line Protocol, a text-based format for writing points to the database. It encapsulates measurement, tag set, field set, and timestamp for each data point.
- Time-series ready: InfluxDB offers many functions and features that take advantage of the fact that InfluxDB is a time-series database, such as downsampling, continuous queries and many more.
- Aggregate Window Function: This function allows for partitioning of data into windows of time and apply an aggregation function to each window. For example, one can calculate the average, sum, count, or any other aggregate measure over fixed time intervals. This is extremely useful in scenarios where one needs to analyze trends over time, such as monitoring average response times within 10-minute intervals throughout a day.

### 1.3.11 JWT Authenticaiton Mechanism

JSON Web Tokens (JWT) offer a secure and efficient method for transmitting information as compact JSON objects that can be verified because they are digitally signed. This format is widely used for authentication and secure data transmission between parties. A JWT is composed of three parts: Header, Payload, and Signature.[40][41]

- Header: Specifies the token type (JWT) and the signing algorithm (e.g., HS256).

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Payload: Contains claims such as issuer, subject, expiration, and additional data related to the user.

```
{
  "sub": "1234567890",
```



```
"name": "John Doe",  
"admin": true  
}
```

- Signature: Generated from encoding the header and payload and signing it with a secret to ensure the token's integrity.
- Resulting JWT can look like this, where each section is divided by a period (.):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0Ijoi.  
4TQL9E3i7MYToMsiKL31yoM1iZmNcFRLzoWtZLx7b8Y
```

JWTs are ideal for decentralized authentication systems, particularly useful in distributed environments like microservices. Each service can independently verify a JWT without needing central authentication, enhancing system scalability and resilience. This approach allows each part of a system to authenticate tokens, streamline security processes, and reduce bottlenecks.

JWTs enable secure, scalable, and efficient user authentication, making them a cornerstone of modern web security strategies.[\[40\]](#)[\[41\]](#)

# Analysis and Design

*This chapter will cover the methodology in the decision process before and during software development, then it will go over the choices of technologies for this project. After that, there is a section explaining the analysis of user requirements; describing the personas and their use cases, functional and non-functional requirements. Finally, we will go over describing the previous solution with a comparison to the new solution, along with architectural and system designs.*

## 2.1 Methodology

As discussed previously, the unique context of this project significantly influences the choices made throughout every stage of the development process.

This section outlines the methodology adopted for the software development process and decision making towards choosing technologies within this project, providing a structured framework to guide our development efforts and also the methodology for other decision making made.

### 2.1.1 Software Development Process

As a project with a single developer, we adopted a very flexible iterative approach, which involves the following elements:

1. **Prioritizing staff feedback** - throughout the development process, we regularly consulted with the primary maintenance staff making sure we communicated the changes made compared to the previous version of the software, but also reaching for new ideas and improvements with each feature. We ensured that the new features and design choices align with their practical needs and address the shortcomings of the previous application.

We set up a weekly and sometimes even a daily communication routine in person with the personnel to keep them and us updated about the progress

and decision making taken during the development process.

2. Iterative refinement - new features were rolled in stages, focusing on the user experience and critical details first. This ensured that changes are small and manageable.
3. Selective testing - as this is a fast moving project with limited resources we focused on manual testing in a combination of unit testing and testing the API interface.
4. Version control - GitHub was used extensively for code management, version control and documentation. In the future, we also plan to use it for issue tracking and potential collaboration.
5. Documentation - We used GitHub for the developer documentation; for general documentation and analysis we used an internal wiki to track long-term features and requirements.

We have decided to use the microservice architecture for the benefits it has, mainly they are: technology agnostic, more scalable, better at isolating faults. We will discuss this choice more extensively later.

### 2.1.2 Technological Freedom in Microservice Architecture

When designing a system based on the microservice architecture, it is important to establish guidelines for technology choices to ensure consistency. One of the main advantages of microservices is that they are technology agnostic - each microservice can take advantage of the most suitable technology according to its needs.[20]

While technological freedom is beneficial, too many choices can hinder developer efficiency and maintainability. When each service can use any technology, the overall system might become hard to work with as developers can become lost and cannot utilize their knowledge and experience acquired from one service to another. It is a good idea to mitigate such problems by restricting the technologies we use throughout the codebase.

This section will focus on finding a good middleground set of programming languages, frameworks and other technologies to use within the application to avoid the codebase becoming too confusing and complex.

### 2.1.3 Guiding Principles

The choices for this application have to account for some core limitations that this project has, those are mainly:

- Project Requirements

- Budget
- Manpower Limitation

The primary consideration of this project is on the features and user requirements that need to be delivered with limited resources both in budget and human resources. The main emphasis is to choose technologies that are suitable for **delivering requested features** as easily as possible, therefore the technologies should promote **developer efficiency** and have a gentle learning curve to streamline development and minimize potential hurdles.

Another factor to consider, that goes hand-in-hand with the previous point, is **popularity**, strength of the support and the community around the technology and the quality of the **documentation**. These factors, although subtle, can tremendously speed up the development process.

We still value **performance** especially for the more demanding areas, such as the database or the services with higher expected load.

After considering these factors, the final chosen technology must also make sense in the grand scheme of things - it must fit in among the other parts of the application. If we choose to use a message broker and then later choose a language that is perfect for the job, but cannot communicate with the message broker, then the choice is still very poor.

### 2.1.3.1 Design Principles and Patterns

In rewriting this application, several key design principles and patterns have been adopted to enhance system functionality, and operational efficiency. These are categorized into microservice patterns, RESTful API practices, and general programming principles.

#### Microservice Patterns

To avoid common pitfalls, we embraced several patterns and ideas from established practices in the field, inspired by Chris Richardson's book on microservice patterns:[19][20]

- Decomposition by Business Capabilities: Services are decomposed based on business functionality which simplifies development, deployment, and scaling of independent service units.
- Shared Database: Each microservice utilizes its own distinct set of collections within MongoDB, ensuring that there is no overlap in data management which minimizes interference and enhances performance.
- API Gateway: Initially, Traefik was considered for its dynamic service routing capabilities; however, Caddy was eventually chosen for its simplicity and effective static and dynamic routing abilities in production.

- Health Check API: Implemented across services for monitoring but not actively used in operations; provides readiness checks and potential for future use in service orchestration and recovery strategies or at least for user's information about the system's health.
- Log Aggregation: Utilizing Docker Compose's built-in capabilities for logging, which aggregates logs from all containers, simplifying debugging and monitoring.
- Access Tokens Using JWT: JSON Web Tokens (JWT) are used for managing access and ensuring secure and stateless communication between services.
- Service Communication: While developing, services communicate over a Docker network which facilitates isolated networking and interaction testing between containers.

### REST API Best Practices

In REST API design, we adopted structured practices to ensure APIs are unified and predictable:[\[26\]](#)[\[42\]](#)

- Resource Naming: Adheres to conventional standards for naming endpoints, focusing on nouns that represent entity relationships clearly and intuitively.
- CRUD Operations: Each resource supports standard CRUD operations, providing a consistent and predictable framework for interacting with data.
- Envelope Pattern: Used in API responses to wrap data, allowing for standardized communication of results and metadata (mostly for error handling).
- Versioning: APIs are versioned to manage changes gracefully, ensuring backward compatibility and clear evolution of service interfaces.
- Query Parameters: Support for extensive filtering, sorting, limiting, and formatting options through query parameters enhances the flexibility and user-friendliness of APIs.
- Non-Adherence to HATEOAS: The system does not implement the Hypermedia as the Engine of Application State (HATEOAS) principle, as it was deemed unnecessary for the application's current needs and would introduce additional complexity.
- Adapted REST Practices: While generally adhering to RESTful principles, the system makes a pragmatic exceptions. One such big concession is for receiving snapshots from babyboxes. Due to backwards compatibility reasons, we receive the snapshots using GET request with the data in the query parameters.

### Programming Principles and Patterns

Additionally, programming practices were aligned with both functional and object-oriented paradigms to optimize code maintainability and system functionality.[43][44][45]

- **Idiomatic Language Use:** Efforts are made to utilize each programming language in its idiomatic form. For instance, Go is employed with its conventional procedural style, focusing on simplicity and efficiency without adopting functional programming paradigms. Conversely, TypeScript is used in a style that embraces more functional programming concepts, leveraging advanced type systems and functional constructs to enhance code safety and reusability. Similarly, we tried following the idiomatic codebase structures and more. This approach ensures that the codebase not only follows best practices for each language but also aligns with the community standards and expectations.
- **Avoid Premature Optimization:** A principle that states that unless there is evidence of some performance issue we should not further optimize. This principle can be applied on any level of granularity: for functions as well as for services.
- **Immutability:** Ensured in data handling to prevent side effects and facilitate easier state management, particularly in services handling complex data structures.
- **Stateless Design:** Encouraged to make services scalable and easily manageable, as stateless applications are easier to deploy and scale.
- **DRY (Don't Repeat Yourself):** A key principle in the codebase to reduce redundancy and increase maintainability.
- **Single Responsibility Principle:** Every module or class should have responsibility over a single part of the functionality provided by the software, which is encapsulated in that module or class.
- **Functional Programming:** Emphasizes the use of pure functions, immutability, and stateless components to build reliable and predictable code. We plan to write code inspired by this paradigm in the microservices and the front-end using JavaScript, on the other hand we plan to avoid this paradigm in the Go services purely based on the nature of the programming languages.

## 2.2 Requirements Analysis

This section lays the foundational framework for this work, establishing the primary specifications that guide all subsequent design and development pro-

cesses. By analyzing and defining both the functional and non-functional requirements, we ensure that the system is fully equipped to meet the specific needs of its users while adhering to high standards of performance and reliability.

The objectives of this section are:

- To identify and understand the key user groups and their specific requirements for the system through user personas.
- To describe the use case scenarios that the system must support.
- To describe clear functional requirements that describe what the system will do.
- To describe non-functional requirements that focus on how the system will perform the required functions.

The importance of comprehensive requirements analysis cannot be overstated, as it directly impacts the usability and effectiveness of the final product. By rigorously defining these requirements, we lay a solid foundation for developing a system that is not only technologically sound but also user-centric, ensuring it effectively supports the day-to-day operations of those who rely on it.

In the subsequent sections, we will explore the methods used to gather user requirements, develop representative use cases, and specify the necessary functional and non-functional requirements. The following chapters will build upon this foundation, detailing the system design, implementation, and evaluation processes.

### 2.2.1 Identification and Analysis of User Requirements

To ensure the system effectively meets the needs of its users, extensive research was conducted, combining various methodologies to gather actionable insights.

We began with interviews with both operational and maintenance staff, aiming to understand their daily interactions with the existing system and identify any challenges they face. These discussions provided valuable context about their routine tasks and highlighted inefficiencies and limitations in the current system. Staff members were encouraged to share how the system could better support their work, offering insights into desired features and functionalities that would enhance their efficiency and effectiveness.

Simultaneously, we conducted an analysis of the current system usage to pinpoint frequent pain points and the features most in need of improvement. This involved observing users as they navigated the existing interface and managed their tasks, which helped clarify which aspects of the system were

obstructing their workflow. The feedback was particularly critical in understanding the practical difficulties users encountered and what they felt were missing capabilities.

Furthermore, we explored how operation staff solved problems using the current software. This routine and problem-solving analysis involved real-time observations and interactive sessions, where users demonstrated their process for addressing issues. This approach was instrumental in uncovering implicit needs not readily articulated during interviews, such as the need for quicker access to operational data or more intuitive navigation paths.

### 2.2.2 User Personas

The user requirements analysis led to the development of detailed personas that represent the core users of the Babybox Monitoring System. These personas help to humanize the abstract data and insights collected, making it easier to design solutions tailored to specific needs.

**Oscar the Operator:** Oscar is an experienced operator responsible for monitoring babyboxes. He relies heavily on the system for real-time data analysis to proactively manage and address operational issues. Oscar benefits greatly from a sophisticated dashboard that integrates advanced analytics and customizable alerts, allowing him to efficiently oversee the system's health.

**Max the Maintenance Technician:** Max's role requires him to be highly mobile, often traveling to various sites to address urgent maintenance issues. He needs a system that is straightforward and quick to use. The ideal interface for Max would display essential information such as a simple status and location of the next babybox.

They both work from different environments; Oscar is more likely to work on a desktop computer, but also might need to use the application on a tablet or a phone. Max will more likely work on his phone.

We decided to put our main emphasis on Oscar and tailoring the application to his needs, while also keeping in mind Max's requirements and needs, where it makes sense.

### 2.2.3 Use Cases

This chapter delineates a series of use case scenarios that illustrate the functional capabilities of application, particularly focusing on the needs of Oscar, the operator.

All use cases assume that Oscar and Max have been onboarded to the application: accounts have been created, and they have successfully logged into the application.

## 1. Monitoring System Health



Actor: Oscar

Goal: To verify the current operational status of all babyboxes.

Main flow:

- a. Oscar accesses the main dashboard, which displays a list of all babyboxes.
- b. Each babybox is listed with a clear indication of its current status, using color codes or icons for quick visibility.
- c. Oscar reviews the list to identify any units signaling potential issues, focusing on those that may require immediate attention.

## 2. Analyzing a Problem with Heating (or cooling, voltage, etc.)

Actor: Oscar

Goal: To diagnose and resolve heating issues in a specific babybox.

Main flow:

- a. Oscar selects the problematic babybox from the dashboard and navigates to its dedicated analysis page.
- b. The system presents visualizations and tabular data from the babybox.
- c. Oscar modifies the type of data he wants to see to identify the problem.
- d. Oscar customizes the data display by adjusting the type and time range of the data shown to better isolate the issue.
- e. He examines historical data and visualizations to pinpoint the underlying cause of the heating malfunction.

## 3. Custom Notifications

Actor: Oscar

Goal: To set up custom alerts for babyboxes deviating from optimal operational parameters.

Main flow:

- a. Oscar goes to the notification settings page within the system.
- b. He defines the scope and specifics of the alert, such as which parameters to monitor and the threshold for triggering an alert.
- c. The system provides options to set conditions based on temperature and voltage readings.
- d. Additional settings are adjusted to minimize notification fatigue, such as delays or streaks.
- e. Oscar finalizes and saves a notification template, which the system uses to monitor conditions and send alerts when parameters breach the set thresholds.

- f. The system later checks for conditions for generating a notification to be met with each new snapshot.
- g. Oscar receives the notification to act upon.

#### 4. Contacting Hospital Staff

Actor: Oscar or Max

Goal: To facilitate direct communication with hospital staff associated with a specific babybox.

Main flow:

- a. The actor selects a babybox and views its detailed information page.
- b. The page lists all relevant hospital staff along with their contact details, including names, phone numbers, emails, and positions.
- c. Oscar or Max uses this information to initiate contact directly from the system interface if needed.

#### 5. Navigating to a babybox

Actor: Max

Goal: To find the quickest route to a hospital for maintenance work.

Main flow:

- a. Max selects his target babybox from a list and accesses its location information.
- b. The system displays comprehensive location details, including the hospital's address and the specific site of the babybox within the facility.
- c. A direct link to open the location in an external navigation app (such as Google Maps or Mapy.cz) is provided to guide Max to the hospital efficiently.

### 2.2.4 Functional Requirements

These requirements outline the specific actions and capabilities that the system must possess to fulfill the needs identified through user interactions and use case scenarios.

#### F1 User Authentication

The system must provide a user management and authentication system. Babybox Dashboard is an internal tool with sensitive information that should be protected behind an authentication system. The system must also allow for creating new users within the interface.

**F2 Dashboard Display**

The system must provide a central dashboard that lists all babyboxes with the latest status indicators.

Each status indicator should utilize visual cues such as color coding or icons to represent different operational states (normal, warning, critical/error).

**F3 Data Visualization and Analysis**

Detailed data visualization tools should be available for analyzing parameters like temperatures and voltages.

Users should have the ability to customize which data are shown, filter based on time range, and other predefined criteria.

**F4 Notifications System**

A customizable alert system must be implemented to allow users to set and adjust thresholds for critical operational parameters.

The system should include settings to control notification frequency and parameters to prevent notification fatigue.

The system should support setting a scope of the notification either for all babyboxes or for a specific babybox.

**F5 Sending notifications**

Notifications should be generated with each snapshot based on if the conditions are met. Notifications should be displayed in the application and also sent as an email.

**F6 Detailed Information**

The system must facilitate easy access to detailed information about each babybox and its related information:

- Contact information - names, phone numbers, email addresses, positions/departments.
- Location information - Hospital name, city, street, postcode, coordinates; with an integration with external navigation applications.
- Network configuration information - Type of configuration, IP addresses of the router, units, camera and gateway.

**F7 Editing Information**

The system must be allow for an easy modification and addition of the detailed information defined in F6.

**F8 PWA and Responsive Design**

The application should have basic PWA functionality and with that also have a responsive design such that it can be used on mobile, table and desktop.

### 2.2.5 Non-functional Requirements

For the development of this system, we have derived non-functional requirements from the functional needs and user interactions identified in previous sections. These requirements are crucial for ensuring that the system not only functions as intended but also adheres to high standards of performance, security, usability, and maintainability.

To systematically address these aspects, we have categorized the non-functional requirements into distinct groups: Performance, Security, Usability, Scalability, and Maintainability. Each category is equipped with specific metrics to quantitatively measure the system's adherence to these standards.

#### [NF1] Concurrent Snapshot Handling

Category: Performance

Description: The system must efficiently manage multiple requests simultaneously to ensure reliable performance under peak usage.

Metric: The system should handle at least 100 incoming requests simultaneously, with the 90th percentile response time be under 2 seconds.

#### [NF2] Secure Communication

Category: Security

Description: To protect data integrity and confidentiality, the system must encrypt all user transmissions using HTTPS, ensuring secure exchanges between the client and server.

Metric: 100% of data transmissions of the user must be conducted over HTTPS.

#### [NF3] Maintainability and System Updates

Category: Maintainability

Description: The system architecture must support easy maintenance and updates with minimal disruption to service. This involves using modular designs and clear documentation that simplify the integration of new features and updates.

Metric: Each (re)deployment should result in less than a 10 minutes downtime and 80% of bugs should be repaired within 1 hour of accepting them.

## 2.3 Choosing technologies for This Project

In this section, we will take the methodology to practice to find the best technologies for this particular project based on the context, limitations, use cases and requirements that were discussed in the previous sections.

### 2.3.1 Programming Languages and Frameworks

The most fundamental choices are in the space of programming languages, this choice is mainly influenced by the type of services the system is going to be composed with. In the case of our project, I identified 3 types of services:

- High-performance services - services that will be under heavier load
- Analytical services - services that will use analytical functionalities
- Standard services - services that do not require any special needs

We have decided to use Go for high-performance services as it is an easy language to use and move quickly with. It is also a very common language to use in microservices and therefore has robust support in terms of libraries and other resources. It does satisfy all the guiding principles, while being very performant.

Python is a great choice for analytical services as it is a de facto standard solution for data science and machine learning use cases, while also obviously being more than suitable for services providing basic statistics and aggregations.

The rest of the services can be implemented in TypeScript as it offers the greatest versatility and community support. It is also a sensible choice as the front-end of this application is going to be written in TypeScript as well.

Programming languages and frameworks, especially those used on the back-end, need to integrate into the ecosystem. They need to have extensive libraries and drivers to communicate with the databases and message brokers we will be using. Fortunately, as we are mostly going to be using popular technologies, we do not have to worry about this too much as there is effort on both ends to be compatible with each other.

### 2.3.1.1 Next.js

We selected Next.js as the main front-end framework to be used. Next.js is a popular React-based framework, which offers several compelling advantages that align with the project's requirements and guiding principles.<sup>[46]</sup>

It supports both **client-side and server-side rendering**, letting one choose which rendering strategy they want to use per each component or page. This is done using the `"use client";` directive at the top of the file.

Next.js improves **developer productivity** using many different features such as folder-based routing, where the routes are created automatically depending on the path they are placed in. It also automatically handles optimization by compressing, lazy loading images and even resizing them based on the device size; it also prefetches links and improves ergonomics of working with metadata.

Next.js is developed and maintained by Vercel, a company that also runs their own cloud providing services that integrate with Next.js effortlessly by easily setup an **automatic deployment**. Although, this project does not aim to use their services for hosting purposes, they were still very useful in the iterative development process to see a demo version of the front-end running in production publicly to show to the staff to get feedback. It also automatically

sets up a **CI/CD** pipeline on GitHub, which helped us catching bugs before going to production.

Last but not least, Next.js is now the **most popular framework** of its kind with an incredibly strong community, rich in resources that can be found online and also has **great documentation**. Next.js is therefore rich in resources and libraries. In the context of this work, we will be using the Next.js plugin for PWA support, which is a powerful plug-in library for adding the PWA functionalities with minimal effort.[47]

Therefore, Next.js was the obvious choice as it shows why it is so popular; its ecosystem, resources, support and features are unmatched in the space of React meta-frameworks.[46]

### 2.3.1.2 JavaScript Runtime

JavaScript runtimes have become increasingly more popular over the past years on the back-end due to the comprehensive library support and big community. Node.js has dominated this space for a while, but recently Bun has made big waves in this area disrupting the dominance of Node.js.[48][49]

The decision to utilize Bun as the JavaScript runtime for this project stems from several key advantages.

Bun offers exceptional **performance** with its core built with Zig and leveraging JavaScriptCore (the engine behind Safari), delivers significant performance gains over Node.js. This is especially noticeable in startup times, file I/O, and certain computationally intensive tasks – all relevant to real-time monitoring applications.[50]

Bun aims to **solve many of the pain-points** within the JavaScript world by including its own built-in bundler, transpiler, supporting TypeScript out-of-the-box and a test runner. It also solves the annoying ES modules vs. CommonJS modules conflicts.

Bun has also great documentation that describes many solutions to common problems one can expect to run into.

It has to be mentioned that Bun is a very young technology and therefore it might lack resources, support and libraries for more niche problems. While this is a concern, Bun is compatible with most of the packages built for Node.js and in many ways is not all that different from Node.js.[50]

### 2.3.1.3 Go

When developing high-performance services within our microservice architecture, we have opted to utilize Go as the primary programming language. Go's design by Google engineers to address issues of efficient compilation, execution, and ease of use makes it an ideal choice for back-end services that demand high performance and scalability.

Go is an **extremely simple** and **efficient language**. Its syntax is clean and concise, which makes it easy to read and write. This simplicity accelerates

the development process and reduces the likelihood of bugs. Its compiled nature means it executes code quickly, which is crucial for performance-critical applications. In addition due to Go's simpleness, the Go compiler offers very fast compile times.

Its simplicity reaches also reaches into its support for **concurrent programming** making it an excellent choice for web servers. Concurrency is handled through goroutines, which are lightweight threads managed by the Go runtime; and even these complexities are usually hidden away by a library. This model allows developers to easily implement asynchronous processes.[51]

Go offers a **robust standard library** with extensive support for a wide range of functionalities. Notably the `net/http` package is highly optimized for building web servers and contrary to other language it is commonly used in production. Most of the frameworks for Go stick to the original mantra of Go and are very simplistic. Moreover, many of the web frameworks implement the `net/http` interface to be fully compatible.

While Go's standard library is powerful, we chose to use **Echo** as the web framework to streamline the creation of our REST API for our services. Echo is a high-performance, extensible framework that simplifies many things over the standard library. This choice has been quite hard to make as many of the frameworks in to Go space are similar. While Echo has a smaller community compared to, for example, Gin; it has a better documentation, in our opinion.[26]

#### 2.3.1.4 Python

For the analytical services component of our microservice architecture, Python has been selected as the programming language of choice. Renowned for its strong presence in data analysis, scientific computing, and machine learning.

Python's vast **array of libraries** such as NumPy, Pandas, SciPy, and scikit-learn, simplifies complex data manipulation and analysis tasks. These libraries provide built-in methods for statistical analysis, machine learning, and more, which are essential for the analytical tasks. Furthermore, it can still be integrated with other technologies such as message queues and databases because of its popularity.

Python benefits from a large and active community, which offers extensive resources, tutorials, and forums for troubleshooting. This community support is invaluable for rapid development and problem-solving for analytical purposes.

To efficiently create **REST APIs** for our analytical services, we have chosen FastAPI as the web framework. FastAPI is a modern, fast (high-performance), web framework for building APIs.

FastAPI is one of the fastest web frameworks for Python and even offers an automatic API documentation generation using Swagger.

All in all, Python is a great language for analytical purposes, but lacks

in speed compared to Go and Bun, which leaves it as a great choice for the analytical services.

### 2.3.2 Web Frameworks

We have already discussed the decision process behind choosing the individual web frameworks.

We mainly looked at web frameworks with the biggest popularity for each language and then looked closely at the top of the list. We then made an opinionated judgment based on the mindset behind the framework and its style. We also looked at the quality of the documentation.

Finally, we considered the performance of these frameworks based on different benchmarks. You can see the benchmarks below, firstly from a minimalist benchmark, focusing on purely answering a request by a response.[52]

Language	Framework	Average Latency [ms]			Percentile 90 [ms]			Percentile 99 [ms]		
		64	256	512	64	256	512	64	256	512
Go	Gin	0.89	2.95	5.39	2.41	7.22	12.17	4.87	14.53	25.51
Go	Fiber	0.62	2.00	3.54	1.56	4.86	7.68	4.04	8.63	13.25
Go	Chi	0.90	3.22	5.90	2.41	7.95	13.61	4.82	16.18	29.61
Go	FastHTTP	0.51	1.94	3.44	1.18	4.80	7.61	2.67	8.75	13.56
Go	HTTPRouter	0.88	3.03	5.52	2.39	7.43	12.47	4.86	15.08	27.17
Go	Echo	0.88	3.02	5.51	2.37	7.40	12.46	4.82	15.04	26.55
JavaScript	Elysia	0.86	2.39	4.03	2.36	5.78	8.58	6.13	12.38	16.70
JavaScript	Fastify	1.19	4.26	9.01	1.85	5.86	11.33	4.37	8.92	18.38
JavaScript	Express	2.75	10.68	22.87	3.45	12.94	26.31	9.57	22.32	49.63
Python	Django	17.67	96.98	197.43	35.30	168.47	356.99	47.93	208.00	424.78
Python	Flask	13.77	81.30	169.43	30.42	146.51	308.72	43.86	178.60	392.93
Python	FastAPI	3.95	15.40	32.45	6.27	21.01	41.96	9.57	32.31	73.42

■ **Figure 2.1** Performance benchmarks of the web frameworks we considered. Showing the average latency, P90 and P99 values for 64, 256 and 512 concurrency.

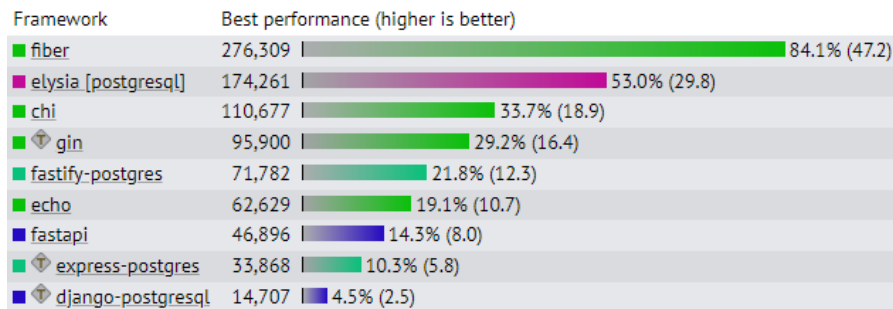
We can see that apart from Elysia, the only Bun framework in the table (the other JavaScript frameworks ran on Node.js runtime), the Go frameworks show a clear dominance, especially Fiber and FastHTTP, but the other frameworks are not too far behind.

Python is clearly the least performant language on this list, which is also shown in the comparison of its web frameworks.

It is important to take these results with a grain of salt, as they are results of a minimalist benchmark which is not simulating any real scenario and the actual real-world performance might be different. These results, however, are relevant as a quick preview of expectations.

Another benchmark focusing on a real-world scenario, by also incorporating other elements such as a database:[53]





■ **Figure 2.2** Performance benchmarks of the same web frameworks showing a score of performance.[53]

We can see similar results to the previous benchmark - the Go frameworks are the fastest along side with Elysia, the only Bun web framework on the list. We can see again that Python's performance is the lowest.

### 2.3.3 Charting library

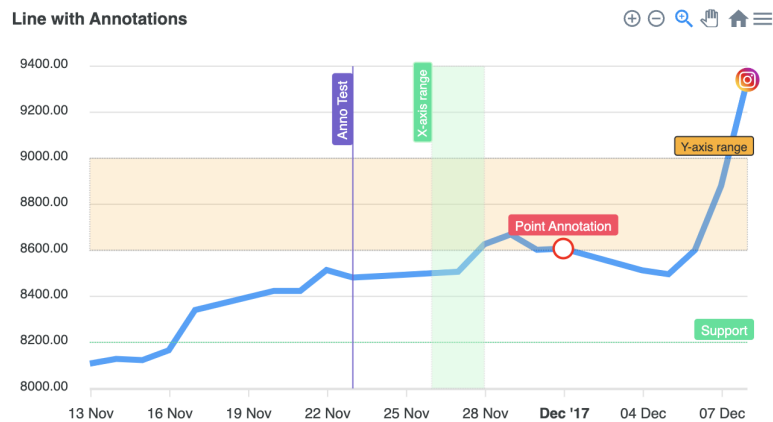
Choosing a powerful and suitable charting library for the front-end of this application is integral as it is one of the main features.

We had several requirements for the charting library:

- Easy integration with React
- Range and value annotations
- Wide range of chart types

We considered several charting libraries such as: Chart.js, Recharts, Nivo, Ant Design, MUI and more.

As it turns out, although most frameworks already support React in some way (be it by being a native React library or by having a React wrapper) and the list of offered chart types is rich, the biggest problem is with range and time annotations.



■ **Figure 2.3** Example of range, x-axis, y-axis and point annotations in a line chart.[54]

As shown on the figure above, these are annotations inside the line chart itself that give additional context. There are many use cases we could use this for within our application:

- X-Axis Range annotation - time segments of some events (heating on/off, cooling on/off, fan on/off, etc.)
- Y-Axis Range annotation - babybox configuration (optimal temperature range, etc.)
- X-Axis annotation - single event (babybox turned on, snapshot sent, etc.)
- Y-Axis annotation - Max/min values (maximum allowable temperature, maximum observed temperature, etc.)
- Point annotation - points of interest (notification generated at that point)

Ultimately, we decided on using ApexCharts for our project as it offers everything we need including the annotations.

ApexCharts supports many different types of charts too: line, bar, area, timeline, radar, radial, heat map, etc. Chart types can be mixed together through the way ApexCharts handles creating series. A series can represent the same data in a unified way for different chart types making them easy to combine together.

Although ApexCharts is not the most popular charting library, it still offers wrapper implementations for different frameworks including React and its documentation is sufficient.

### 2.3.4 Databases

In the development of this work, selecting the appropriate databases was a critical decision that required careful consideration and extensive benchmarking. The system's need to handle a mix of data types, particularly time-series data generated by babyboxes (specifically temperature and voltage snapshots and events) alongside more static data (such as babybox information, user details, and notification templates), necessitated a dual-database approach. This strategy ensures that each type of data is managed optimally according to its specific access patterns and storage needs.

We have done extensive work in understanding the advantages of a time-series database compared to a normal general-purpose database. Keeping in mind our guiding principles, we have decided that the advantages based on the project requirements and the time saved using the time-series database will be worth the effort in increasing the complexity of the system.

We have considered our options aligned with the budget constraints and will explain the decision making process when choosing an open-source database to best align with our needs.

#### 2.3.4.1 Time-series Database

For the development of the system, a comprehensive benchmarking study was conducted to evaluate the most suitable time-series database technologies for managing high-frequency, time-stamped data generated by babyboxes. The findings of this study have been detailed in a paper, which we worked on.<sup>1</sup> This section summarizes the key points from the paper, that focused on both specialized time-series databases and general-purpose databases to ascertain their effectiveness in handling time-series data:[55]

- InfluxDB, TimescaleDB, and QuestDB were evaluated for their specialized time-series capabilities.
- PostgreSQL and MongoDB were included to assess the performance of general-purpose databases when adapted to time-series data.

The benchmarking involved simulating data from IoT devices, typical of what babyboxes might produce:

- Outside temperature - temperature following natural daily fluctuations.
- Heating and Cooling temperatures - primarily stable temperatures with heating and cooling spikes.
- Voltage - primarily stable voltage with failures.

---

<sup>1</sup>The full paper can be found here: <https://github.com/zbyju/timeseries-benchmark/blob/main/paper.pdf>

The performance of each database was tested with different volumes of data and in different categories:

- Data Ingestion: How quickly each database could absorb data.
- Query Performance: Efficiency in retrieving data over last day for a specific station (babybox).
- Data Aggregation: Time of calculating the average over all datapoints.

The benchmarking results highlighted significant differences in performance across the databases:

Insert [s]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	0.08	0.18	0.61	2.79	10.50
PostgreSQL	0.08	0.30	1.22	5.80	23.88
TimescaleDB	0.11	0.35	1.46	6.73	27.48
QuestDB	0.01	0.03	0.22	0.78	3.21
MongoDB	0.06	0.22	0.87	4.16	19.41

Get [ms]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	6.85	6.27	7.29	8.73	9.26
PostgreSQL	0.87	1.24	0.94	1.04	1.54
TimescaleDB	1.39	1.47	1.59	1.31	1.52
QuestDB	5.31	10.55	6.64	7.39	7.36
MongoDB	2.77	3.79	6.36	11.54	55.85

Avg [ms]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	4.30	4.92	6.96	12.20	19.58
PostgreSQL	1.05	1.29	2.13	3.42	6.06
TimescaleDB	1.32	1.24	1.19	1.12	1.28
QuestDB	6.98	9.83	7.51	6.13	5.93
MongoDB	2.31	4.10	8.73	16.60	50.45

■ **Figure 2.4** Benchmark results showing ingestion speeds (top image), query speeds (left image) and aggregation speeds (right image).

- Ingestion Speeds: All the databases using the line protocol (InfluxDB and QuestDB) have performed the best in the ingestion benchmark. SQL database were a bit slower than MongoDB.
- Query Performance: SQL-based systems like TimescaleDB (an extension of PostgreSQL) performed well in query operations, followed by pure time-series databases, while MongoDB was struggling.
- Aggregation Capabilities: SQL-based databases were dominating again with TimescaleDB scaling better than standard PostgreSQL. InfluxDB and QuestDB were doing fine, while MongoDB was scaling the worst.

Given the Babybox Dashboard's primary load on data ingestion due to time-stamped data from babyboxes, databases optimized for high ingestion

rates are preferred. Among the candidates, line protocol-based databases, InfluxDB and QuestDB, stood out for their ingestion capabilities.

QuestDB showed excellent performance in our benchmarks across various metrics but its relative immaturity and limited feature set pose risks for long-term use. In contrast, InfluxDB, as the most established time-series database, offers not only high performance but also robust features and extensive community support.

We would also seriously consider TimescaleDB, if we were using PostgreSQL as our database; as it is an elegant solution working as a plugin for PostgreSQL to improve working with time-series data.

Therefore, InfluxDB has been selected for its proven reliability and suitability for managing high volumes of time-series data efficiently, making it the optimal choice. This decision ensures the system can handle current demands and adapt to future growth.[55]

#### 2.3.4.2 General-purpose Database

In our project, while a time-series database efficiently handles the voluminous data from the babyboxes, the system also requires a robust solution for managing general purpose data. This includes static and structured data such as babybox details, user information, and notification templates. To limit complexity in our system architecture, our goal was to select a single database technology that could effectively manage all these types of general purpose data.[38][37]

The choice between traditional SQL databases and more modern NoSQL databases was central to our architectural strategy. SQL databases, exemplified by PostgreSQL, offer robust data integrity, extensive querying capabilities, and strong transactional support. PostgreSQL, a popular open-source relational database, is renowned for its comprehensive feature set and compliance with SQL standards, making it a strong candidate for applications requiring complex queries and reliable transactions.

However, the relational model of SQL databases imposes certain limitations, particularly in terms of schema flexibility. Schema modifications in SQL databases can be cumbersome and disruptive, potentially slowing down development as the application evolves. This rigidity contrasts sharply with the dynamic and flexible schema capabilities in the world of NoSQL databases, which adapt more readily to changes in data structure and application requirements.

MongoDB, a leading NoSQL database, stores data in a JSON-like document format that naturally supports the hierarchical data structures prevalent in modern web applications. This schema-free model allows each document to have its own structure, which can include arrays and nested documents, providing a highly flexible environment that is ideal for the diverse and evolving data needs.

Furthermore, the schema-free nature of MongoDB significantly accelerates development. Developers can modify the data model on the fly, an important benefit in the fast-paced development cycles, where adaptability to changing requirements is key.[38][37]

### 2.3.5 Middleware

”Middleware is software and cloud services that provide common services and capabilities to applications and help developers and operators build and deploy applications more efficiently. Middleware acts like the connective tissue between applications, data, and users.“[56]

In the architecture, middleware plays a crucial role in facilitating efficient and secure communication between the various components of the system. We have utilized a message broker, API gateway and other utilities that help integrate disparate parts of the system into a cohesive whole. This section explains the functionalities and strategic roles of selected middleware technologies as well as containerization tools, which are instrumental in deploying and managing the system’s infrastructure.

#### 2.3.5.1 Message Broker

In microservice architectures, maintaining a high level of decoupling between services is essential for ensuring system flexibility, scalability, and maintainability. Message brokers play a pivotal role in achieving this by mediating communication paths among services. They facilitate decoupling by allowing services to exchange messages without being directly connected; instead, messages are sent to a common mediator (the broker), which routes them to the appropriate destination based on predefined configurations.

Message brokers usually support many different types of messaging models; we think that all of these models are going to be needed in the future and as such we want to choose a versatile message broker that will support at least these models:

- Message Queues: Standard queues that ensure point-to-point message delivery.
- Publish/Subscribe: A publisher sends messages to a channel without knowing about the subscribers, ensuring wide distribution of messages.
- Request/Reply: Supports synchronous communication patterns typically used for direct service-to-service communication.

While there are several message brokers available, RabbitMQ and Kafka are two of the most prominent in the industry.[57]

**Kafka** focuses on handling large-scale, high-volume data streams. Originally designed by LinkedIn to manage massive amounts of event data, Kafka

serves not only as a message broker but also as a platform for log aggregation. It stores messages in a sequential manner within topics, akin to logs, which can be replayed, enabling historical data analysis and robust system recovery features. This log-centric design is particularly beneficial for applications requiring durable and reliable data processing, such as event sourcing systems where past messages need to be retrieved and processed to restore or replicate application state.

Kafka thrives in a distributing environment where it truly shines, supporting robust data partitioning and replication across multiple nodes in a cluster. This setup ensures high availability and fault tolerance, making Kafka an excellent choice for enterprise-level applications that require resilience at scale. Moreover, Kafka's performance does not degrade with an increase in data, maintaining high throughput rates under heavy loads.

In contrast, **RabbitMQ** is more traditional in its approach, designed to cater to a wide range of messaging needs across various applications. It supports Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (STOMP), and Message Queuing Telemetry Transport (MQTT), which are standards for integrating heterogeneous system components. Unlike Kafka, RabbitMQ excels in scenarios requiring complex message routing, load balancing, and quick message delivery without the need for long-term data storage or replay capabilities.

RabbitMQ provides flexibility through its support for multiple messaging patterns, including not just basic point-to-point and publish/subscribe models, but also sophisticated routing capabilities that can direct messages based on content and other headers. This makes it particularly useful for applications where delivery order, priority management, and routing complexity are critical.

For this project, RabbitMQ was chosen primarily for its simplicity and immediate alignment with the system's operational requirements. While Kafka offers extensive capabilities around data retention and distributed processing, these features introduce complexity in terms of setup, management and operational overhead—factors that have very little to no upsides. It also does not offer the same versatility as RabbitMQ does when it comes to messaging models.

RabbitMQ's ease of deployment, rich feature set, and support for diverse messaging models make it well-suited for applications that do not require Kafka's robust data logging and replay functionalities. Additionally, RabbitMQ's compatibility with many programming languages enhances its versatility, allowing seamless integration across different software environments. Furthermore, RabbitMQ's management interface provides straightforward monitoring and troubleshooting tools, which are invaluable for maintaining system health and performance without the need for extensive configuration or specialized expertise.[57]

### 2.3.5.2 API Gateway

In the architecture, the API gateway is a critical component, acting as the entry point for all client requests to the backend services. It simplifies the complexity of interacting with microservices by providing a single, unified API interface to external clients. Moreover, it handles a variety of cross-cutting concerns such as authentication, SSL termination, rate limiting, and load balancing.[23][24]

For our system, the choice of the right API gateway was influenced by several factors including simplicity, automatic HTTPS support, and ease of integration with our existing infrastructure.

**Caddy** has been part of our infrastructure for some time and is primarily used as a reverse proxy. It stands out for its simplicity and minimal configuration requirements. One of Caddy's hallmark features is its support for automatic HTTPS, which effortlessly handles SSL/TLS certificate issuance and renewal through Let's Encrypt. This eliminates many of the manual steps typically associated with certificate management and ensures that our communications are always secured without additional overhead.[58]

Given its proven reliability and ease of use, Caddy was a natural choice for deployment in the production environment. It effectively manages incoming requests and routes them to the appropriate services, while also dealing with HTTPS, thereby enhancing security and simplifying the operational aspects of our infrastructure.[58]

**Traefik**, on the other hand, was considered for its robust integration capabilities with Docker and Docker Compose. Traefik uses labels specified in the Compose files to automatically discover services and configure routing rules, which significantly simplifies the process of dynamically adding or removing services without manual gateway configuration.[23]

Despite its advantages, the complexity and management overhead of using Traefik in production while also using Caddy for our other projects and applications led us to reserve Traefik for development use only. Its powerful features make it an excellent candidate for scaling scenarios where automatic service discovery and more granular load balancing capabilities become necessary.[23]

### 2.3.5.3 Docker and Docker Compose

From the onset of designing Babybox Dashboard, the utilization of Docker was a foundational decision. Docker's containerization technology and the microservice architecture goes hand-in-hand and enable each other. Docker helps encapsulating microservices into manageable units but also for ensuring consistency across various environments. The inherent isolation that Docker provides allows each microservice to be packaged with its dependencies, eliminating the common development pitfall of "it works on my machine". This discrepancy is often due to environmental differences that Docker containers mitigate by maintaining uniformity from development through to production.



In the development environment, **Docker**'s role extends beyond merely avoiding environmental discrepancies. Containers can be used to spin up instances of each microservice independently with configurations tailored for development, which may include additional debugging tools or configuration settings not used in production.

Docker can also be used for seamlessly setting up a production and development environments. Development environments tend to have different requirements such as being able to run each service in isolation for debugging and also enabling hot-reloading for quick development. Production environment should create production ready builds for performance and security, stripped of unnecessary packages.

We also decided to use **Docker Compose** which further enhances Docker's capabilities by allowing for the definition, orchestration, and scaling of multi-container applications through simple YAML configuration files. This is particularly advantageous when dealing with microservices, as it allows for:

- Clear service definitions: Each service is defined with its environment variables, volumes, and other dependencies clearly stated, simplifying setup and replication.
- Network configuration: Docker Compose manages the internal networking through which containers communicate with each other. This network isolation simplifies network management and enhances security by controlling inter-service communication paths.
- Simplified deployment: With Docker Compose, the entire application stack can be deployed using a single command, ensuring that all services are launched in the correct order with the proper configurations.

## 2.4 System Design

Babybox Dashboard, initially developed with a functional yet rigid architecture, faced increasing challenges in mainly flexibility and maintainability. As the demand for more sophisticated functionalities and the need for system expansion became apparent, it was crucial to transition towards a more robust, future-ready solution. This transition aimed to address not only the immediate inefficiencies but also to lay a foundation that would support the system's growth and adaptability in the face of evolving technological trends and user requirements.

This section of the thesis delineates the process of redesigning the system architecture. It begins with an evaluation of the previous solution, highlighting the core limitations that necessitated a redesign. Following this, the proposed solution is detailed, describing the new architectural design, the rationale behind the design choices made, and the integration of modern design principles and patterns. The section concludes with a comprehensive presentation of the

domain model, which organizes and defines the system's primary entities and their interactions, ensuring alignment with the newly adopted architectural strategies. This structured approach ensures a holistic understanding of the system's design evolution and its alignment with long-term strategic goals.

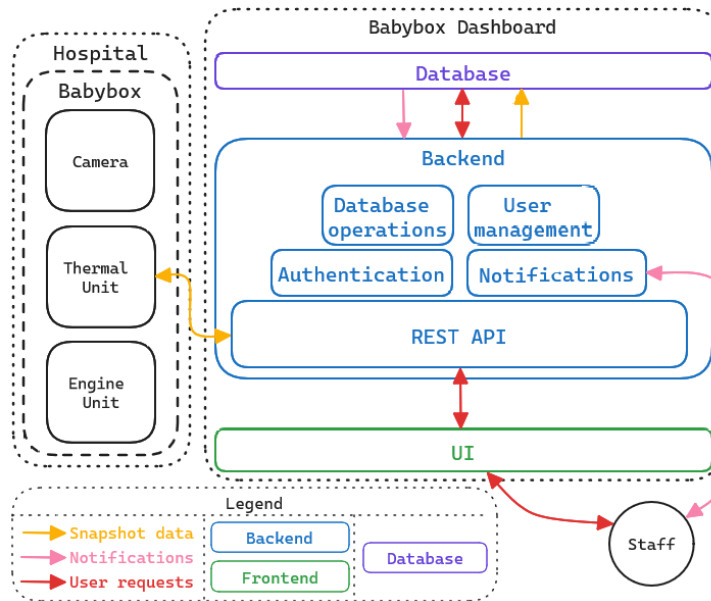
### 2.4.1 Previous Solution

Babybox Dashboard was initially developed to provide real-time monitoring and management capabilities for babyboxes. Its primary functions included tracking operational status, managing alerts and notifications, and administering user and device configurations. The dashboard enabled users to visualize data trends, receive updates on babybox conditions, and respond to alerts quickly, playing an integral role in ensuring the safety and effectiveness of the babybox service.

From a technical standpoint, the system utilized a combination of Vue 2 for the frontend and Node.js with Express for the backend. Vue 2 was selected for its reactivity and component-based architecture, which facilitated the development of a dynamic user interface. The backend, powered by Node.js and Express, handled all server-side logic including data reception, user authentication, and notification processing.

The entire data management was centralized in MongoDB, which stored everything from user information to time-series dataFlex. This setup allowed for straightforward data operations but lacked the flexibility to optimize storage and queries tailored to the distinct characteristics of time-series and relational data.

The application infrastructure was not containerized, relying instead on PM2 for process management. PM2 is a process manager for Node.js applications that provides an easy way to manage and daemonize applications. It offers features such as automatic restarts, and log management, which helped in maintaining application uptime and simplifying deployment processes.



■ **Figure 2.5** Data flow diagram of the previous solution, where the blue Back-end component is the one big monolith.

The initial design of the application, created as a monolithic application, posed significant challenges to system expansion and feature enhancement. Given the monolithic structure, any modification or addition of features required scaling or updating the entire system, which was not only resource-intensive but also prone to increasing the potential for errors. Additionally, the early stages of development suffered from inexperience, resulting in a codebase that wasn't optimized for easy updates or future expansion. This lack of foresight in design made it cumbersome to integrate new functionalities or adapt to evolving user needs and technological advancements.

As Babybox Dashboard continued to operate, the frameworks and libraries it relied upon aged—some becoming obsolete and others undergoing major updates that the existing system architecture could not seamlessly integrate without significant refactoring. For example, transitioning from Vue 2 to a more current version represented not just a simple update but a comprehensive overhaul of the frontend codebase.

The monolithic architecture significantly limited the system's flexibility. Different parts of the application, such as user interface rendering, data processing, and notification management, would ideally benefit from specific technologies best suited to their operational requirements. For instance, handling time-series data from babyboxes could be more efficiently managed by a specialized time-series database rather than the general-purpose MongoDB used across the system. More importantly, it would be beneficial to use different programming languages and libraries for different features. For example, using Python and its ecosystem of libraries would be beneficial in the more analytical

parts of the application. The inflexible nature of the monolithic design made it difficult to implement the most effective technological solutions for individual components, thereby hampering overall system efficiency.

The infrastructure supporting the system was minimally set up, lacking many components that could facilitate development and deployment efficiency. The absence of containerization meant that replicating production environments for testing was fraught with inconsistencies, leading to the “it works on my machine” syndrome. Additionally, the lack of a CI/CD pipeline, insufficient documentation, and inadequate testing practices further complicated the development process and increased the maintenance overhead. The system’s setup did not leverage modern development practices like Docker, which could streamline environment management and improve deployment reliability.

The decision to undertake a complete rewrite of Babybox Dashboard was driven by these identified issues. The rewrite will transition the system from a monolithic architecture to a microservices-based architecture, enhancing flexibility by allowing individual services to use the most appropriate technologies and databases. This modular approach will facilitate easier updates and quicker integration of new features. Additionally, introducing containerization through Docker will standardize development and production environments. This comprehensive overhaul is aimed at making the system more adaptable, maintainable, and capable of meeting current and future needs more effectively.

### 2.4.2 Proposed Solution

The redesigned solution aims to address the shortcomings of the previous monolithic architecture while leveraging insights gained from past design, implementation experiences and observing users interactions. The proposed solution adopts a microservices architecture, which offers significant improvements in system flexibility, maintainability, and scalability. This architectural shift is designed to mitigate the issues inherent in the original setup, such as difficulty in expanding features, challenges in maintenance, and the inflexibility of integrating optimal technologies for specific tasks.

In the new architecture, the application will be decomposed into several smaller, loosely coupled services. Each service will be responsible for handling a distinct aspect of the system, allowing for independent scaling, development, and maintenance. This modularity enables the selective use of the most appropriate technologies and databases tailored to the specific needs of each service.

To effectively support this diverse ecosystem, the solution will utilize a mix of programming languages and technologies that are best suited for the tasks at hand. Go will be employed for services requiring high-performance and efficient concurrency management. TypeScript, with its robust typing system, will be used to enhance the development of the user interface and other front-end components. Python will be utilized for services that benefit from its extensive library support, particularly for data analysis and possibly machine learning

tasks. On the database front, InfluxDB will manage time-series data from the babyboxes, optimizing for high write throughput and efficient querying of time-stamped data, while MongoDB will continue to handle more general data storage needs.

This strategic use of multiple technologies and the transition to microservices are geared towards creating a maintainable, flexible, and robust platform. This approach not only facilitates easier updates and maintenance but also prepares the system for future expansion and the integration of new technologies as they emerge. The following sections will delve deeper into the specifics of this architectural design, the individual components, and the overarching design principles that guide the development of the new system.

#### 2.4.2.1 Transition to a Microservice Architecture

The transition from a monolithic architecture to a microservice-based architecture is a pivotal shift in how the application is structured and managed. This transformation capitalizes on the clarity of established requirements and a well-defined product scope, which significantly aids in delineating the capabilities around which the microservices are designed. Each service in the new architecture is aligned around a specific function, enhancing the system's flexibility, scalability, and maintainability.

The process of decomposing a monolith into microservices involves several challenges that must be carefully managed:

- **Overly Granular Services:** Excessively small services can lead to an inflated number of components to manage, which might complicate the architecture and increase overhead. Such granularity can also dilute the focus of services, making the system harder to maintain.
- **Oversized Services:** Conversely, if services are too large, they may retain some of the monolithic architecture's drawbacks, such as tight coupling and reduced flexibility. These large services can hinder the independent scalability and resilience that microservices aim to provide.

To ensure a smooth and effective transition to a microservices architecture, it is essential to meticulously address a variety of key aspects that are critical for laying a solid foundation for this complex system structure:

- **Service Boundaries:** Establishing clear boundaries based on distinct business capabilities is critical. This ensures that services are cohesive and maintain loose coupling with others, allowing for independent operation and easier scalability.
- **Data Management:** While each microservice owning its database is ideal for ensuring loose coupling, the complexity of managing multiple databases would not suit this project. In this redesigned architecture, each service

manages its distinct set of database collections (or tables, buckets, etc.) within the database, which prevents interference with other services and is a step towards database-per-service architecture without full implementation complexity.

- **Communication Strategies:** Implementing robust communication mechanisms, such as asynchronous messaging through RabbitMQ, supports service autonomy and enhances the resilience of the system against failures.
- **Technology and Language Diversification:** The use of different technology stacks for different services is encouraged to optimize performance based on service requirements. However, limitations need to be established to prevent excessive diversification, which could lead to a fragmented system that is hard to manage. Keeping the technology stack consistent where possible—using a limited set of languages and frameworks—helps maintain system coherence and manageability.

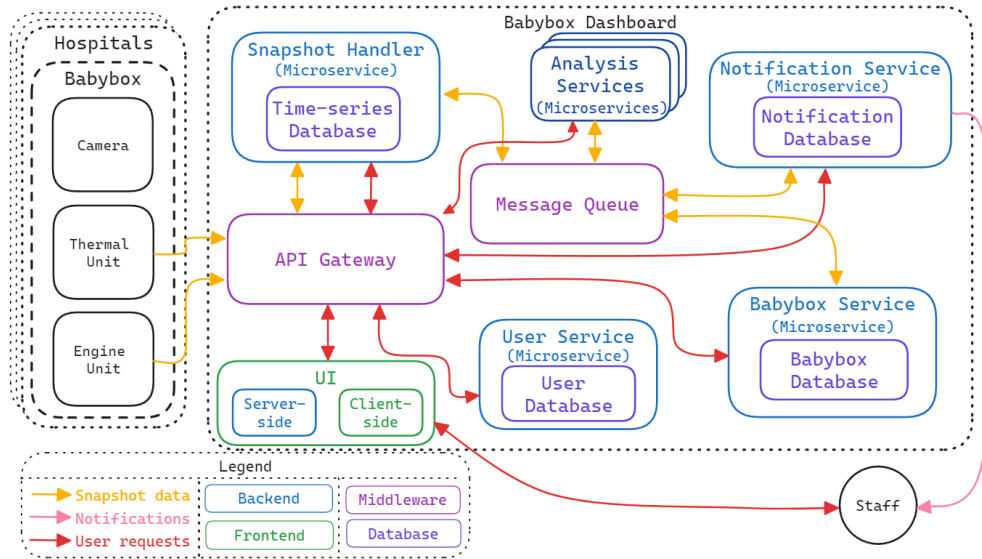
Transitioning to a microservice architecture as a single developer presents unique challenges, particularly in managing the system’s complexity effectively. Instead of adopting a typical multi-team approach, the strategy revolves around simplifying the system design to ensure manageability while still delivering all required functionalities. Each microservice is carefully designed with clear interfaces and assigned responsibilities, significantly reducing the cognitive load and making the overall system easier to understand and manage.

RabbitMQ plays a critical role as the communication hub, streamlining the interactions between services. This setup not only simplifies the messaging patterns but also centralizes message handling, enhancing the system’s reliability and scalability. Furthermore, the integration of decentralized technologies like JSON Web Tokens (JWT) for security empowers each service to independently manage authentication and authorization without the need to communicate with any other service. This decentralization enhances service autonomy, secures interactions, and facilitates scalable expansion as the system evolves.

The careful planning and implementation of this microservice architecture aim to fully leverage the benefits of this architectural style while consciously addressing its complexities. By clearly defining service boundaries, optimizing data management, and curating a limited yet effective technological palette, the architecture is designed to be robust, scalable, and adaptable. This strategic overhaul isn’t just about technological advancement but about laying a sustainable and evolvable foundation that supports ongoing improvement and accommodates future growth, ensuring the system remains effective and responsive to evolving needs.

### 2.4.2.2 Architectural Design

The architectural design of the updated system leverages a pragmatic infrastructure to manage the routing and processing of various data types. This setup ensures that data from different sources is handled efficiently through designated pathways, as depicted in the system design diagram.



■ **Figure 2.6** Data flow diagram of the new microservice solution.

**Snapshot Data Flow:** Snapshots and event-related data are continuously generated by babyboxes and transmitted directly to an API gateway. This gateway routes the incoming data to the Snapshot Handler microservice, which is optimized for processing time-series data. The microservice then stores this data in InfluxDB, suited for high write throughput and fast querying capabilities essential for time-series data management. Additionally, this data is also published to RabbitMQ, which distributes it across all subscribed services via fan-out exchanges, facilitating data analysis and further processing.

**User Interaction with the Frontend:** The frontend interacts with the system either through server-side or client-side rendering. In both cases, user requests for data are routed through the API gateway to the appropriate microservices' REST APIs.

**Notification Data Flow:** Notifications are generated by a dedicated Notification Service that triggers emails to users based on specific conditions or events detected within the data. This service forms an essential communication link between the system's operations and the user, ensuring timely updates on critical events.

The use of an API gateway centralizes the entry points for all incoming data and requests, simplifying the management of data flows across the system's

landscape. RabbitMQ plays a pivotal role in decoupling data producers from consumers, providing a robust messaging framework that enhances the overall responsiveness and efficiency of the system.

It should be pointed out, that the architecture suggests each microservice owns its own database. The diagram more so tries to symbolically illustrate that it is indeed the microservice who owns its data (even if they are stored in a shared database). More importantly, the diagram is supposed to convey the idea that the concrete placement of the data is not relevant as it is a subject to change if there is the need.

The architecture described meets the current functional requirements of the system but also supports efficient maintenance and management practices. We also think that it is a robust solution for future expansions.

### 2.4.2.3 Design of Individual Components

The Babybox system utilizes a collection of microservices, each designed to handle specific functional scopes efficiently. Here's a detailed look at each service, outlining their roles and the expected load based on system usage patterns.

- **Snapshot Handler** (microservice)
  - **Functional Scope:** This service processes and stores incoming data from the babyboxes, including snapshots and event notifications. It ensures data is validated, formatted, and persisted in a time-series database for real-time and historical analysis. The Snapshot Handler also publishes this data to a message broker to facilitate asynchronous processing by other services and provides data access through a REST API.
  - **Expected Load:** High, due to the continuous and simultaneous influx of data from approximately 100 babyboxes.
- **Babybox Service** (microservice)
  - **Functional Scope:** Manages comprehensive information about each registered babybox, such as location, contacts, and network settings. This service is essential for operational management and exposes babybox details through a REST API.
  - **Expected Load:** Medium, reflecting the less frequent updates and but frequent queries.
- **User Service** (microservice)
  - **Functional Scope:** Handles user management including registration and authentication by issuing tokens, which are used to check for authentication in other components. It exposes its data through a REST API.



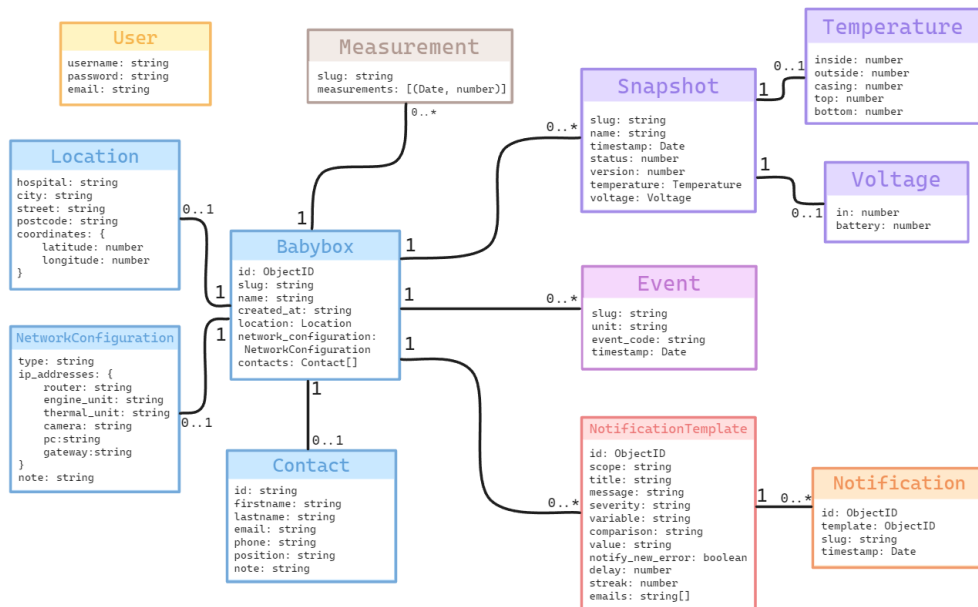
- Expected Load: Low, given the infrequent user authentication events relative to data processing activities and speed is not critical.
- **Notification Service** (microservice)
  - Functional Scope: Analyzes data from babyboxes to generate notifications based on specific triggers. This service manages notification templates and the delivery of alerts, accessible via a REST API.
  - Expected Load: Low, as notification generation is event-driven and only occurs in response to particular data conditions.
- **Other Analysis Microservices**
  - Functional Scope: These services perform targeted analyses on subsets of data relevant to their specific functions. They operate independently, providing specialized data processing and interfacing with other system components through REST APIs.
  - Expected Load: Variable, typically low to medium, depending on the complexity and frequency of the required analyses.
- **Front-end**
  - Functional Scope: Serves as the user interface of the system, capable of running code both server-side and client-side to provide dynamic data visualizations and user interactions. It fetches and displays data necessary for users to oversee and interact with various aspects of the system.
  - Expected Load: Medium, due to most computation being done client side anyway.
- **API Gateway**
  - Functional Scope: Acts as the central point for all incoming and outgoing HTTP/HTTPS requests. It functions as a reverse proxy, directing requests from users to the appropriate microservices, thereby facilitating seamless interactions across the system.
  - Expected Load: High, as it manages all network traffic, ensuring secure and efficient data transmission to and from the microservices.
- **Message Broker**
  - Functional Scope: Operates as the middleware that facilitates communication between microservices. It ensures that updates and new data/events are consistently propagated throughout the system, enabling services to respond to real-time changes.
  - Expected Load: High, tasked with robustly handling a vast volume of messages and maintaining low latency in data distribution.

As we have discussed, we will approach each service based on the methodology and mindset we have described and having a clear understanding of its functional scope and load is going to play a major role.

By designing each component with a clearly defined role and aligning their capabilities with the anticipated operational demands, the system stays maintainable and predictable. Each service is optimized to handle its assigned tasks effectively, ensuring the architecture supports both current needs and future growth.

#### 2.4.2.4 Domain Model

The domain model provides an abstract representation of the data entities and their relationships. It is important to note that this model does not directly correspond to the actual database schemas but rather offers a conceptual overview, which helps in understanding how data is organized and managed across the system rather than depending on the implementation and technological details.



■ **Figure 2.7** Domain model

#### ■ **Babybox Entity:**

- **slug:** Unique identifier, a string of lowercase letters with spaces replaced by hyphens.
- **name:** A human-readable name for the babybox, initially the same as the slug but can be changed by the user.
- **created\_at:** Timestamp marking the creation of the babybox record.

- `contact_information`: Includes `firstname`, `lastname`, `phone`, `email`, `position`, and a `note`.
  - `network_configuration`: Captures the `type` of network setup, `ip-addresses` of key components (`engine_unit`, `thermal_unit`, `camera`, `router`, `gateway`), and a `note`.
  - `location`: Details the hospital name, `city`, `street`, `postcode`, and geographical coordinates - `latitude` and `longitude`.
- **Snapshot Entity:**
- `slug`: Associates the snapshot with a specific babybox.
  - `timestamp`: The precise moment the snapshot was captured.
  - `status`: Indicates the operational status, with 0 for normal and other values indicating issues.
  - `temperature` measurements: Specific readings from `inside`, `outside`, `casing`, `top`, and `bottom` of the heat exchanger.
  - `voltage` measurements: Includes voltage inputs from the power supply (`in`) and `battery`.
- **Event Entity:**
- `slug`: Links the event to a babybox.
  - `unit`: Specifies the unit from which the event originated.
  - `event_code`: Categorizes the event type.
  - `timestamp`: Records the time of the event occurrence.
- **Notification Template Entity:**
- `id`: Unique identifier for the template.
  - `scope`: Defined per babybox (`slug`) or globally (`"global"`).
  - `title`: User defined name of the notification.
  - `message`: User defined message attached to each notification.
  - **Condition Setup**: Comprises of a `variable` (e.g., `temperature.inside`), a `comparison` operator (e.g., `<`), and a `value` that together define the trigger condition.
  - `severity`: Categorized as `low`, `medium`, or `high`, indicating the urgency of the notification.
  - `notify_new_error`: Boolean flag to immediately trigger notifications for new issues.
  - `delay`: Time interval to wait before issuing another notification for the same event.

- **streak:** Number of consecutive data points that meet the condition before a notification is issued.
- **emails:** Recipients of the notification.
  
- **Notification Entity:**
  - **id:** A unique identifier of the notification.
  - **template:** Reference to the notification template.
  - **timestamp:** Time at which the notification was generated.
  - **slug:** Identifier for the babybox associated with the notification.
  
- **User Entity:**
  - **username:** Unique identifier for each user.
  - **email:** User's email address.
  - **password:** Hashed password for user authentication.
  
- **Measurement Entity:**
  - **slug:** Unique identifier for the babybox from which the measurement was taken.
  - **measurements:** Array of tuples containing the timestamp and the battery voltage over time.

# Implementation

*In this chapter, we delve into the practical application of the theoretical concepts and architectural design choices discussed earlier, focusing on the actual implementation of the Babybox Dashboard system. This includes a detailed examination of the system's infrastructure, the microservices themselves, and the front-end part of the application, providing insights into both the technical challenges and solutions employed throughout the development process.*

## 3.1 Implementation Process

The implementation of Babybox Dashboard was approached methodically to ensure a robust and user-centric solution. Here's an overview of the process:

1. Analysis and Technology Selection:
  - The project began with an in-depth analysis phase, where requirements were gathered and use cases were defined. This stage was critical for understanding the specific needs and expectations of the users.
  - Simultaneously, technology selection was carried out to identify tools and frameworks that would best meet these requirements. This involved choosing suitable databases, backend frameworks, and front-end technologies.
2. System Design:
  - With a clear set of requirements and technologies selected, the next step was to design the system architecture. This provided a blueprint of how the application components would interact and laid the groundwork for the implementation phase.
3. Initial Service Implementation:

- Implementation kicked off with the development of the snapshot handler service. This service was prioritized to start gathering data early in the project lifecycle, ensuring that by the time other parts of the application were developed, there was already meaningful data to work with.
4. Front-End Development and Testing:
    - Attention then shifted to developing the front-end to ensure the application was user-friendly and efficient. The initial user interface was crafted and deployed using Vercel's cloud platform for quick testing and iteration.
    - This approach allowed for continuous user feedback, which was integral to refining the UI. The design-first approach for the front end ensured that once the user interface was set, the backend could be tailored to support the exact data needs and functionalities required by the front end.
  5. Development of Additional Services:
    - With the front-end established and the snapshot handler in place, development proceeded with other critical services: the Babybox service, User service, and finally, the Notification service.
    - Each service was developed in sequence, with ongoing iterations based on user feedback. This iterative process helped in fine-tuning each service according to real user interactions and requirements.
  6. Iterative Development and Feedback:
    - Throughout the development process, the project benefited from being agile and responsive to feedback. Regular interactions with potential end-users helped in refining features and ensuring the system met practical needs.
    - The development was a solo effort, which underscored the importance of structured planning and self-reliance. Managing the entire stack single-handedly highlighted the need for a highly organized approach to both development and testing.

## 3.2 Infrastructure

The infrastructure was designed to avoid excessive complexity while prioritizing maintainability and future expandability. This approach ensures that the system remains manageable and adaptable, accommodating changes and growth over time without the risk of becoming cumbersome. The selection and integration of key technological components form the core of this infrastructure, supporting the system's operational needs and aligning with current standards in software development.

Included in this robust yet straightforward infrastructure are several essential components. **Containerization** is implemented through Docker and Docker Compose, which encapsulate each service within its own environment, ensuring consistent deployments across development and production environments.

**RabbitMQ** acts as the message broker, facilitating asynchronous communication between microservices, which enhances responsiveness and decouples system components.

**MongoDB** is utilized for its schema-less design, offering the flexibility required in dynamic environments often encountered in modern web applications.

Lastly, **InfluxDB** is incorporated for its specialized handling of time-series data, pivotal for effectively monitoring and analyzing performance metrics from Babybox units.

The whole application is deployed on a single node on our own server. This choice is driven by the convenience and control offered by managing our server, which is already employed for other projects, and the desire to avoid dependency on external cloud services.

Additionally, we have established a development environment that runs directly on the developers' machines. This setup mirrors the production environment closely. This approach not only accelerates development cycles but also minimizes the discrepancies between development and production setups, enhancing the reliability and predictability of new releases.

### 3.2.1 Code Structure

In the development of the microservice architecture, we considered employing a **monorepo** approach to centralize source control and streamline development processes. A monorepo facilitates simplified dependency management, as updates are applied uniformly across all services, and enhances collaboration by standardizing tools and scripts. Additionally, this configuration can significantly ease development operations and integration with CI/CD pipelines.

However, monorepos are not without challenges. The growth of the repository can complicate codebase management, potentially slowing down build processes and complicating navigation. Furthermore, the risk of inadvertently introducing tight coupling between services increases with shared dependencies, and complex branch management can further complicate version control.

#### 3.2.1.1 Nx

To manage these complexities within a monorepo, we evaluated Nx, a powerful tool that extends beyond JavaScript and TypeScript to support a diverse range of technologies including Go and Python. Nx is particularly appealing for its features like generators, which automate repetitive tasks and ensure uniformity

in how certain operations—such as service scaffolding—are executed across different parts of the codebase. This capability not only saves time but also enforces consistency in the development process.

Despite these advantages, implementing Nx was not straightforward. The tool introduced a steep initial learning curve and integrating it into our existing workflow presented several challenges. Additionally, we encountered specific issues with an Nx plugin for Go which was undergoing a change in maintainers and was not operational at the time, further complicating its adoption.

Given these hurdles, and considering the overhead of introducing and maintaining Nx in our development environment, we decided against its immediate adoption. We recognized that while Nx could potentially help us in development and enforce best practices across services, the immediate costs and complexities did not justify the integration. However, the flexibility of our architectural approach allows us to revisit this decision in the future should our needs or the stability and features of tools like Nx change.

### 3.2.1.2 Current Setup

Currently, our project structure involves a monorepo with a `/apps/` folder containing individual microservices. This setup maintains a clear separation of concerns, with each service operating independently and managing its own domain logic and data. This decoupling ensures that the lack of shared codebases does not adversely affect system functionality, and the system remains adaptable for potential future integration of tools like Nx to enhance development practices.

The code structure of the services themselves follows idiomatic approaches. In Go, we heavily utilized the `internal` folder, which has a special meaning which makes sure that there is no tight coupling between services as that code can only be imported internally within each respective service.

Microservices written in Bun and Python also follow their idiomatic approaches; this way we believe that orientation in the codebase becomes much easier.

### 3.2.2 Containerization

In the system, containerization plays a pivotal role not just in ensuring application consistency across environments but also in enhancing the development workflow. The project utilizes Docker and Docker Compose as key tools to improve the developer experience by enabling features like hot reloading and simplifying the setup process, allowing the entire environment to be launched with a single command.



### 3.2.2.1 Docker

The system employs two types of Dockerfiles tailored for different environments—development and production:

- Development Dockerfiles (`Dockerfile.dev`): These are designed to maximize developer productivity by enabling hot reloading. This feature allows the system to automatically reload and apply changes in real time as developers modify the code, significantly speeding up the development process. The setup varies by the technology stack:
  - Go: Utilizes *Air* for hot reloading.
  - Bun: Runs bun with the `--watch` option.
  - Python: Leverages *Uvicorn* with the `--reload` flag.
- Production Dockerfiles (`Dockerfile`): These are optimized for performance and stability. In this configuration:
  - Go: The application is compiled into a streamlined binary.
  - Bun: The application is transpiled into a production-ready format.
  - Python: Uvicorn runs without the hot reload feature, enhancing performance.

Each Dockerfile follows a structured approach to build the Docker images:

```
1  # Start with a base image for the specific service
2  FROM base-image
3
4  # Set the working directory within the container
5  WORKDIR /app
6
7  # Copy over the file that lists project dependencies
8  COPY dependency_definition_list dependency_definition_list
9
10 # Install the project dependencies
11 RUN install_dependencies
12
13 # Copy the rest of the application's source code
14 COPY . .
15
16 # Command to run the application
17 CMD ["run", "application"]
```

■ **Code listing 3.1** Pseudo-Dockerfile showcasing the basic structure of the Dockerfiles in this application.

### 3.2.2.2 Docker Compose

Docker Compose is used to orchestrate the entire suite of services, ensuring each component interacts seamlessly with others while maintaining isolation:

- Production (`docker-compose.yml`): Specifies paths to production Dockerfiles and sets up the operational environment, including dependencies, network settings, and environment variables necessary for the production setup.
- Development (`docker-compose.dev.yml`): Overrides the Dockerfile paths to use development versions and can modify other settings to tailor the environment for development needs.

To launch the system:

- For production: The command: `docker compose up --build` is used. It builds and starts services based on the production configurations.
- For development: By running: `docker compose -f docker-compose.yml -f docker-compose.dev.yml -p babybox-dashboard-dev up`, the system combines the base Docker Compose configuration with the development Compose file by overriding the common configuration found in both files, which leads to a development environment spinning up equipped with hot reloading and other developer-friendly features.

This dual approach in Docker Compose setup not only speeds up the development and deployment processes but also ensures that the environments are optimally configured for their respective purposes, development or production, thereby supporting the project's goals of maintainability and future expandability without introducing unnecessary complexity.

### 3.2.2.3 Environment Variable Management

Environment variables are centrally managed in a `.env` file, enhancing security and simplifying configuration changes. Docker Compose is configured to inject these variables into services as needed:

```
1  environment:
2    - APP_ENV=production
3    - JWT_SECRET=${JWT_SECRET}
4    - INFLUXDB_URL=http://influxdb:8086
```

■ **Code listing 3.2** Configuration a service through environment variables in a Docker Compose file.

In this example, `APP_ENV` and `INFLUXDB_URL` are going to be statically set; `JWT_SECRET` is going to be assigned the value from the `.env` file.

This method ensures that all services have access to the configurations they require without hardcoding sensitive information into the codebase or Docker configurations.

Furthermore, to not leak any sensitive information such as secrets or passwords we do make `.env` file public by pushing it into a GIT repository; instead we push a `.env.example` file, which makes it clear which variables need to be set without exposing the values.

#### 3.2.2.4 Container Dependencies

One challenge with Docker Compose is its `depends_on` functionality, which does not guarantee that a dependent service is fully operational before starting services that rely on it. This can cause issues, particularly when a service expects a database or RabbitMQ to be available immediately. Although Docker Compose supports automatic restarts on failure, this is not always effective in development environments with hot reloading as the software providing the hot reloading functionality does not actually shut down but waits for changes to restart the service, meaning Docker will not trigger the restart.

To address this, each microservice includes an initialization process that retries connections to dependencies several times with delays if necessary. This approach effectively mitigates issues related to services starting before their dependencies are fully operational.

### 3.2.3 API Gateway

In the infrastructure, access to back-end services is managed differently in development and production environments, each tailored to meet the specific needs of these scenarios efficiently.

#### 3.2.3.1 Development Environment

In development, we use Traefik as an API Gateway to provide communication between the services and the front-end. This is done so that the developer environment is more similar to the production environment. For other types of communication, such as service-to-middleware communication, we just use the docker network.

A MongoDB service running in a container named `mongodb` on port 27017 is accessible via `mongodb:27017`. A service like `snapshot-handler` listening on port 8080 can be reached through `http://snapshot-handler.localhost` from the front-end as the requests are passed through Traefik. This method is straightforward and sufficient for development purposes, where simplicity and speed of setup are crucial.

We have also heavily considered Traefik as our API Gateway in production as it is a popular choice for handling reverse proxy tasks with automatic HTTPS management and is particularly well-suited for dynamic microservice environments. Traefik's configuration typically involves either using a configuration file or, when used with Docker Compose, setting labels, like:

```
1 labels:
2   - traefik.enable=true
3   - traefik.http.routers.snapshot-handler.entrypoints=websecure
4   - traefik.http.routers.snapshot-handler.rule=Host(`snapshot-handler.domain.com`)
5   - traefik.http.services.snapshot-handler.loadbalancer.server.port=8080
```

■ **Code listing 3.3** Configuring snapshot-handler service for Traefik using labels.

These labels allow Traefik to automatically detect services and manage traffic routing based on the specified rules. However, given the existing infrastructure and the usage of Caddy across other projects, introducing Traefik was deemed to potentially add unnecessary complexity. We therefore opted to leave Traefik for development only for now.

### 3.2.3.2 Production Environment

For production, the approach is more structured to ensure security, reliability, and scalability. We use **Caddy** as a reverse proxy to manage requests from the internet to the internal services efficiently. Caddy stands out for its simplicity and automatic HTTPS setup, which secures communication by default.

The configuration of Caddy involves specifying routes and target services straightforwardly. For example, to route traffic intended for a service-handler to the correct backend service, the Caddyfile would include:

```
1 snapshot-handler.domain.com {
2   reverse_proxy IP:PORT
3 }
```

■ **Code listing 3.4** Configuring snapshot-handler service in Caddyfile.

This setup ensures that any request to `https://snapshot-handler.domain.com/` is appropriately directed to the internal service running at the specified IP and port.

By keeping the development environment simple with Traefik and Docker's networking capabilities and leveraging Caddy's straightforward and secure reverse proxy functionality in production, the system achieves a balance of ease

of use, security, and consistency across its deployment environments. This approach ensures that both the development and production setups are optimized for their respective needs without compromising functionality.

### 3.2.4 RabbitMQ

We used RabbitMQ as the backbone for asynchronous communication between different services, particularly in handling and distributing events related to babybox data, such as newly received snapshots. The use of RabbitMQ ensures that services can react to events in real-time without being tightly coupled to each other, enhancing the system's scalability and responsiveness.

#### 3.2.4.1 Docker Compose Configuration

RabbitMQ is configured within Docker Compose, leveraging environmental variables defined in the `.env` file for setting credentials. It operates on the default port 5672 for messaging operations, with the management dashboard accessible on port 15672. This setup allows for easy monitoring and management of RabbitMQ, providing insights into message throughput, queue lengths, and other operational metrics.

The management dashboard has proven to be quite useful for debugging and monitoring the state of the application and how data is being consumed.

#### 3.2.4.2 Naming Conventions

A clear and systematic naming scheme is key for the organization and management of exchanges and queues within RabbitMQ. For this system, the exchanges are named following the pattern of `domain.event`, which in the case of receiving new snapshots is `snapshot.received`. This naming helps in identifying the purpose of the exchange and the type of events it handles.

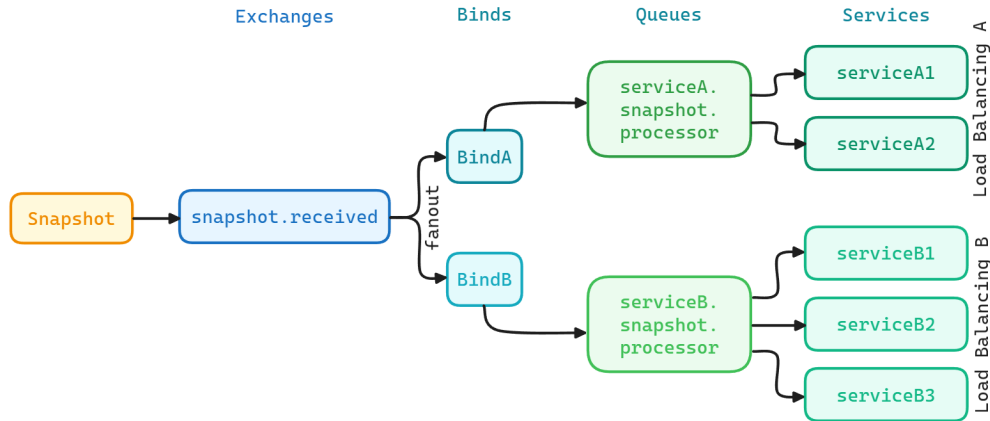
Similarly, queues follow a structured naming convention: `service\_name.domain.name`, such as `babybox-service.snapshot.processor`. This convention ensures that the queues are easily identifiable, related to specific services, and categorized by the data they consume.

#### 3.2.4.3 Communicating through the Message Broker

The configuration of the message broker is decentralized; each service autonomously sets up its part of the messaging architecture.

Producers are responsible for initializing the exchanges and configuring them to use the `fanout` strategy (publish/subscribe strategy), which is ideal for broadcasting messages to multiple consumers. On the other side, consumers set up their queues and create bindings to connect these queues to the

relevant exchanges. This decentralized approach allows each service to independently manage its subscriptions and messages, thus promoting flexibility and scalability.



■ **Figure 3.1** Visualization of distributing snapshots among consumers.

For example, as shown on the visualization, when a snapshot from a baby-box is received in the `snapshot-handler` service, it is published to the `snapshot.received` exchange. Services that process these snapshots have their own queues bound to this exchange, ensuring that they receive the messages pertinent to their operational logic. This setup not only keeps the architecture clean but also allows for easy scaling as more services or types of messages are added over time.

#### 3.2.4.4 Types of Data

Currently, the only data being sent through RabbitMQ are the snapshots. We expect this to change rather quickly as other services appear or get into a situation where they need to access other types of data.

We have expected this to happen sooner rather than later and as such the system's design accommodates this growth seamlessly, allowing for new types of data to be exchanged.

### 3.2.5 MongoDB

MongoDB was chosen as the general-purpose database for its schemaless nature, which offers significant flexibility in how data is structured and managed. This flexibility is vital for a system that needs to adapt and evolve without cumbersome migrations or downtime. As well as its developer experience and productivity it offers with its tooling and resources.

### 3.2.5.1 Docker Compose Configuration

The deployment of MongoDB within the system uses Docker Compose, configured via environment variables in the `.env` file for aspects like authentication. This setup ensures that MongoDB integrates smoothly with the rest of the services, maintaining secure and reliable database access.

### 3.2.5.2 Practical Utilization of Schemaless Design

Despite being schemaless, which theoretically allows any form of document to be stored in a collection, it is key to maintain a logical structure to the data stored. The schemaless nature is not used in our system to store random data formats but to allow certain documents to optionally include additional fields without disrupting the existing structure.

For instance, the `babybox` entity primarily includes fields like `slug`, `name`, and `created_at`. Detailed data such as `location`, `network_configuration`, and `contacts` are optional, which means not all documents in the collection need to have these fields. This approach avoids the need for database migrations if new fields are introduced; documents without these new fields operate perfectly within the system norms, as the application logic is designed to handle such absences gracefully.

This method of using MongoDB's schemaless feature ensures that the system can seamlessly introduce new attributes to entities like `babyboxes` as requirements evolve or new features are implemented. The system thus remains agile and responsive to changes, with minimal overhead for managing data schema changes.

Through creating proper types in our services for these entities we can ensure that we handle the cases where these fields might be missing. This is especially necessary on the front-end, to let the user know that this information is not available, or to hide components that cannot be rendered due to missing data.

### 3.2.5.3 Efficient Data Retrieval

A potential problem that we are aware of is entities growing too big while there also arises a use case of needing only part of the entity. This could lead to inefficient queries and data transfers.

Such problem could be solved using projection. Projection allows specifying which fields should be returned by a query, thereby optimizing performance by fetching only the necessary data.

In the connection with REST, the filtering can be chosen by the client using a query parameter. If the need for such feature was growing, we could introduce a GraphQL for such service, which is great for such use cases as the client can specify exactly what kinds of fields they want. This would not be

a problem as we are using API versioning and the GraphQL API could listen on endpoints `/v2/` while the REST API is on `/v1/`.

#### 3.2.5.4 Indexing Considerations

As of now, the system does not employ extensive indexing due to the low volume of data—less than 100 babyboxes, a modest amount of users and limited amount notification template data. Introducing indexes could actually slow the database down as they slow down insert, update and delete queries due to the indexes having to be maintained and updated when the data changes.

However, as the notifications are going to be generated, indexing considerations are becoming relevant for the `notifications` collection specifically, particularly for improving access speeds to frequently queried fields like the babybox `slug`. Indexing this field could reduce query times for specific operations—mainly different queries revolving around filtering notifications based on a specific babybox.

#### 3.2.5.5 Tooling and Development Support

MongoDB’s ecosystem offers powerful tools such as Studio 3T and MongoDB Compass, which enhance database management capabilities.

Studio 3T facilitates direct interaction with the database, making data manipulation and query building intuitive and efficient. We used Studio 3T to quickly inspect the data in production.

We also used MongoDB Compass extensively as it is a great tool for understanding the data in the database, viewing statistics, debugging performance and creating queries and aggregation pipelines.

It support generating queries and even aggregation pipelines using a natural language on a real database with real data. It then visualizes the stages of aggregation pipelines and how the data looks like in after each stage. This visual representation is invaluable for debugging and optimizing complex queries, particularly when dealing with large datasets and sophisticated data transformations. The generated query or aggregation pipeline can be directly transformed to code in a specified programming language further improving developer productivity.

We used it also for debugging performance and understanding the benefits of introducing indexes.

### 3.2.6 InfluxDB

InfluxDB is employed specifically for its prowess in handling time-series data, which is crucial for storing and analyzing snapshots and event data generated by babyboxes. The decision to use InfluxDB was solidified by a thorough benchmark analysis that not only demonstrated its great performance with



synthetic but realistic data sets but also highlighted its commendable developer experience.

### 3.2.6.1 Docker Compose Configuration

InfluxDB is run within the system's infrastructure using Docker Compose, ensuring ease of deployment and consistency across development and production environments. Similar to other services, InfluxDB's configuration, including credentials and connection settings, is managed through environment variables specified in the `.env` file. This approach secures and simplifies access to the database, maintaining clean and manageable codebases.

### 3.2.6.2 Data Storage

The database is designed to store two primary types of data:

- **Snapshots:** These are recorded with their `slug` (identifier of a babybox) and `version` (babybox firmware version to know how to parse/treat the data) as tags, which facilitate fast and indexed queries alongside the `timestamp`. The temperature and voltage measurements are stored as fields.
- **Events:** For events, the `slug` and `unit` (which unit sent the event) are stored as tags, with the `event_code` stored as a string field. This structuring ensures that events are indexed and can be efficiently queried and analyzed over time.

The use of tags for indexing, along with timestamps, enables InfluxDB to perform well for queries that are typical in monitoring systems, such as retrieving time-based data sequences or aggregating data across time intervals.

Currently, the system does not store notifications in InfluxDB, despite them being timestamped and technically fitting the time-series data model. The decision to keep notification data in MongoDB instead stems from their tight integration with notification templates. Storing notifications in MongoDB simplifies making aggregated queries that combine notifications with their corresponding templates, a task that would be more complex if the data were spread across different databases.

### 3.2.6.3 Development Tools and Visualization

InfluxDB's developer portal offers visualization tools that significantly enhance the development and monitoring process. We utilized both Flux, InfluxDB's functional data scripting language, and the UI to set up queries and visualize the data. This feature was particularly valuable during the development phase, as it allowed for creating and debugging queries and immediate visual feedback on how data are stored and can be queried, ensuring accuracy and effectiveness in data handling.

### 3.3 Microservices

This project includes the development of several microservices, each focused on a specific domain. The Snapshot Handler Microservice handles the snapshots and events coming from babyboxes, while the Babybox Microservice manages the metadata for each babybox. The User Microservice provides authentication and user management, and the Notification Microservice manages notifications and their distribution. Finally, the Battery Analyzer Microservice captures and analyzes measurements of the babybox battery. Together, these microservices form a cohesive, maintainable, and efficient system.

#### 3.3.1 Snapshot Handler Microservice

The Snapshot Handler microservice is a cornerstone of the Babybox monitoring system, designed to manage the influx of data from various babyboxes efficiently. This microservice, developed using Go and the Echo framework, is crucial for handling both real-time data intake and user requests for data retrieval.

##### 3.3.1.1 New Firmware API Improvements

Due to constraints imposed by the babyboxes' hardware and firmware, the API endpoints could not fully adhere to ideal RESTful practices. The system had to accommodate legacy firmware that sends data through unconventional endpoints such as `/BB.{slug}.data`. This endpoint receives data through query parameters like `BB={slug}&T0={value}&T1={value}...&T8={value}`, complicating the standardization of data intake.

As part of this thesis, a new firmware version was developed to establish a more standardized API structure. For newer versions of the firmware, the following improvements were made:

- Snapshots are received via `/send/snapshots/thermal/{slug}` using the same query parameters for consistency with older versions.
- Events data is captured through `/send/events/{slug}`, with each event encoded as a string that indicates the unit from which the event was sent (thermal or engine) followed by a code specifying the event.

To maintain compatibility with legacy firmware and handle new features in the updated firmware, snapshots are labeled with either version 1 or version 2 based on the endpoint they came from. This version-based labeling enables the system to differentiate between older and newer firmware snapshots, allowing for specific processing operations tailored to each version.

The new firmware also increased the frequency of snapshot data submissions, reducing the interval from every 10 minutes to every 5 minutes. This

adjustment ensures more timely data collection and enhances the accuracy of real-time monitoring.

This way, despite the introduction of new API endpoints and changes in snapshot frequency, the system remains compatible with older firmware versions and ready to be expanded upon.

### 3.3.1.2 Slug Conversion and Usage

Babyboxes self-identify using names based on their geographical location—such as BRNO, PRAHA2, PRAHA6. These names are transformed into ‘slugs’ that act as unique identifiers within our system. The conversion process involves standardizing the names into a slug format by converting all letters to lowercase, replacing spaces with hyphens, maintaining numbers, and removing other characters. These slugs are crucial for tracking and associating data accurately across the system’s databases and services.

### 3.3.1.3 Data Handling and Storage

Upon receipt, snapshot data is immediately formatted into InfluxDB points, with measurements stored as fields and the babybox’s slug and firmware version stored as tags. This structuring facilitates efficient time-series data analysis and storage as InfluxDB automatically indexes the data. Simultaneously, these snapshots are published to RabbitMQ in JSON format, allowing other microservices within the architecture to react to and process the incoming data appropriately.

### 3.3.1.4 API Endpoints for Data Retrieval

The Snapshot Handler microservice provides comprehensive endpoints to access the data:

- `/snapshots`: This endpoint is generally used for debugging purposes and returns all snapshots stored in the system.
- `/snapshots/{slug}`: Targets data retrieval for a specific babybox. It supports query parameters like `from`, `to`, `n`, and `fill`, which allow users to filter the snapshots by date, limit the number of results, and specify how missing data points should be handled.
- `/snapshots/{slug}/summary`: Provides a summary of data for a given babybox over a specified period. This endpoint returns what looks like a single snapshot, but instead of values for temperatures and voltages it returns aggregated data - minimum, maximum and average values. This query can be adjusted using the `from` and `to` query parameters.

For event data, the service offers:

- `/events`: Retrieves all event records.
- `/events/{slug}`: Fetches events for a specific babybox, with similar filtering capabilities as the snapshot endpoints. Again with the ability to specify `from`, `to` and `n` query parameters.

The common query parameters that can be used to adjust the result set that is going to be returned are:

- `from`: Specifies the start date for the data retrieval in YYYY-MM-DD format and it is set to the beginning of the day. It defaults to one year ago if not provided. This parameter is pivotal for bounding the query to a specific timeframe.
- `to`: Sets the end date for data retrieval, also in YYYY-MM-DD format and it is set to the end of the day, and defaults to the current date. This parameter allows users to define the period for which they need data, up to the very day of the query.
- `n`: Limits the number of data points returned. This parameter is useful for managing the volume of data retrieved, particularly in scenarios where only a sample or the most recent entries are needed.

As an example, a request made to the snapshots endpoint: `/snapshots/brno?from=2024-01-01&to=2024-01-01`, the service would respond with all data recorded from the babybox with slug `brno` on the day 1.1.2024 (the whole day would be selected as `from` is set to 1.1.2024 00:00:00 and `to` is set to 1.1.2024 23:59:59)

### 3.3.1.5 API Versioning and Envelope Pattern

To ensure backward compatibility and ease future updates, the API incorporates versioning. Each user endpoint begins with a version prefix, such as `/v1/`, ensuring that any future changes do not disrupt existing integrations.

The response from these endpoints adheres to the envelope pattern, which includes a data object for the payload and a metadata section containing error status and messages. A response follows this pattern: `{ data: ..., metadata : { err: boolean, message: string } }`.

### 3.3.1.6 Authentication

Each endpoint within the `/v1/` group of endpoints (user endpoints) checks for the JWT token that should be attached in each request in the `Authorization` header. Only after checking its legitimacy and expiration date, is the request handled. Otherwise it appropriately sends a `401 status - Unauthorized`.

### 3.3.1.7 Health Check Endpoint

The microservice also includes a health check endpoint. This endpoint provides very simple information about its status and version.

## 3.3.2 Querying Data from InfluxDB

For querying, we utilize Go's `fmt.Sprintf` function to dynamically construct queries for InfluxDB. This method allows us to embed user-specified parameters directly into our Flux queries, adapting to varied data retrieval.

The `fmt.Sprintf` function in Go formats strings with placeholders, which are replaced by subsequent arguments. This functionality is beneficial for constructing database queries, where parameters such as bucket names, time ranges, and limits might change based on user input.

Here is an example of how we construct a query for retrieving data:

```
1 fluxQuery := fmt.Sprintf(`from(bucket: "%s")
2   |> range(start: %s, stop: %s)
3   |> filter(fn: (r) => r._measurement == "%s" and r.slug == "%s")
4   |> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn:
  ↪ "_value")
5   |> sort(columns: ["_time"], desc: true)
6   |> limit(n: %d)`, service.bucket, from.Format(time.RFC3339),
  ↪ to.Format(time.RFC3339), measurementNameThermal, slug, n)
```

■ **Code listing 3.5** Query template for a snapshot range query based on slug.

- **from:** Specifies the InfluxDB bucket from which to retrieve data. The bucket name is dynamically inserted based on the service configuration.
- **range:** Sets the time range for the query. The start and stop parameters are formatted as RFC3339 strings, ensuring accurate parsing by InfluxDB (e.g. 2024-01-20T12:34:56Z).<sup>[59]</sup>
- **filter:** Filters the data to include only entries that match a specific measurement (measurements from the thermal unit) and a particular `slug`. This ensures that the query returns only the relevant data associated with a designated babybox.
- **pivot:** Reorganizes the data format by essentially merging all the rows based on its time into one row with multiple columns; otherwise we would get multiple rows with each temperature and voltage separate.
- **sort:** Orders the results in descending order by time, ensuring that the most recent records are presented first.
- **limit:** Restricts the number of entries returned by the query.

For generating aggregated data reports - minimum, maximum, and average values over a specified period, we use a template that inserts aggregation functions dynamically:

```

1  queryTemplate := `from(bucket: "%s")
2  |> range(start: %s, stop: %s)
3  |> filter(fn: (r) => r._measurement == "%s" and r.slug == "%s" and
↪ r._field == "%s")
4  |> %s()
5  |> yield(name: "%s_%s")`
6
7  queries := ""
8  fields := []string{
9      "temperature_inside",
10     "temperature_outside",
11     "temperature_casing",
12     "temperature_top",
13     "temperature_bottom",
14     "voltage_in",
15     "voltage_battery",
16 }
17 aggregations := []string{"min", "mean", "max"}
18
19 for _, field := range fields {
20     for _, aggregation := range aggregations {
21         query := fmt.Sprintf(queryTemplate, ...)
22         queries += query
23     }
24 }

```

■ **Code listing 3.6** Query creation for an aggregated query.

The differences between the previous query are:

- **filter:** The filter function now further narrows the data to only a specific variable within the measurement (e.g. `temperature_inside`).
- `|> %s()` is replaced with the desired aggregation function (`mean()`, `min()`, or `max()`), tailoring the query to compute specific statistical metrics.
- The yield function labels the output of the query, which is important because the query will yield multiple outputs (for each field and for each aggregation function).

The resulting query is constructed by iterating over all fields and all aggregation functions and concatenating all the queries together.

### 3.3.2.1 Gap Filling Strategies

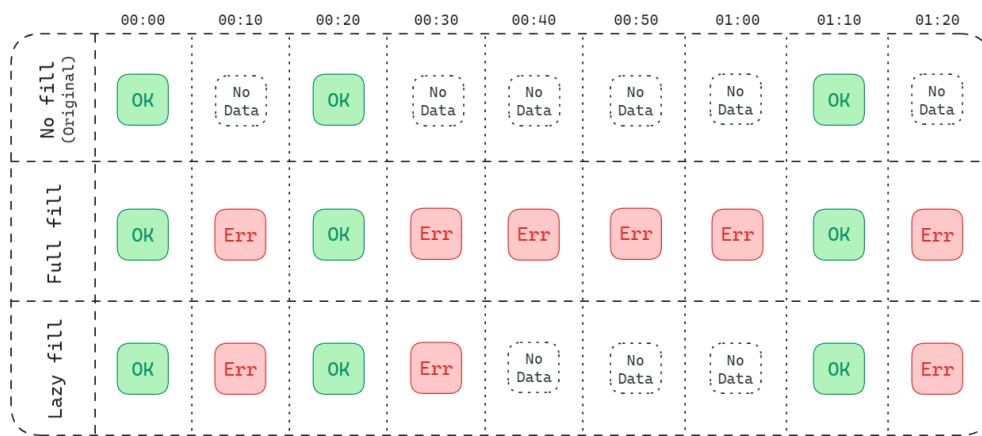
As babyboxes send their snapshots, there might be some outages on either end, network problems, or other reasons why the snapshots do not arrive to be stored. Thus, we should make it clear to the user that these data points are missing.

In the new application, we made a transition from a static to a dynamic gap filling strategy in the Snapshot Handler microservice, which marks a significant improvement in how data integrity is maintained while managing storage efficiency. Previously, a cron job was responsible for identifying and filling data gaps by creating and storing synthetic data points directly in the database. This method, though effective in maintaining complete datasets, led to unnecessary storage of filler data that had no analytical value.

The new dynamic approach addresses data gaps in real-time during data retrieval, eliminating the need to store synthetic data persistently. API clients specify their preferred gap filling strategy via the `fill` query parameter when making API requests to the endpoint `/snapshots/{slug}`. This allows for on-demand data manipulation, aligning data presentation with immediate user needs without impacting the underlying data storage.

- No Filling (query parameter missing, "no" or "false"):
  - If this option is selected, the API directly returns the data as stored, including any gaps resulting from non-transmission periods or system downtime. This method is straightforward; no loop or additional processing is involved.
- Standard Filling (query parameter "fill"):
  - This algorithm initiates when gaps of over 12 minutes (giving a bit of a buffer over the usual 10 minutes for some inconsistencies) are detected between sequential data points. It operates by looping through the dataset from the `from` date to the `to` date specified in the query.
  - For each gap detected, the algorithm inserts synthetic data points every 10 minutes until it reaches the timestamp of the next available real data point or fills up to the `to` date.
  - In practice, if a gap is found, a synthetic data point is created at 10 minutes after the last real data point, and the process repeats from this new synthetic point, continually checking for and filling any subsequent gaps.
- Lazy Filling (query parameter "lazy"):
  - Similar to the standard filling, this method begins by identifying gaps larger than 10 minutes.

- Instead of filling every 10-minute interval within a gap, lazy filling places only one synthetic data point at the start of the gap. It then jumps to the next real data point, thereby placing significantly fewer synthetic points.
- This strategy leads to every gap of more than 12 minutes being filled with exactly one synthetic data point. This process reduces the computational load compared to the full filling algorithm while still providing some indication of a gap in the data.



■ **Figure 3.2** Visualization of the filling algorithms.

An accompanying figure visually differentiates the results of each strategy. It illustrates a timeline where:

- No filling leaves clear gaps.
- Standard filling densely populates gaps with synthetic points at regular intervals.
- Lazy filling sparsely populates gaps, marking only the beginning of each detected gap with a single point.

In our efforts to provide this functionality, we have also explored several methods to dynamically manage gaps directly within the database using the InfluxDB's `aggregateWindow` function which looked promising in the beginning:

```
1 |> aggregateWindow(every: 10m, fn: last, createEmpty: true)
```

■ **Code listing 3.7** Using `aggregateWindow` for filling gaps in time-series data.



This Flux query command divides the dataset into 10-minute windows and applies the last function to each window - each window's result is the last value within that window. If a window lacks data points, the function is designed to create an empty data point.

However, several issues arose with this approach, making it less viable for our specific needs:

- **Timestamp Inaccuracy:** InfluxDB assigns the timestamp of the window's end to any created empty data points, rather than preserving the actual time of the missed data. This behavior misrepresents the timing of data, which is somewhat important for accurate monitoring and analysis in scenarios where precise time stamps of events are needed for diagnostic or operational purposes.
- **Data Point Exclusion:** The last function within each window might inadvertently exclude relevant data points if more than one entry is present in the same window. For example, if two data points are logged at 12:00:00 and 12:09:59, only the second is retained and the first is ignored. This can lead to loss of potentially valuable data. Note that other functions such as `mean` (calculating the average of each window) would not solve the issues as the following window would still be missing a value.
- **Misleading Last Data Point:** The function often assigns the current time to the last data point in the sequence, due to the window ending at the `to` time (which is set to the current time in most cases). This can lead to confusion and inaccuracies in data interpretation.

Given these limitations, while the `aggregateWindow` function offered a streamlined, database-integrated solution to data gap handling, the trade-offs in terms of data accuracy and integrity were significant. That is why we ultimately decided to develop our own filling algorithms.

### 3.3.3 Babybox Microservice

The Babybox Service microservice plays a key role in the monitoring system by managing detailed information about each babybox. This service is vital for maintaining an accurate registry of babyboxes and incorporating data from newly detected units efficiently.

#### 3.3.3.1 Core Functionality

Central to the service's function is the creation and updating of babybox entries in the MongoDB database. This process occurs automatically when new babyboxes are detected through the snapshots they send. Detection is facilitated by RabbitMQ on the `snapshots.received` exchange, where each snapshot is checked against existing database entries:

- If the babybox's unique slug is not found in the database, it indicates the arrival of a snapshot from a undiscovered babybox. The service responds by creating a new entry for this babybox, setting the `name` initially to match the `slug` and recording the `created_at` timestamp to the current time.
- If the slug already exists, then it gets ignored as far as this service is concerned.

Snapshots are read from RabbitMQ by having the babybox service create its own queue for these snapshots and attaching it to the `snapshot.received` exchange, which is managed by the snapshot handler service. This exchange is set to use the `fanout` strategy leading to the babybox service reading all the new snapshots and making sure there is no snapshots coming from untracked babyboxes.

### 3.3.3.2 API Design and Endpoint Functionality

The Babybox Service mirrors the structure of the Snapshot Handler microservice, employing several best practices for secure and efficient data management:

- API Versioning is implemented to manage changes effectively without disrupting service for existing clients.
- JWT Authentication secures access, requiring valid tokens for operations that access or modify babybox data, thereby enhancing the system's security.
- Envelope Pattern is used in API responses to include metadata alongside the primary data, facilitating better client-side processing and error handling.

The service provides a range of endpoints, each serving specific functions:

- `POST /babyboxes`: Primarily used for debugging, this endpoint allows manual creation of babybox entries.
- `GET /babyboxes`: Offers a list of all registered babyboxes, showing basic identifiers: `slug` and `name`, useful for listing all the babybox on a page.
- `GET /babyboxes/{slug}`: Delivers detailed information for a specific babybox.
- `PUT /babyboxes/{slug}`: Enables comprehensive updates to a babybox's details by requiring the full updated object to be submitted.

### 3.3.4 Notification Microservice

The Notification Microservice is an essential component of the Babybox monitoring system, designed to handle all aspects related to notifications. This includes managing notification templates, storing notification data, and facilitating the sending of notifications to users via email. This service plays a pivotal role in ensuring that relevant parties are promptly and reliably informed about significant events or statuses related to babyboxes.

#### 3.3.4.1 REST API

It offers a comprehensive REST API that supports various operations related to notifications and templates:

- **GET /notifications:** Retrieves all notifications, primarily used for debugging and administrative overview.
- **GET /notifications/{slug}:** Fetches notifications for a specific babybox identified by its slug. This endpoint supports additional filtering by start and end date parameters formatted in YYYY-MM-DD, allowing users to narrow down the search to a specific time frame.
- **DELETE /notifications/{id}:** Allows deletion of a specific notification by its ID, useful for managing erroneous or outdated notifications.
- **POST /notifications/{template\_id}/{slug}:** Primarily for testing and debugging, this endpoint facilitates the manual creation of a notification based on a specified template ID and babybox slug.
- **GET /templates:** Lists all notification templates, providing an overview of the different notification criteria set up within the system.
- **GET /templates/{slug}:** Retrieves notification templates applicable to a specific babybox or globally. It includes an optional global boolean query parameter that, when set to true, returns both the templates specifically for the given slug and those marked as global.
- **GET /templates/id/{id}:** Provides details of a specific template by its ID, useful for detailed template management and editing.
- **POST /templates:** Supports the creation of new notification templates, allowing users to define new criteria and notification behaviors.
- **DELETE /templates/{id}:** Enables the deletion of a notification template by its ID, which is essential for removing outdated or unnecessary notification rules.
- **PUT /templates/{id}:** Updates an existing notification template by replacing the entire template object. This is crucial for adapting notification behaviors as the monitoring requirements evolve.

### 3.3.4.2 Template Configuration and Snapshot Processing

Users can create notification templates that detail the circumstances under which notifications should be sent. These templates include a scope (specific babybox `slug` or "global"), condition made out of 3 parameters: `variable` (which variable are we monitoring; e.g. "temperature.inside"), `comparison` (binary predicate that is used in the condition to compare the variable with a value; e.g. "<"), `value` (a numeric value to compare against), and metadata like the notification's `title`, `message`, `severity`, and recipients email addresses.

When a new snapshot arrives through the `snapshot.received` exchange from RabbitMQ, the service first identifies the applicable templates. It filters these templates to include only those whose `scope` matches the babybox's `slug` from the snapshot or those set as "global". For each relevant template, the service checks if the snapshot meets the predefined conditions.

### 3.3.4.3 Strategies to Refining Notification Frequency

Given the high potential for numerous snapshots generating an excessive volume of notifications, we have developed several strategies to manage and refine when notifications are triggered:

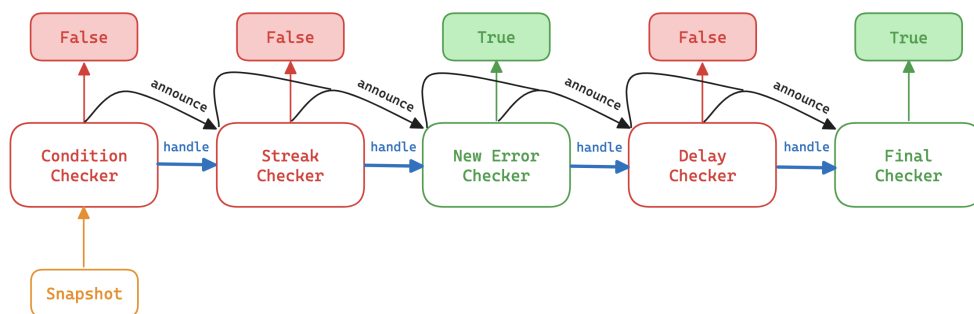
- **Streak Requirement:** This feature requires a certain number of consecutive snapshots to meet the condition before a notification is generated. This strategy is particularly useful for avoiding false alarms due to data fluctuations that momentarily cross threshold values.
- **New Error Detection:** This strategy prioritizes the detection of new errors. If a snapshot meets the condition and represents a change from previous snapshots (i.e., the previous did not meet the condition but the current one does), a notification is immediately triggered. This ensures prompt alerting for new issues, independent of their the delay limitation checker.
- **Delay checker:** After a notification is generated, a delay period is enforced during which no further notifications for the same condition are issued, regardless of the incoming snapshot data. This delay prevents a flood of messages for persistent or recurring issues, helping to manage the volume of outgoing communications effectively.

### 3.3.4.4 Chain of Checkers

Our approach in checking for different conditions and limitations for generating notifications was heavily inspired by the chain of responsibility pattern, where each checker in the chain has a specific role:

- **Condition Checker:** This initial checker evaluates whether the snapshot meets the basic conditions specified in the template.

- **Streak Checker:** If the condition is met, the streak checker assesses whether the condition has been met consecutively the required number of times. It does this using an internal state. It has a dictionary `key`  $\rightarrow$  `streak`. The key is created by using the `id` of the template and the `slug` of the snapshot - creating a unique identifier for this combination; the streak is then tracked based on each snapshot that comes in.
- **New Error Checker:** This checker intervenes to ensure that any new error triggers an immediate notification, even if the following checker would decide otherwise. It does so by having an internal dictionary, similar to the streak checker, `key`  $\rightarrow$  `last_decision`.
- **Delay Checker:** This checker applies a cooldown period, during which no further notifications are generated from the same template for the same babybox, regardless of the snapshot data. It does so by checking the timestamp of the last notification generated for this template/babybox combination.
- **Final Checker:** This is a filler checker at the end of the chain. It is there so that the code of the previous checkers is a bit less complicated as they do not have to check if there is another checker after them in the chain. If a snapshot gets to this checker, then it always creates the notification.

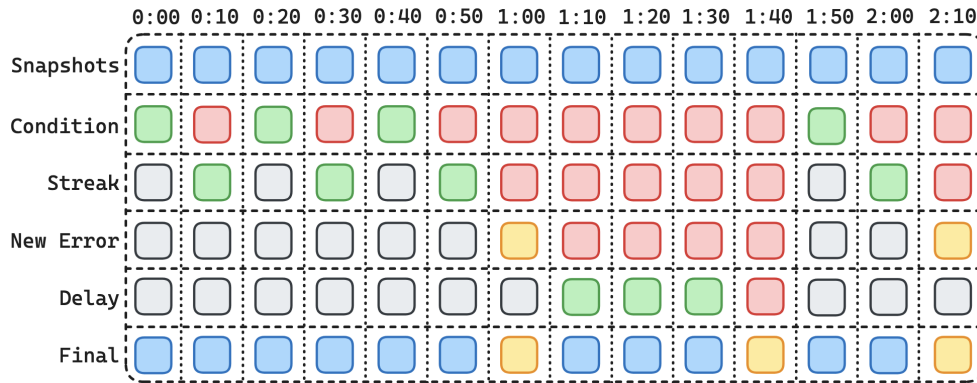


■ **Figure 3.3** Visualization of the notification checking chain.

In this figure, we can better picture how this chain is constructed. We can see that the condition, streak and delay checker have the ability to make an early return with the value `false` (notification should not be created) and the new error and final checkers have the ability to make an early return with the value `true` (notification should be created).

Each checker has a next checker and passes the snapshot to it using the `handle` method. If an early decision is made, then it uses the `announce` method instead, which informs all the following checkers of the decision that has been made. This is done for checkers to update their state based on the decision that

has been made (e.g., streak checker resetting the streak of this template/baby-box combination to 0 if the condition was not met).



■ **Figure 3.4** Visualization of the notification checking in action (streak set to 2, new error set to `true`, delay set to 35 minutes).

In this figure, we can see how these checkers work together on a notification template with streak set to 2, new error turned on and delay set to 35 minutes. The color-coded square represents data coming through the checkers:

- Blue for regular untagged snapshots.
- Green for snapshots that do not meet the triggering condition and are thus ignored.
- Red for snapshots that meet the condition and may lead to a notification depending on further checks.
- Yellow for snapshots that generated a notification.
- Gray for announcement calls.

If a snapshot is coded red by the condition checker, it proceeds to the streak checker. In our case a single alerting data point will not trigger a notification unless followed by another (streak has to be at least 2), that is why no notification is generated during the first 5 snapshots.

In the new error checker, a notification can already be generated if it gets recognized as a new error. In the last snapshot, we can see that the notification is generated even though the delay between the notifications is only 30 minutes (from 1:40 to 2:10).

The delay checker plays its role in preventing too frequent notifications to be generated, which we can see for snapshots at 1:10 to 1:30, after that another notification is generated because the time from the previous one (which happened at 1:00) is bigger than 35 minutes.

If a decision is made, then the internal state of the checkers is updated through announcement calls, which is symbolized in gray squares.

We can see that most checkers return an early `false` (meaning do not generate a notification), the new error checker and in some sense the final checker are the only two checkers that can return an early `true`.

#### 3.3.4.5 Email Notification Mechanism

The primary function way of notifying the users is to send timely alerts in the form of emails to users based on specific conditions defined within user-created notification templates. These notifications are crucial for maintaining the operational integrity of the babyboxes by alerting maintenance staff to potential issues as they arise.

The service employs Python's `smtplib` with SSL to handle email notifications. This choice allows for secure transmission of email data over the network. When a notification condition is met, the service assembles an email that includes all intended recipients, sending a single email regardless of the number of recipients. This method focuses on efficiency knowing that email recipients are within the same organization, thus eliminating concerns over privacy breaches that might occur if recipients were external parties.

The content of the email is formatted in HTML, enhancing readability and allowing for more detailed messages. The email's subject is taken from the notification template's title, and the body includes the template message along with the slug or slugs associated with the babyboxes triggering the notification. This comprehensive format ensures that recipients receive clear and actionable information.

To mitigate the issue of notification fatigue—where users receive too many emails in a short period—additional strategies have been implemented. Instead of sending emails instantaneously with each triggered notification, the service aggregates notifications intended for the same template. These are temporarily stored in a dictionary mapping each template ID to an array of relevant babybox slugs.

A scheduled task (cron job) runs at the second minute of each hour and subsequently every five minutes (e.g., 02, 07, 12, 17 minutes past the hour). During each run, the task checks this dictionary and compiles all pending notifications into a single email per template. This consolidated approach means that users receive fewer emails, each potentially covering multiple notification events, thus significantly reducing the volume of incoming emails while still ensuring all critical information is communicated effectively.

We were trying to find a good balance between waiting too long before sending the emails, or not waiting at all on the other end. The former approach leads to less emails but prolongs the time between the event happening and the email being sent. The latter leads to immediate email notification, but way more emails are generated. The 5 minute window for collecting the

notifications into one was chosen based when the snapshots are sent - every 5 or 10 minutes, therefore we expect to send maximum of one email per template for each time we get new data.

### 3.3.5 User Microservice

The User Service microservice within the Babybox monitoring system serves two critical functions: user management and authentication. This service is essential to ensure that only authorized users can access the system and manage the babyboxes effectively.

#### 3.3.5.1 REST API

The service provides a suite of endpoints focused on user management and secure authentication:

- **POST /login:** This endpoint handles user login requests. It checks submitted usernames and passwords against the database records. If the credentials are valid, it generates and returns a JWT that the user can use for subsequent authenticated requests.
- **GET /users:** Retrieves a list of all users from the database, ensuring that sensitive password data (actually, the password hash) is never included in the response.
- **GET /users/{username}:** Fetches detailed information about a specific user by username.
- **DELETE /users/{username}:** Allows for the deletion of a user by their username.
- **POST /users:** Facilitates the creation of a new user by submitting a username, password, and email.

#### 3.3.5.2 Registration Process

Users are registered by providing a **username**, **password**, and **email**. Since this is an internal tool intended for a limited number of users, the registration process is straightforward without the need for email confirmations. The system uses Mongoose to interact with MongoDB, utilizing schemas to define data structures and hooks to perform operations before saving data, such as password hashing.

The following Mongoose pre-save hook demonstrates how passwords are handled securely before storing user data:



```
1 userSchema.pre("save", async function (next) {
2   this.password = await Bun.password.hash(this.password);
3   next();
4 });
```

■ **Code listing 3.8** Using a pre-hook in Mongoose for hashing passwords before saving to MongoDB.

This hook is triggered before a user document is saved to the database. The password is hashed using Bun’s `Bun.password.hash()` function, which defaults to the *Argon2id* algorithm—a robust choice for password hashing due to its resistance to brute-force attacks. The hash function is configured to automatically handle salt generation and storage within the hash itself, making the storage and verification process straightforward and secure.

### 3.3.5.3 Login Process

The login process involves several key steps:

1. **Username Verification:** The system first retrieves the user document based on the submitted username. If no matching user is found, it indicates an error in the username entry.
2. **Password Verification:** The submitted password is verified against the stored hash using `Bun.password.verify()`, which checks the password against the hash stored in the database. This function is efficient as it reads the algorithm details and parameters directly from the hash, eliminating the need for reconfiguring each time.
3. **JWT Creation:** Upon successful password verification, a JWT is generated:

```
1 const token = signJWT(user);
2
3 export function signJWT(user: UserSanitized) {
4   const payload = { username: user.username, email: user.email };
5   const options = { expiresIn: "7d" };
6
7   const secret = process.env.JWT_SECRET;
8   const token = jwt.sign(payload, secret, options);
9
10  return token;
11 }
```

■ **Code listing 3.9** Generating a new JWT in Bun.

This function constructs a JWT with a payload containing the user’s username and email, setting an expiration of 7 days. The `jwt.sign()` method

is used to create the token, which the user can then use to authenticate subsequent requests across the system. The `jsonwebtoken` package uses the HS256 algorithm by default (HMAC using SHA256) to sign the JWT.

#### 3.3.5.4 Integration with Other Services

The generated JWT is critical for securing the APIs in other services, where similar JWT handling mechanisms are employed to validate user requests. For example, in the Echo framework, JWT middleware can be configured to protect routes, ensuring that only requests with valid JWTs can access certain endpoints (in this case the `/v1/` group of endpoints). This is quite simple to do as Echo is well equipped for working with JWTs:

```
1  type JWTCustomClaims struct {
2      Username string `json:"username"`
3      Email    string `json:"email"`
4      jwt.RegisteredClaims
5  }
6  config := echojwt.Config{
7      NewClaimsFunc: func(c echo.Context) jwt.Claims {
8          return new(middleware.JWTCustomClaims)
9      },
10     SigningKey: []byte(jwt_secret),
11 }
12 v1Group.Use(echojwt.WithConfig(config));
```

■ **Code listing 3.10** Protecting a group of endpoints in Go's Echo by checking the JWT validity.

Similarly in Python with FastAPI we can check the JWT validity before each request starts being processed by the endpoint handler using the `jwt` package:

```
1  async def get_current_user(  
2  authorization: HTTPAuthorizationCredentials = Security(security),  
3  ) -> dict:  
4      token = authorization.credentials  
5      try:  
6          payload = jwt.decode(token, get_jwt_secret_key(),  
7              ↪ algorithms=["HS256"], options={"verify_exp": True})  
8  
9          return payload  
10     except jwt.PyJWTError:  
11         raise HTTPException(  
12             status_code=401,  
13             detail="Unauthorized",  
14         )  
15     ...  
16 @router.get("/", response_model=...) async def endpoint_handler(user=Depends(get_current_user)):
```

■ **Code listing 3.11** Protecting endpoints in Python's FastAPI by checking the JWT validity.

Lastly in Bun with Elysia we can also define a group of endpoints and attach a function that will check the validity of the request before each processing of the request. This is again pretty straight forward using the `jsonwebtoken` library:

```

1  export function isAuthenticated(headers: Record<string, string |
   ↪  undefined>): {
2    isAuth: boolean;
3    payload: JWTPayload | null;
4  } {
5    const token = extractToken(headers["authorization"]);
6    if (token.isNone()) {
7      return { isAuth: false, payload: null };
8    }
9
10   try {
11     const res = jwt.verify(token.unwrap(), process.env.JWT_SECRET);
12     const payload = jwtPayloadSchema.parse(res);
13
14     if (payload.exp && Date.now() >= payload.exp * 1000) {
15       return { isAuth: false, payload };
16     }
17     return { isAuth: true, payload };
18   } catch (error) {
19     return { isAuth: false, payload: null };
20   }
21 }
22 ...
23 app.group("/users", (app) =>
24   app.onBeforeHandle(({ headers, error }) => {
25     const { isAuth } = isAuthenticated(headers);
26     if (isAuth === false) {
27       return error(401, "Unauthorized");
28     }
29   })
30   .use(userRoutes),
31 ),

```

■ **Code listing 3.12** Protecting a group of endpoints in Bun’s Elysia by checking the JWT validity.

### 3.3.6 Battery Analyzer Microservice

The Battery Analyzer microservice is a component of Babybox Dashboard, specifically designed to analyze and assess the health of the accumulator within babyboxes. This service is crucial for pre-emptive maintenance and ensuring reliability when babyboxes switch to battery operation.

#### 3.3.6.1 REST API

This microservice manages and processes voltage data from babyboxes, providing a REST API with functionalities similar to other services in the system, including a healthcheck endpoint, JWT authentication for security, and API versioning strategy. The key endpoints of this service are:

- **GET /measurements:** Retrieves all recorded measurements, mainly utilized for debugging and data verification purposes.
- **GET /measurements/{slug}:** Accesses measurement data for a specific babybox by its slug.
- **POST /measurements:** Supports the creation of new measurement entries, typically used during testing phases or when manually adding data.
- **DELETE /measurements/{measurement\_id}:** Allows for the deletion of specific measurement entries to maintain data integrity and relevance.

Each entry in MongoDB associated with these endpoints includes a MongoDB generated ID, the `slug` of the babybox from which the measurement was taken, and an array of tuples that record each measurement's datetime and corresponding voltage of the accumulator. Importantly, a `quality` attribute of the battery is not stored directly in the database but is dynamically calculated on request, based on the array of sub-measurements.

### 3.3.6.2 Taking a Measurement

When designing this service, we kept in mind that the vast majority of input voltage outages are going to occur during planned diesel generators tests in hospitals. These tests typically interrupt the main power supply for about 20 minutes, during which the babybox operates solely on its accumulator, providing a window for battery health evaluation.

When a power outage is detected—indicated by the input voltage dropping near 0V—the microservice initiates a recording of battery voltage levels. It captures an initial measurement while the power is still on, followed by subsequent measurements after the power cuts off. These measurements continue at regular intervals until either the power is restored or a maximum duration of three hours is reached. Each recorded entry includes a timestamp and the corresponding voltage, forming an array that is then stored in the database. This data collection process can later be used for visualization, alerting and estimating the state of health of the battery.

### 3.3.6.3 Battery Quality Assessment Idea

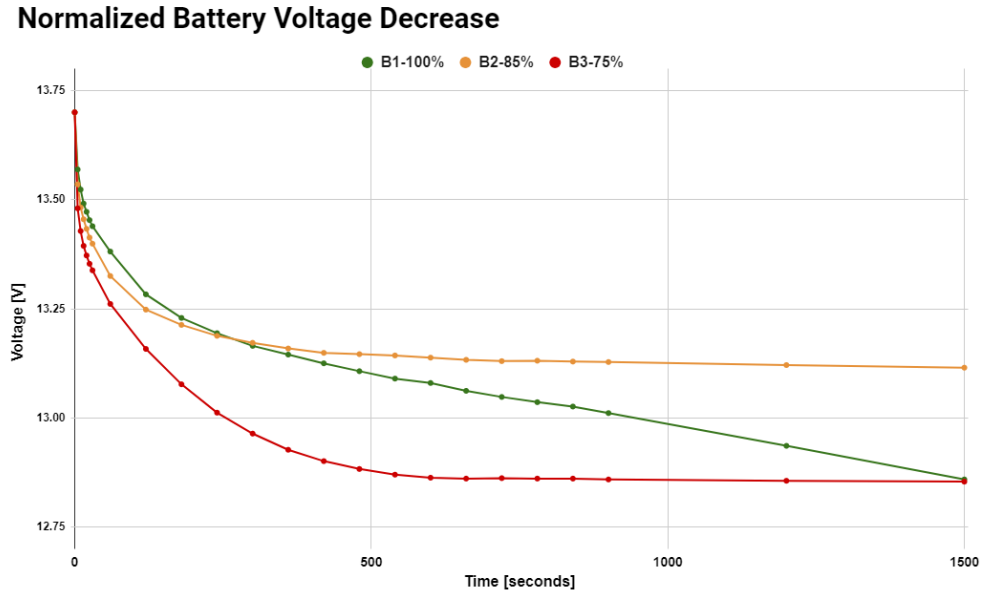
The initial idea for assessing battery quality was based on the response of the battery voltage over the duration of the power outage. The initial hypothesis was that by analyzing the decline in voltage from the moment the power outage, valuable insights could be made regarding battery's health. Our initial approach was looking at the first 12 minutes of the data, where the decline is the biggest. This decision was also supported by the context in which these measurements are going to be taken in reality; most of the measurements will offer only 10-20 minutes of data (2 snapshots with power outage with the old firmware), so this was also a practical decision to normalize the process.

#### 3.3.6.4 Empirical Research and Methodology

To validate this approach, empirical research was conducted using a testing babybox, where the conditions could be closely monitored. The methodology involved:

- **Equipment Setup:** High-precision instruments were used for the voltage measurements, specifically the UNI-T UT71A multimeter, which offers high precision for measuring voltage.
- **Test Conditions:** A testing babybox equipped with different accumulators that are regularly used in babyboxes around Czechia. These accumulators were charged under normal conditions and then subjected to a simulated power outage.
- **Battery Quality:** Each battery has been measured for its state of health using the DHC tester BTJ41 to then compare to the voltage measurements to see a correlation.
- **Measurement Intervals:** Voltage readings were taken just before the power outage and then at precise intervals—at 5, 10, 15, 20, 25, 30 seconds, followed by every minute up to 15 minutes and then at 20 minutes and 25 minutes, to capture the dynamic voltage response accurately.

The research aimed to correlate the voltage response with the known state of health of the batteries. However, the results indicated that the voltage measurements within the first 20 minutes (the typical duration of hospital generator tests) did not consistently correlate with the batteries' health as determined by traditional testing methods as seen on the figure below.



**Figure 3.5** Normalized battery voltage decrease over time (y-axis starts at 12.5V and ends at 13.8V; x-axis is from 0 seconds to 25 minutes).

This finding suggested that such method cannot be used for assessing the health of the accumulator. The suspected problem of this method of estimating the state of health of the accumulator is that the current drawn, which is about 0.6A during the outage as only the necessary parts are operational, from the accumulator is too low for us to calculate the internal resistance which determines the state of health. We would need to momentarily load the battery with higher current to obtain a proper measurement, which is not achievable with our setup. The results of our research show that our method is not suitable for estimating the state of health of the battery, as seen in the figure - the battery with the highest quality is actually the one with the fastest voltage decline, which is the opposite of what we would have expected.

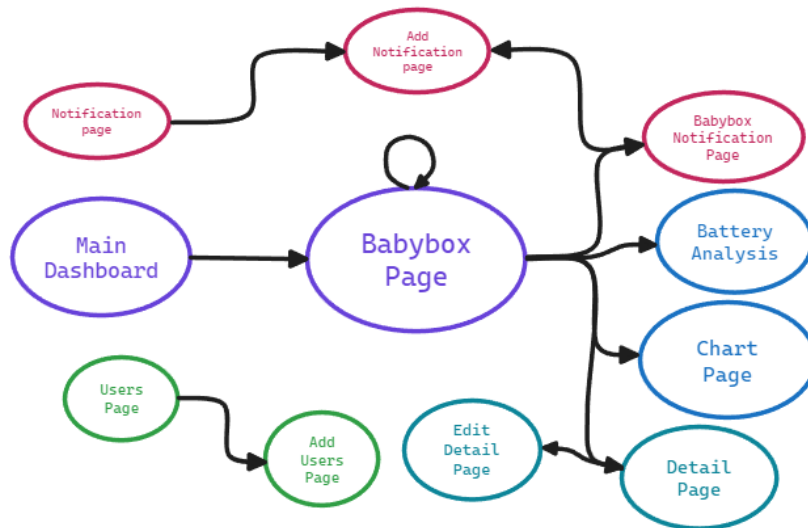
The insights gained from this study have led to a reevaluation of the approach to battery health assessment within the Battery Analyzer microservice. In the future, we would like to explore other methods. Currently, we leave the data accessible to the staff in the application so that they can view the measurements as they are taken over time.

### 3.4 Front-end

When designing the frontend of Babybox Dashboard, user experience was a primary focus. We aimed to create an intuitive and efficient interface that would make daily operations seamless for users, keeping in mind the insights gained from our user persona analysis. Our goal was to design an application

that not only addressed the user's needs effectively but also simplified their interactions with the system.

### 3.4.1 Application Flow



■ **Figure 3.6** Diagram showing the flow through the application.

The primary interface starts with a main table that lists all the babyboxes in the system, displaying their most recent status. This view allows users to quickly assess which babyboxes might require attention. Upon selecting a babybox, the user is taken to a detailed page that not only shows information about the recent status and statistics but also offers links to more specialized views:

- **Chart Page:** This section includes a comprehensive chart that displays historical data over selected period of time, allowing for detailed trend analysis and operational oversight.
- **Detailed Information:** Users can access information about each babybox - location, contact information and network configuration.
- **Analysis:** Includes tools and outputs, such as battery performance analyses.
- **Notifications:** Users can see notification templates for the specific babybox and also add new ones.

In addition to that, there is also a page showing all the notification templates with the option to add a new one and a similar page to manage users.

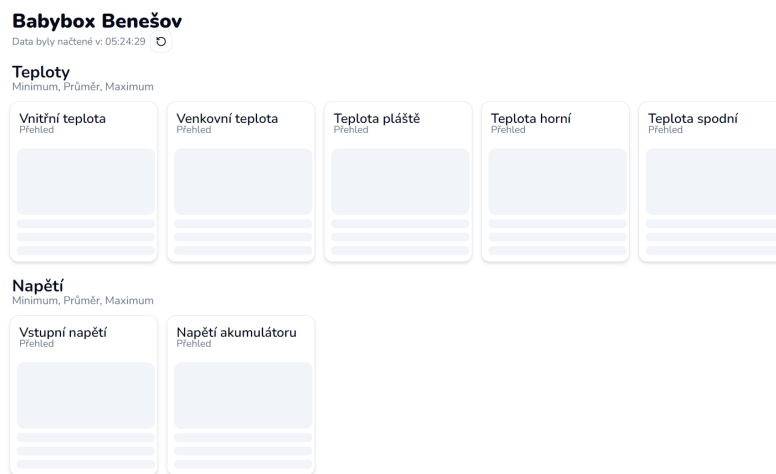


### 3.4.2 User Experience

One of the key features implemented to enhance user experience is the combobox in the navigation menu. This component lists all known babyboxes and includes a search functionality, allowing users to quickly find and navigate to a specific babybox page from anywhere within the application. This feature significantly reduces the time and effort required to manage multiple babyboxes, making the system more accessible and user-friendly.

Each babybox page also includes a side menu that offers quick navigation options, enabling users to seamlessly cycle through different babyboxes or jump back to the collective list. Such design considerations ensure that users can efficiently monitor and evaluate the status of any babybox with minimal navigation.

To enhance the interactive experience, the application employs skeleton components during data loading phases. These components mimic the layout of the content that is being loaded, providing users with a visual cue of what to expect, thereby improving the perceived responsiveness of the application. In addition, toast notifications are used to communicate the results of user actions, whether successful or unsuccessful, further helping to provide clear and immediate feedback that is crucial for effective user interaction.



■ **Figure 3.7** Skeleton components indicating that the overview widgets are loading/fetching data.

The application uses tooltips to assist users in understanding more complex aspects of the system without cluttering the interface with excessive text or navigating away from the current task. This feature is particularly beneficial to new staff or when introducing new features to help with understanding all the tools the application provides. Furthermore, tooltips are used to provide more context or additional information in certain scenarios.

Understanding that staff may need to access the application under different

lighting conditions, especially since they operate 24/7, the front-end includes an adaptive display setting. Users can toggle between dark and light modes, or they can choose the system mode, which automatically adjusts the display to match the user’s system settings. This flexibility ensures that the application is comfortable to use at any hour, reducing eye strain and enhancing visual comfort during nighttime or in low-light environments.

Security and seamless user experience are critical, especially in an application handling sensitive operational data. If a user’s session expires or if they attempt to access functionalities without proper authentication, the system proactively handles these events. If a user tries to navigate the dashboard without being authenticated, they are immediately redirected to the login page. This redirection is accompanied by a toast notification that informs them of the need to log in.

### 3.4.3 Implementing PWA

To achieve PWA functionality, we utilized the `next-pwa` plugin, a powerful yet easy-to-use library designed to seamlessly integrate PWA features into Next.js applications. This plugin simplifies the setup of service workers, which are needed for PWA to work.

One of the foundational requirements for a PWA is HTTPS, which secures the connection between the user and the application. We leveraged Caddy as our API gateway to automatically handle HTTPS, ensuring all data transmitted remains encrypted.

Another component needed for PWA is the manifest. It includes essential details such as the application’s `name`, `short_name`, and `description`, along with a set of icons in various sizes. All of this information is used when users “install” the application on their home screens. Furthermore, we specified that the application should launch in `standalone` mode with a `portrait` orientation, which provides a native-app-like experience by hiding the browser UI when opened.

To further refine the user experience specifically for the PWA, we made some CSS adjustments that are applied only when the application is running in standalone mode:

```
1 @media all and (display-mode: standalone) {...}
```

■ **Code listing 3.13** Adjusting CSS styling for PWA users only.

We made an extra effort of optimizing the design for mobile and table as we expect the usage of our application to increase on such devices. In fact, we adopted the mobile first design approach, where we designed the application for phone devices and then adjusted the design to accompany wider screens.<sup>[60]</sup>

We have done so using Tailwind directives such as `lg:` which can be prefixed to any class name to make that class name only apply on large screens. We have also made an extensive use of `flexbox`, which dynamically adjusts the positioning of elements on the screen.



**Figure 3.8** Screenshots of the PWA application in standalone mode looking like a native application.

We hope the PWA functionality to bring a better user experience when using the application on mobile or tablet devices providing a more native experience.

### 3.4.4 Login and Logout Functionality

The application includes dedicated pages for handling user authentication, specifically designed to facilitate secure access to the system. These pages, login and logout, are important for maintaining the security and integrity of user sessions and ensuring that only authorized personnel can access the dashboard functionalities.

The login page features a straightforward form that integrates with the

User Service to authenticate users and retrieve a JWT token. This token is critical as it serves as the user's credential for all subsequent requests within the session, ensuring secure communication between the client and the server.

Upon successful authentication, the JWT is stored in the browser's `localStorage` to persist the login state across page reloads. This token is also integrated into a React Context, specifically designed to manage authentication states throughout the application. The context structure is defined as follows:

```
1 interface AuthContextType {
2   token: string;
3   isAuthenticated: boolean;
4   isLoading: boolean;
5   login: (username: string, password: string) => Promise<boolean>;
6   logout: () => void;
7 }
8 ...
9 const { token, isAuthenticated, isLoading, login, logout } = useAuth();
```

■ **Code listing 3.14** Structure of the auth-context that can be used throughout the application.

Within the application, components can access the authentication state using the `useAuth` hook, which exposes the `token`, `isAuthenticated`, `isLoading`, `login`, and `logout` properties and methods:

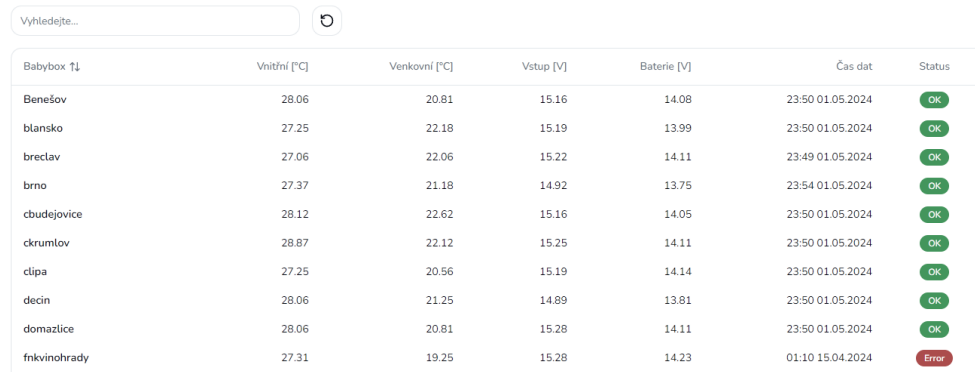
- **Check Authentication:** The `isAuthenticated` boolean indicates whether the user is currently authenticated.
- **Check Loading State:** The `isLoading` flag shows whether the authentication context has completed its initial loading and setup.
- **Perform Login and Logout:** Functions `login` and `logout` handle user login and logout operations, respectively.

The logout page clears the JWT from both the React Context and `localStorage` and confirms it to the user that the logout has been successful. After logging out, users are redirected away from the dashboard to prevent unauthorized access, usually back to the login page.

### 3.4.5 Starting Page

The starting page of the application features a table that provides a comprehensive overview of all the babyboxes in the system. The design of this page has largely remained the same, focusing on delivering latest data in a clear and accessible manner. To enhance the user experience and avoid overwhelming users with information, the status of each babybox is now displayed in a

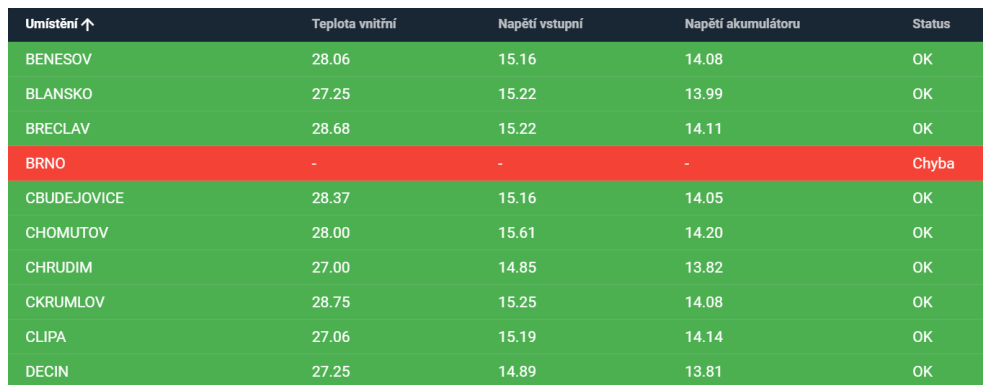
separate column within the table. This adjustment helps to make the table layout cleaner and more readable.



Babybox ↑	Vnitřní [°C]	Venkovní [°C]	Vstup [V]	Baterie [V]	Čas dat	Status
Benešov	28.06	20.81	15.16	14.08	23:50 01.05.2024	OK
blansko	27.25	22.18	15.19	13.99	23:50 01.05.2024	OK
breclav	27.06	22.06	15.22	14.11	23:49 01.05.2024	OK
brno	27.37	21.18	14.92	13.75	23:54 01.05.2024	OK
cbudejovice	28.12	22.62	15.16	14.05	23:50 01.05.2024	OK
ckrumlov	28.87	22.12	15.25	14.11	23:50 01.05.2024	OK
clipa	27.25	20.56	15.19	14.14	23:50 01.05.2024	OK
decin	28.06	21.25	14.89	13.81	23:50 01.05.2024	OK
domazlice	28.06	20.81	15.28	14.11	23:50 01.05.2024	OK
frkvinohrady	27.31	19.25	15.28	14.23	01:10 15.04.2024	Error

■ **Figure 3.9** New table containing all the babyboxes on the starting page.

The table quickly communicates the current status of each babybox, displaying key metrics such as the inner and outside temperatures, input and battery voltages, and the timestamp of the latest data received. The status column is particularly vital as it indicates whether the babybox is functioning normally or if there has been a disruption in data transmission (shown as an error if the last data received is older than 12 minutes). This status functionality is explained to the users via tooltips.



Umístění ↑	Teplota vnitřní	Napětí vstupní	Napětí akumulátoru	Status
BENESOV	28.06	15.16	14.08	OK
BLANSKO	27.25	15.22	13.99	OK
BRECLAV	28.68	15.22	14.11	OK
BRNO	-	-	-	Chyba
CBUDEJOVICE	28.37	15.16	14.05	OK
CHOMUTOV	28.00	15.61	14.20	OK
CHRUDIM	27.00	14.85	13.82	OK
CKRUMLOV	28.75	15.25	14.08	OK
CLIPA	27.06	15.19	14.14	OK
DECIN	27.25	14.89	13.81	OK

■ **Figure 3.10** Table showing a list of all the babyboxes in the previous version of the application.

For a comparison, the previous table colored the whole row based on the status of the babybox, this design was arguably more eye-catching, however, after replicating this design in the new solution we decided to change it as we thought it was too overwhelming and did not match the rest of the application.

### 3.4.5.1 Data Tables

We utilized Tanstack Table, a powerful library suited for building interactive tables in React and TypeScript environments, for creating a general and highly versatile data table component, which can be adjusted to fit specific scenarios using props to adjust for example:

- Filtering: Allows users to filter data based on specific criteria.
- Sorting: Users can sort data according to different columns.
- Pagination: Manages data presentation in a paginated format.

Here's a snippet of how the table is set up:

```
1 function onClick(row: Row<Babybox>) {
2   router.push("/dashboard/babybox/" + row.getValue("slug"));
3 }
4 ...
5 <DataTable
6   columns={columns}
7   data={babyboxes}
8   sorting={[{ id: "name", desc: false }]}
9   rowClickAccessor={onClick}
10  hideColumns=["slug"]
11  filterColumnName="name"
12  onRefresh={onRefresh}
13 />
```

■ **Code listing 3.15** Using the DataTable component.

The code above includes the configuration of:

- `columns` defines the structure and headers of the table.
- `data` contains the array of babyboxes to be displayed.
- `sorting` sets the default sorting behavior.
- `rowClickAccessor` is a function that triggers when a row is clicked, navigating the user to the babybox page.
- `hideColumns`, `filterColumnName`, and `onRefresh` manage the visibility of columns, the column used for filtering, and a refresh function callback for reloading the data.

### 3.4.6 Babybox Page

The babybox page has been significantly enhanced to provide a more comprehensive and immediate understanding of each babybox's status at first glance. This page is central to monitoring the detailed performance of individual babyboxes and offers a richer user experience.

One of the key features of the babybox page is the inclusion of charts displaying the latest snapshots of data collected from the babybox. This visual representation helps users quickly grasp the current status and historical trends of various parameters such as temperature and voltage levels. Additionally, summary statistics for the last week, three days, and a day are presented alongside the chart for quick comparisons and trend analysis, allowing users to immediately detect any anomalies or significant changes in the babybox's conditions.

#### Babybox Domažlice

Data byly načtené v: 02:03:29 🔄

##### Teploty

Minimum, Průměr, Maximum



■ **Figure 3.11** Partial screenshot of variable overview widgets showing the first 3 variables - chart and minimum, maximum and average statistics over the last week, 3 days and 1 day.

Below the variable overviews, a data table displays the latest snapshots, detailing all recorded variables. Each entry in the table is accompanied by an icon indicating whether there was a significant change compared to the previous snapshot. This is achieved through a column definition in the table setup, which calculates the percentage change and displays an appropriate icon to reflect the increase, decrease, or stability of the value:

```

1   cell: ({ getValue, table, row }) => {
2     const val = getValue();
3     const valStr = val.toFixed(2);
4     const currentIndex = row.index;
5     const currentValue = row.getValue("temperature_inside");
6     const previousValue = currentIndex > 0
7       ? table.getRowModel().rows[currentIndex - 1]
8         .getValue("temperature_inside")
9       : undefined;
10    const percentageChange = previousValue && currentValue
11      ? calculatePercentageChange(previousValue, currentValue)
12      : 0;
13    const arrow = percentageChange > 1 ? (
14      <ArrowUpRight size={16} className="text-red-600..." />
15    ) : percentageChange < -1 ? (
16      <ArrowDownRight size={16} className="text-blue-600..." />
17    ) : (
18      <Minus size={16} className="text-slate-700..." />
19    );
20
21    return <div className="...">{valStr} {arrow}</div>;
22  }

```

■ **Code listing 3.16** Column definition (for the temperature inside column) to provide icons indicating a sudden change in the data.

This table is also supported by additional statistics that calculate the time from the last snapshot and the average gap between snapshots and visually indicate if these values within acceptable bounds.

Nejnovější data								
Čas	Vnitřní	Venkovní	Plášť	Horní	Dolní	Vstupní	Baterie	Status
2.5.24 01:00	27.56 —	19.93 —	46.31 —	25.75 —	26.00 —	15.28 —	14.08 —	OK
2.5.24 00:40	27.25 ↓	20.12 —	26.75 ↓	25.87 —	26.12 —	15.28 —	14.08 —	OK
2.5.24 00:30	27.56 ↗	20.31 —	27.62 ↗	25.87 —	26.12 —	15.28 —	14.08 —	OK
2.5.24 00:20	27.81 —	20.43 —	28.87 ↗	25.93 —	26.18 —	15.28 —	14.08 —	OK
2.5.24 00:10	28.06 —	20.68 ↗	30.75 ↗	25.93 —	26.18 —	15.28 —	14.08 —	OK
2.5.24 00:00	28.18 —	20.81 —	34.37 ↗	25.87 —	26.18 —	15.28 —	14.11 —	OK
1.5.24 23:50	28.06 —	20.81 —	42.31 ↗	25.81 —	26.12 —	15.28 —	14.11 —	OK
1.5.24 23:40	27.25 ↓	20.50 ↓	39.37 ↓	25.87 —	26.06 —	15.25 —	14.08 —	OK
1.5.24 23:30	27.12 —	20.50 —	26.12 ↓	25.93 —	26.12 —	15.28 —	14.08 —	OK
1.5.24 23:20	27.43 ↗	20.62 —	26.81 ↗	26.00 —	26.18 —	15.28 —	14.08 —	OK
1.5.24 23:10	27.68 —	20.87 ↗	27.62 ↗	26.06 —	26.25 —	15.28 —	14.08 —	OK

Zobrazeno 11 záznamů.  
 Poslední záznam je 3 minuty starý.  
 Průměrná doba mezi příchodem dat je 659.60 sekund.

■ **Figure 3.12** Screenshot of the table showing the latest snapshots.



The page also includes sections for displaying the latest events and notifications related to the babybox. Events are presented in another data table, providing a chronological view of occurrences that might affect the babybox's operation.

Notifications, on the other hand, are grouped by the notification template and displayed as badges further grouped by days, providing a clear visual of the frequency and types of alerts generated for the babybox.

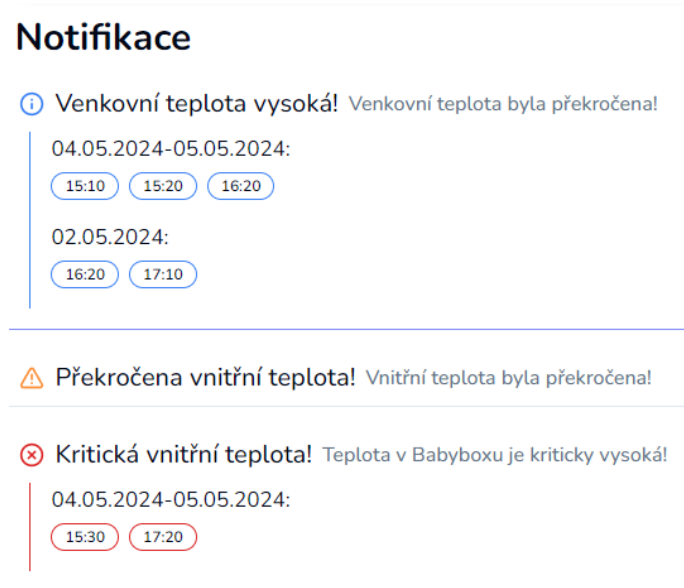
#### 3.4.6.1 Grouping notification

To achieve clarity in the design of displaying the notifications, we have implemented a series of pure functions that process notifications in a structured manner, ensuring the UI remains intuitive and user-friendly.

The first step in our notification processing pipeline involves grouping notifications by their associated templates. This is essential because notifications related to the same issue should be viewed collectively. Each notification carries a template ID, which we use to aggregate related notifications into groups, thus making it easier for users to navigate and interpret the data. This function transforms the `Notification[]` into `{ template: string; notifications: Notification[] }[]`.

Following this, the aggregated notifications are further organized by the day they were generated. This day-by-day grouping helps in identifying trends or issues that might be developing over time. This function transforms the output of the last function into a `{ template: string; days: { day: string; notifications: Notification[] } }[]`.

To enhance the usability of our notification system, we also combine notifications from consecutive days into single entries if applicable. This method simplifies the timeline of events, reducing the clutter in the UI and making it easier for users to track ongoing situations without getting overwhelmed by too much granular data.



■ **Figure 3.13** Notifications displayed using the accordion component grouped by their `template_id` and further grouped by the day on which they occurred.

The final structured data, processed through these functions, is ready for providing a clear and concise display of the notifications. The rationale behind using such a structured approach is to minimize the cognitive load on users who need to make decisions based on the notifications. By grouping and simplifying the notification data, we ensure that users can quickly grasp the situation without having to sift through disorganized information. The final structure is displayed through an accordion component to hide overwhelming amount of unnecessary notification data unless the user needs to see it.

### 3.4.7 Chart Page

The chart page is designed to offer a comprehensive view of data through an interactive and detailed chart. This page is central to the analytical capabilities of the application, allowing users to visually assess the operational status and historical data of babyboxes over selected periods.

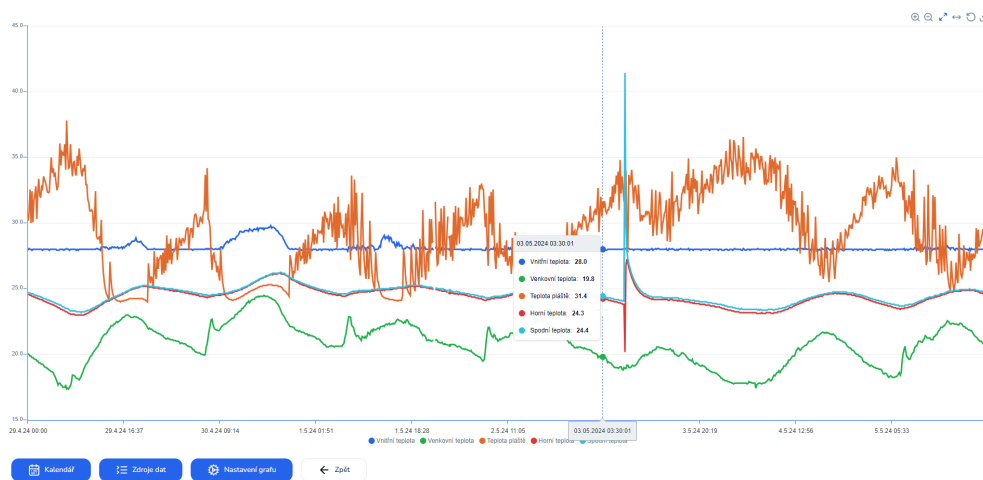
#### 3.4.7.1 Chart Component

The main feature of this page is a large, dynamic chart that displays all selected data types—temperature readings, voltage levels, and various event types—across a user-specified time range. This visualization is not only a tool for monitoring but also for analyzing patterns or anomalies over time, making it invaluable for proactive maintenance and quick responses to potential issues.

Beneath the chart, the interface includes a series of intuitive controls to customize the data display. Upon clicking the button, a drawer component appears, revealing the customization options:

- **Calendar Selection:** Users can adjust the time range for the chart data either by selecting dates directly from a pop-up calendar or by using quick-select buttons that automatically set ranges such as the last day, three days, a week, or two weeks.
- **Data Source Toggles:** There are switches to toggle the display of different data types, such as temperatures and voltages, allowing users to customize the chart to show only the relevant data they need.
- **Variable Visibility:** Further granularity is provided by controls on the chart itself, where users can hide specific data points or variables to focus on others.
- **Chart Customization:** Users can adjust visual aspects of the chart, such as the line stroke width and the type of data smoothing applied, choosing from options like no smoothing, standard smoothing, or cubic smoothing. These adjustments are made through a drawer component that organizes options neatly without cluttering the main interface.

Below the chart, the page displays statistics for the selected time period - minimum, maximum, and average values for each variable. This summary provides a quick numerical overview complementing the visual data representation.



■ **Figure 3.14** The main line chart displaying the temperatures over one week time.

Additionally, a detailed data table mirrors the chart's data in a structured tabular format. This table benefits from the same flexible data handling as

the chart, including pagination to manage large datasets efficiently, ensuring the interface remains responsive and user-friendly.

### 3.4.7.2 Consistent Color Coding

To enhance usability and visual coherence, each variable type is assigned a consistent color that is used throughout the application. This color coding is implemented using CSS variables and integrated with Tailwind CSS. For instance, the internal temperature variable might be defined as follows:

```
1  /* CSS configuration */
2  layer base {
3    :root {
4      ...
5      --inside: 221.2 83.2% 53.3%;
6      ...
7    }
8  }
9
10 /* Tailwind configuration */
11 ...
12 inside: {
13   DEFAULT: "hsl(var(--inside))",
14   foreground: "hsl(var(--inside))",
15 }
16 ...
```

■ **Code listing 3.17** CSS and Tailwind configurations working together to add custom colors.

Using this setup, we can easily apply these colors across different UI elements by using class names such as "text-inside" or "bg-inside", which are automatically created by Tailwind. This approach not only maintains a cohesive visual identity but also simplifies dynamic theming and responsive design adjustments across the platform.

### 3.4.7.3 SearchParams as State

For this page, we have leveraged URL search parameters to maintain and manipulate application state directly from the browser's address bar. This method proves particularly effective for sharing views and settings between users and for initializing the application with specific pre-configured states.

Search parameters, often referred to as query parameters, are appended to URLs and can influence the application state upon page load. By using these parameters, we can dynamically adjust what is displayed on the chart page, such as the time range or the data sources that are visible. The key benefits of this approach include:

- **Link Sharing:** When a user wants to share a specific view of the data with a colleague, they can simply send a URL containing the appropriate query parameters. This ensures that the recipient sees exactly the same data state as the sender, which is invaluable for collaboration and troubleshooting.
- **Direct Navigation:** This method allows us to create links that navigate directly to a pre-configured view. For instance, a link could lead a user to a chart that only displays temperature data, simplifying navigation and enhancing user experience.

Implementing this functionality in a Next.js application is made easy thanks to the `useSearchParams()` hook. This hook provides a way to access and manipulate the URL's query parameters within our React components. We use this information on page load, when we initialize the state of the application from the search parameters. Then we track the state as the user makes changes to the settings of the chart. For example, to update the time range, we might use the following code snippet:

```
1  const [searchParams, setSearchParams] = useSearchParams();
2  ...
3  const from = searchParams.get("from");
4  const to = searchParams.get("to");
5  ...
6  const updateDateRange = (dateRange) => {
7    const existing = Object.fromEntries(searchParams?.entries() ?? []);
8    setSearchParams({
9      ...existing,
10     from: dateRange.from,
11     to: dateRange.to,
12   });
13   setDateRange(dateRange);
14 };
15 ...
16 <Link href={`~/dashboard/babybox/${slug}/chart?sources=temperature`} >View
   ↪  Temperatures</Link>
```

■ **Code listing 3.18** Updateing search from and to parameters in Next.js.

This approach not only updates the URL but also ensures that the internal state of our application remains in sync with the URL, providing a seamless user experience. As an example of this in action could be a link that directs a user to the chart page with only the temperatures being shown.

#### 3.4.7.4 ApexCharts Performance

In working with ApexCharts to develop the line chart component, we encountered performance problems, particularly when dealing with extensive datasets.

As our solution involved rendering potentially thousands of data points, performance issues arose with our initial choice of the ApexCharts library. To address the lag and sluggishness in chart rendering, we implemented several optimization strategies:

- **Optimizing Data Handling:** We switched the x-axis configuration to a `datetime` type, which is more efficient for handling time-series data in ApexCharts. This adjustment significantly reduced the computational load during chart plotting.
- **Reducing Visual Complexity:** We disabled animations, markers, and data labels within the charts. Each of these features, while visually appealing, adds to the processing time required to render the chart, especially when dealing with large volumes of data.
- **Delaying Chart Rendering:** To ensure that the user interface remains responsive, we implemented a strategy to delay the rendering of the chart until all other components of the page had loaded. This was achieved through a controlled use of React's `useState` and `useEffect` hooks:

```
1  const [display, setDisplay] = useState<boolean>(false);
2  useEffect(() => {
3      setDisplay(false);
4      setTimeout(() => setDisplay(true), 1);
5  }, []);
```

■ **Code listing 3.19** Code for delaying the render of the line chart component.

This approach prevents the chart from blocking the initial load of the page, thereby improving the perceived performance.

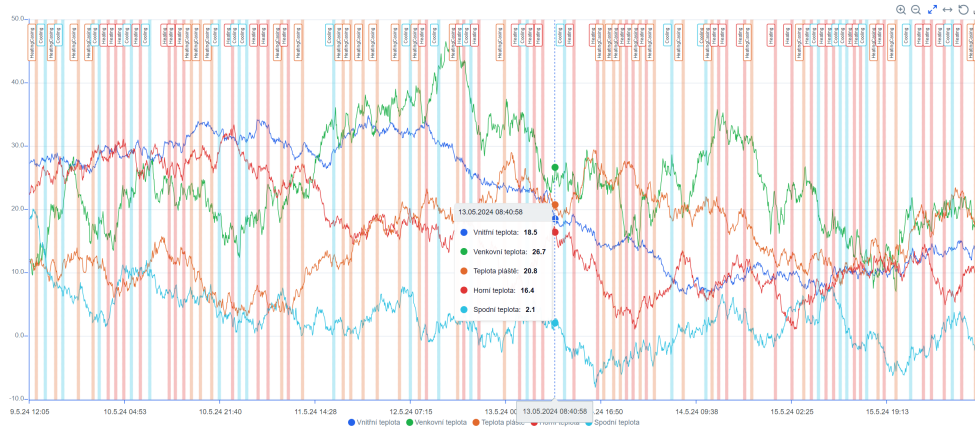
- **User Feedback During Load:** We integrated a skeleton component that appears while the chart is loading. This visual cue informs users that data is being processed and helps manage expectations regarding load times.

Despite these improvements, the performance of the chart component is still not at an ideal level. The current measures have mitigated the issue to some extent, but have not completely eliminated performance bottlenecks. As a result, we are considering further strategies to enhance chart performance, including excluding other performance optimizations but also exploring alternative libraries.

#### 3.4.7.5 Displaying Event Data

Visualizing events as intervals on the chart enhances clarity and user understanding of the context of the data. This visualization helps users quickly

grasp periods during which specific conditions, like heating or cooling, were active.



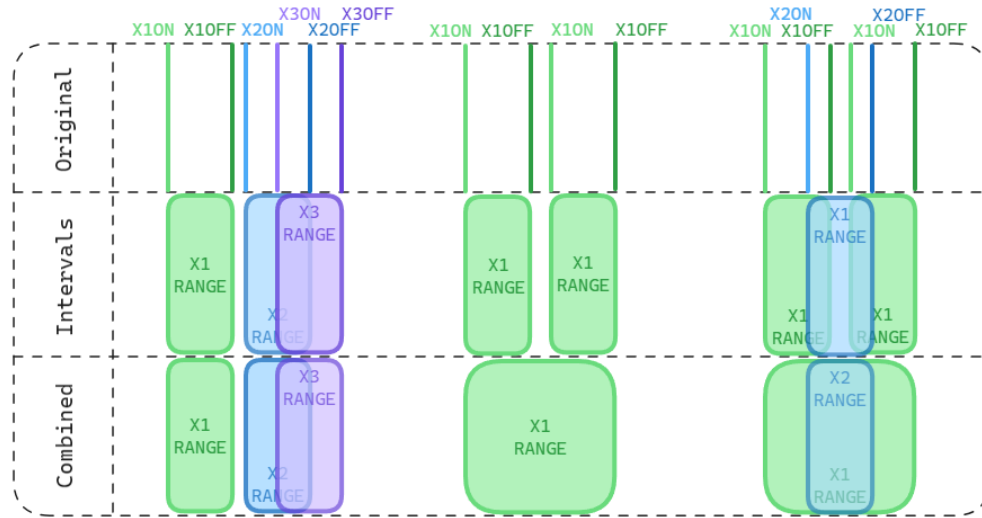
**Figure 3.15** The main chart displaying the color coded events with labels over one week of time.

To achieve this, we developed a method to transform discrete event data into continuous intervals to then display as annotations on the chart.

Each event in the system, such as “heating on” or “cooling off”, marks a change in state. Our goal is to map these events to intervals that represent the duration each state was active.

1. **Initial Collection:** As we process the array of events, we keep track of ongoing events using a dictionary. Each event type, like heating or cooling, is monitored separately.
2. **Event Termination:** For each new event, we check if it should terminate any ongoing events. This involves a predefined array of terminating events for each type. For example, “heating off” or a generic “babybox reset” event might end a “heating on” interval. Including broad reset events ensures that all states are correctly reset, even if a specific “off” event was missed due to data gaps or errors.
3. **Building Intervals:** As we find starting and terminating events, we create intervals marked by a start and end timestamp. These intervals are stored as soon as both boundaries are identified.

We create these intervals in one sweep and then we combine the intervals of the same type that occur close together into a single interval. This reduces the number of intervals displayed and focuses the user’s attention on significant changes. While this method may sacrifice some precision, it enhances performance and reduces visual clutter, making the chart more accessible and easier to understand.



■ **Figure 3.16** Visualization of the intervals algorithm creating and then combining the intervals together.

The attached figure illustrates this algorithm in action. It shows how individual intervals are constructed based on event data and then combined based on their proximity and type.

### 3.4.7.6 Gap Filling Algorithms

The approach to fetching snapshot data is tailored to suit different display needs, ensuring that the data representation is both accurate and efficient. For displaying the latest snapshots, the application fetches the data directly as they are, without any modifications. This straightforward approach is suitable for simple display, where the primary focus is on the most recent data points, and there is no need to highlight gaps in data collection.

However, when it comes to visualizing the data on the chart, simply displaying raw data can be misleading due to the presence of gaps. These gaps might not be visually apparent when the data points are connected directly, which could misrepresent the actual monitoring situation. To address this, we implement a filling algorithm, as previously detailed, opting for the “lazy” filling strategy for chart displays. This method adds a single data point for each gap, sufficiently indicating the presence of a gap while minimizing the number of additional data points. This approach is particularly important for maintaining performance with ApexCharts, which can struggle with large datasets.

Below the chart, there’s a need for a more detailed and exhaustive view of the data. Here, we apply the full filling method, which inserts data points for all missing snapshots within the selected timeframe. This ensures that users can see a complete timeline, including periods where data was not received,

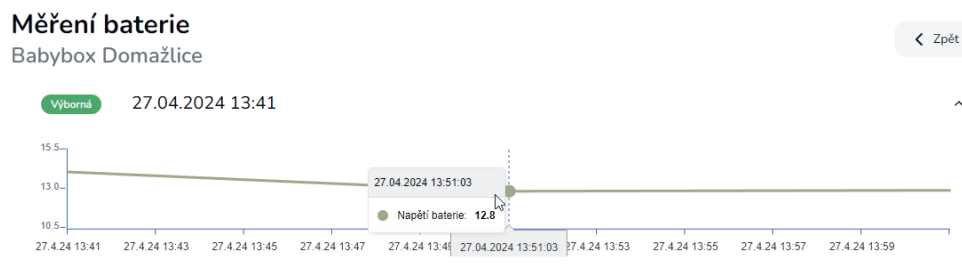


providing a comprehensive overview necessary for thorough analysis.

### 3.4.8 Battery Analysis Page

The battery analysis page serves as a tool for monitoring battery performance, focusing on visually representing the battery's voltage fluctuations during power outages. Each measurement taken by the battery analyzer microservice is displayed using the accordion component. Upon expanding the accordion a chart appears that tracks the voltage of the battery from the onset of a power outage to its conclusion or for a maximum duration of three hours. The chart provides a clear visual history of the battery's voltage stability or decline during these periods.

While the user interface is designed to potentially display the estimated quality of the battery, this feature does not provide accurate information due to the challenges in determining battery health from short monitoring intervals.



■ **Figure 3.17** User interface of the battery measurement.

#### 3.4.8.1 Babybox Information Page

The babybox information page is a dedicated section within the application that displays detailed information about each babybox. This page provides a comprehensive view of user-inputted details including the babybox's name, location, network configuration, and contacts associated with the hospital staff. For added convenience, the location information includes interactive buttons that link to Google Maps or Mapy.cz, facilitating quick navigation directly from the interface.

At the top of the page, there is an edit button that enables users to update the babybox's information. Clicking this button transforms the display into a form layout, retaining the original data structure but replacing static text with form inputs. This allows users to easily modify the details as needed, ensuring that the database remains current and accurate.

### 3.4.9 Notification and User Pages

The notification and user pages serve specific administrative functions, allowing users to manage notification templates and user accounts respectively. Each page presents a table listing the existing records, such as notification templates or user details, with each row featuring an action column. This column provides options to edit or delete the record directly from the table, facilitating quick and efficient management. Deleting a record is protected by a modal popup to prevent deleting something accidentally.

Additionally, both pages include a prominent button that navigates users to a form where they can create new notification templates or register new users.

## Testing and Evaluation

*The following chapter outlines the practical approach taken to validate the functionality and performance of the system. This includes detailed descriptions of our methods for unit testing and API testing. Subsequent sections delve into the evaluation of both functional and non-functional requirements, assessing how well the system meets the specified criteria. Additionally, this chapter presents insights from user feedback.*

### 4.1 Testing

During development, the approach to testing was very pragmatic, shaped by the constraints of resources and the need to ensure functional reliability within those limits. This resulted in a focused strategy, prioritizing certain types of tests over others to maximize the impact of the testing effort on the application’s overall quality.

#### 4.1.1 Unit testing

Keeping in mind the manpower constraints, an emphasis was placed on unit testing of the logic and core functionalities, facilitated by the practice of Test-Driven Development (TDD).[61]

Test-driven development requires writing tests for each function or feature before implementing the actual code, starting with defining a clear “contract” for the function’s inputs and outputs. This method ensures clarity, as each unit of code is designed to fulfill a specific, pre-defined role, reducing ambiguity in implementation. Functions are crafted to be pure, meaning they operate without side effects and do not rely on or alter the external state, which simplifies updating and refactoring since changes in one part of the system are less likely to impact other parts. Such functions are also easier to think about.

By isolating functions and defining clear input-output behaviors, the tests can run in isolation, making them simpler to write, execute, and debug. This

isolation allows for pinpointing the exact location of defects within the codebase.

```
1 describe("groupNotificationsByTemplate", () => {
2     test("groups an empty array of notifications", () => {
3         const notifications: Notification[] = [];
4         const grouped = groupNotificationsByTemplate(notifications);
5         expect(grouped).toEqual([]);
6     })
7     test("groups multiple notifications under the same template", () =>
8         ↪ {...})
9 })
10 describe("groupNotificationsByDay", () => {...})
11 describe("mergeConsecutiveDays", () => {...})
describe("processNotifications", () => {...})
```

■ **Code listing 4.1** Example of a test suite for the notification pipeline of grouping them by template ID, day of year and then merging together.

On the code example, we can see how thinking in TDD way makes code naturally split into pure functions with single concern while enforcing defining the types of inputs and outputs of the functions. The usage of TDD on itself makes the code adhere to a higher quality; in total we made around 50 test cases throughout the application.

This focused approach to unit testing, embedded within the TDD framework, ensured that despite limited resources, we could maintain a high standard of quality and functionality. We found that this approach sped up the development process both in the long term and in the short term as we got high confidence when the tests passed.<sup>[61]</sup>

### 4.1.2 Testing the API interface

During the development of the application, we leveraged Postman extensively for manual API testing. Postman's user-friendly interface facilitated the testing of various API endpoints, significantly speeding up the process compared to traditional methods like using curl commands. The tool allowed for easy manipulation of API requests and visualization of responses, enabling us to quickly adjust inputs and query parameters to observe the effects.

One of the substantial advantages of using Postman was its collaborative features, which allowed us to synchronize our API tests and configurations across different development environments seamlessly.

Postman proved particularly useful when integrating and testing authentication mechanisms within our application. The tool supports a variety of authentication schemes, including JWT. By handling authentication tokens dynamically, Postman allowed us to simulate authenticated sessions effectively,

facilitating thorough testing of secured endpoints without the repetitive manual input of credentials.

Setting up JWT for authentication involves configuring the Authorization tab of a request to use a Bearer Token. Once configured, Postman can automatically append the JWT to the Authorization header of the HTTP request, ensuring that all subsequent requests within a collection are properly authenticated. This streamlined our testing process, especially when dealing with multiple endpoints that required authentication.

As we progressed in the development process, we started using Postman for regression testing. Once our APIs were established and functional, we utilized Postman's automation features to conduct regression tests. These tests were designed to ensure that newly introduced changes did not affect existing functionalities. With Postman, we could automate checks for correct status codes, validate response bodies, and ensure that our API's contracts remained consistent over time.

For automated testing, Postman's ability to run collections of requests through its built-in runner or via Newman, a command-line Collection Runner, allowed us to automate the execution of scenarios for our endpoints. We could script tests to assert various aspects of the API response, such as the HTTP status codes, response payload, and the correctness of headers, directly within Postman.

```
1  pm.test("Status code is 200", function () {
2      pm.response.to.have.status(200);
3  });
4  pm.test("Token is valid", function () {
5      var jsonData = pm.response.json();
6      pm.expect(jsonData.data.token).to.be.a('string');
7  });
8  pm.test("Content type is present and correct", function () {
9      pm.response.to.have.header("Content-Type", "application/json");
10 });
11 pm.test("Response has the correct username", function () {
12     var responseJson = pm.response.json();
13     pm.expect(responseJson.data.username).to.eql("zbynek");
14 });
```

■ **Code listing 4.2** Examples of API testing using Postman to check the status code, headers and data payload.

This approach not only maintained the integrity of our APIs but also boosted our confidence in the system's stability with each new release, minimizing the risk of introducing regressions into the live environment.

We have also used Postman for performance testing and evaluating the non-functional requirements.

## 4.2 CI/CD Pipelines

In this project, we set up a CI/CD pipeline on GitHub to ensure consistent code quality, automated testing, and efficient deployment. The pipeline runs on every push to the `main` branch and is composed of several stages to ensure robustness and security throughout the development and deployment processes.

### 4.2.1 Continuous Integration

The first stage of the pipeline involves running all the test suites for each microservice. The pipeline configuration leverages GitHub Actions to wait for changes and execute unit tests, ensuring that any new code maintains the existing functionality without introducing regressions.

```
1  on:
2    push:
3      branches:
4        - main
5  jobs:
6    run_cicd:
7      runs-on: ubuntu-latest
8      steps:
9        - name: Checkout Code
10         uses: actions/checkout@v4
11        - name: Set up Node.js
12         uses: actions/setup-node@v3
13         with:
14           node-version: "20"
15        - name: Install npm dependencies for Web App
16         working-directory: apps/web
17         run: npm install
18        - name: Run Tests for Frontend
19         working-directory: apps/web
20         run: npm run test
21        - name: Set up Go
22         uses: actions/setup-go@v3
23         with:
24           go-version: "1.22"
25        - name: Run Tests for Snapshot Handler
26         working-directory: apps/snapshot-handler
27         run: go test ./...
28        ...
```

■ **Code listing 4.3** Snippet of the Continuous Integrity part of the pipeline for running automated tests.

The pipeline installs all the dependencies, sets up the environment for the service itself and then runs the tests.

### 4.2.1.1 Additional Workflows

In addition to running test suites, we included automated code analysis and dependency management as another two workflows added to our GitHub Actions.

CodeQL analyzes the codebase for potential security vulnerabilities and coding issues. It checks for common security flaws, enabling us to identify and resolve them early.

Dependabot ensures that all dependencies remain up-to-date and secure by periodically scanning for outdated or vulnerable packages and generating pull requests to update them.

## 4.2.2 Continuous Deployment

Our production server is managed on-premise, requiring secure SSH access for deployments. We set up SSH port forwarding and adjusted firewall settings to allow the GitHub pipeline to connect directly to the server.

Then we utilize GitHub Secrets to securely store the server's private SSH key and other necessary variables such as the username. The CI/CD pipeline retrieves the private key during deployment to authenticate via SSH.

```
1  ...
2  - name: Install SSH Keys
3    run: |
4      mkdir -p ~/.ssh
5      chmod 700 ~/.ssh
6      install -m 600 -D /dev/null ~/.ssh/id_rsa
7      echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_rsa
8      ssh-keyscan -p ${{secrets.SSH_PORT }} -H ${{ secrets.SSH_HOST }} >
9      ↪ ~/.ssh/known_hosts
10     chmod 644 ~/.ssh/known_hosts
11  - name: Connect and Deploy
12    run: ssh -p ${{ secrets.SSH_PORT }} ${{ secrets.SSH_USER }}@${{
13     ↪ secrets.SSH_HOST }} "cd ${{ secrets.WORK_DIR }} && git checkout ${{
     ↪ secrets.MAIN_BRANCH }} && git pull && docker compose up --detach
     ↪ --build && exit"
```

■ **Code listing 4.4** Snippet of the Continuous Delivery part of the pipeline for automatically deploying to a remote server.

After successfully connecting to the server, the pipeline downloads the newer version of the application using Git and builds the application using Docker Compose. This automatically redeploys the application with the newer code.

## 4.3 Documentation

Technical documentation is integrated within the codebase to ensure that developers and technical staff have immediate access to necessary information:

- **Code Comments:** Code is supplemented with comments that explain complex logic and decisions, making the code self-explanatory and easier to understand or modify.
- **README.md Files:** Every microservice and the front-end repository contain a README.md file that provides an overview of the service, setup instructions, and usage details, serving as a quick reference for developers.
- **Commit Messages:** The commit history is utilized as part of the documentation strategy. Detailed commit messages include a body that explains the rationale behind major decisions, providing a chronological documentation of changes and reasoning.
- **Internal Wiki:** Comprehensive details, architectural decisions, and project guidelines are documented in an internal wiki, which serves for other projects as well.

To ensure that end-users can effectively utilize the system, user documentation is embedded directly within the application:

- **Help Page:** The application includes a dedicated help page accessible to the users. This page houses detailed documentation on how to use various features of the application, practical tips, and troubleshooting steps.
- **Contextual Help:** Tooltips and contextual help elements are distributed throughout the application interface, providing users with instant guidance and clarifications on functionality directly within their workflow.

## 4.4 Evaluation and User Feedback

This section provides an analysis of how effectively the application meets its intended functional and non-functional requirements, as established during the initial analysis phase. By incorporating continuous user feedback and rigorous testing into the development process, this section discusses how the system has evolved to improve user experience and system performance, setting a baseline for ongoing adaptation and growth.

### 4.4.1 Requirements Fulfillment

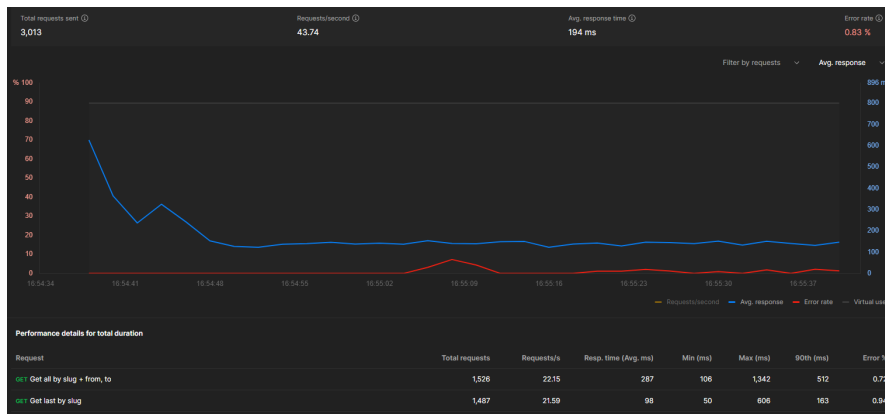
We successfully implemented all the functional requirements identified during the initial analysis phase. To continuously improve and adapt the system,



we will persistently monitor user interactions and gather feedback to refine features and address any emerging challenges. Over time, some aspects of the system may require adjustments based on additional data and user experiences. The last chapter of this work discusses future improvements and directions we would like to explore.

Regarding the non-functional requirements, we focused on performance, security, and maintainability:

**Performance:** We leveraged Postman’s performance testing tools to evaluate the API’s response under stress. We simulated a load with 100 virtual users to test critical API endpoints over a one-minute interval, specifically observing the 90th percentile response time. The performance metrics from this test, particularly for complex queries like retrieving a week’s worth of gap-filled snapshot data, remained within acceptable limits. This testing phase was conducted in the production environment to account for all real-world operational variables, including potential overhead from network components like the API gateway.



■ **Figure 4.1** Performance testing in Postman.

The results of this test were very positive, as the 90th percentile latency was well under 1 second even for very expensive endpoints.

**Security:** Our use of Caddy as an API gateway has significantly streamlined our security setup. Caddy automates the deployment of HTTPS, ensuring encrypted communication channels by default and automatically redirecting HTTP requests to HTTPS. This setup ensures the use of HTTPS for all user requests.

**Maintainability:** While it’s too early to assess the maintainability of the system since it has only recently been deployed, our design and technology choices were made with long-term maintainability in mind. Future evaluations will focus on the system’s adaptability to changes and the ease with which the development team can implement updates and resolve issues.

As we move forward, these initial evaluations will serve as a baseline for

continuous improvement, ensuring that the system not only meets current user needs but also adapts to future demands and technological advances.

### 4.4.2 User Feedback

Throughout the development process, we actively engaged with a user from both user groups (operations and maintenance) to gather continuous feedback on each new feature (even those heavily inspired from the previous solution).

We let the users use the application for a limited time based on the task to accomplish that we gave them (e.g., create a notification template), and by observing their interactions with the feature/part of the application we were focusing on and posing open-ended questions, we gained valuable insights through their direct feedback or by observing the struggles they were having, which ultimately guided our enhancements and adaptations of the system.

This approach ensured that we retained the effective elements of the previous solution while introducing modernized functionalities and improvements.

Key components of the application, such as the initial table of babyboxes and the comprehensive chart page, were deliberately kept similar to maintain user familiarity. These sections were, however, augmented with additional functionalities to enhance user experience. The babybox page, in particular, underwent significant redesigns to transform it into a more useful and informative hub.

The notification system was another area of focus, redesigned to mitigate notification fatigue—a challenge with the previous system. Inspired by the earlier implementation but refined to better meet current needs, the new system has been well-received by the staff. It balances familiarity with innovation, making the transition smoother while improving their daily workflows.

Feedback regarding the application's performance has been overwhelmingly positive, especially concerning its responsiveness and the seamless loading experience—even during data retrieval.

Looking forward, we aim to improve the application for maintenance workers and are interested in their insight as they start using this system as they were not included in the usage of the previous application.

The introduction of PWA functionality, improved mobile and tablet support and enhanced responsiveness are steps toward making the application more accessible and useful in more situations.

## Conclusion

In this thesis, the primary objective was to develop a microservice-based application for monitoring and managing data from Babyboxes distributed across the Czech Republic. This system was specifically designed to enhance the efficiency and effectiveness of both operational staff and maintenance technicians interacting with these devices.

The project began with an extensive analysis of existing solutions and relevant technologies, which informed the conceptualization and design of a new, more capable system. This involved a thorough review of the previous system's architecture and user requirements, leading to the selection and implementation of appropriate technologies that would support a microservice architecture.

In the implementation phase, we established a robust infrastructure that sets the foundation for future expansions and enhancements of the application with a CI/CD pipeline to run automated tests and deploy to our server, setting up a versatile environment that supports both production and development modes, featuring streamlined deployment processes and hot reloading capabilities in development to facilitate rapid iteration and testing.

We enhanced the data collection mechanisms from the babyboxes, and developed multiple microservices, each tasked with specific capabilities to provide data collection, visualization, aggregation, analysis along with a custom notification system. We developed a PWA application with a modern user interface focused on user experience to enable efficiency for working with the application.

Feedback from staff has been integral throughout the development process, driving continuous improvement and adaptation of the application to better meet the needs of the staff. With the introduction of the system to maintenance workers, a new user group, the application now provides new functionalities to support their specific requirements.

This work has demonstrated that even with inherent complexities and resource demands, a microservice architecture can achieve a balance that offers

maintainability and scalability, suitable for projects with limited budget and manpower. Developed by a single developer, this project highlights how, with the correct approach and careful balancing, microservices can be beneficial even for smaller projects. By clearly defining boundaries and responsibilities within the system, the architecture reduces cognitive load, making the system more manageable and less overwhelming than a monolithic architecture might be. Thus, the architecture not only supports current operational needs but also provides a flexible foundation for future enhancements and integrations. This setup proves that even with minimal resources, a well-considered microservice approach can significantly benefit project development by simplifying reasoning about the codebase and enhancing system comprehensibility.

## 5.1 Future Improvements

This work has laid a solid foundation for numerous future enhancements. As we look toward the future, we aim to introduce new groundbreaking capabilities while also iterating over existing functionalities to enhance both the user and developer experiences. This continuous evolution is vital for maintaining the relevance and effectiveness of the system, ensuring it remains a robust and adaptive tool for operational staff and maintenance technicians.

### 5.1.1 Kubernetes

We made a deliberate effort to avoid premature optimization, focusing instead on establishing a robust and flexible foundation for the application. As we look to future improvements, one potential avenue is the adoption of Kubernetes for deployment. Kubernetes is a powerful orchestration tool widely used in managing microservices, providing enhanced scalability and resilience through its automated container deployment, scaling, and management capabilities.

Currently, the system does not demand the high-level scalability that Kubernetes is designed to handle, as the number of babyboxes and user requests remains relatively stable and manageable. However, the future could bring more complex challenges, such as an increase in the types and volumes of snapshots and events sent from babyboxes. If babyboxes begin sending more complex or frequent data—particularly as more devices are upgraded to newer firmware versions—the load on the system could increase to a point where Kubernetes could start making sense for us to adopt.

### 5.1.2 Analysis Services

The development of additional analysis services stands out as an obvious next step. The introduction of a service dedicated to monitoring the electrical relays within the babyboxes could provide significant insights into their operational

health and longevity. By analyzing the frequency and patterns of relay triggers, captured through event data, we can predict their lifespan and identify potential failures before they occur. This predictive maintenance capability would not only enhance the reliability of the babyboxes but also optimize maintenance schedules.

Similarly, tracking the operational duration of the fans within each babybox could allow us to estimate their wear and service life accurately. By aggregating runtime data, this service could alert maintenance personnel to upcoming fan replacements.

Another enhancement would be the creation of a comprehensive service that consolidates data from various sources to produce detailed reports on the status of each babybox. These reports could serve as official documents during maintenance visits, providing hospital staff with a transparent overview of the babybox's condition and performance over time. Such documentation could improve communication with hospital personnel and ensure that all parties are informed about the babybox's operational status and any upcoming maintenance needs.

### 5.1.3 Maintenance Microservice

Introducing a maintenance service would significantly enhance its utility for maintenance technicians and operators. This service would function as a centralized issue tracking system, recording both historical and current problems encountered with each babybox, alongside their respective resolutions.

By systematically cataloging issues, this service would provide technicians with a prioritized list of tasks during maintenance visits, ensuring that the most critical problems are addressed promptly. This could dramatically improve the efficiency of maintenance operations, as technicians would arrive well-informed and prepared to tackle specific issues.

For hospital staff and administrators, the maintenance service would serve as a valuable repository of information. By generating detailed reports based on the tracked data, the service could offer insights into the frequency and nature of problems, highlight potential areas for improvement in babybox design or usage, and provide a record of maintenance history for each unit. These reports could be useful for archival purposes, helping to maintain a clear historical record of each babybox's operational integrity and maintenance needs.

### 5.1.4 Improving Current Components and Services

Expanding and enhancing the Babybox Dashboard's capabilities could significantly boost its efficiency, user experience, and overall effectiveness:

- **Infrastructure:** Implementing a more robust testing strategy, including automated tests with higher coverage, would be very beneficial. Introducing a dedicated testing environment would allow for isolated testing

scenarios that do not affect the production database, ensuring that new features and updates can be validated thoroughly before deployment.

- **Babybox Service:** Enhancing data retention capabilities by storing more information about each babybox, such as components or relevant documentation in PDF format, could greatly aid maintenance efforts. Additionally, enabling the service to generate and archive essential files automatically would help many operational processes.
- **User Service:** Improving the synchronization between the user service and the front-end, particularly in terms of session management, would enhance user experience. Implementing auto-renewal of authentication tokens could prevent disruptions in user sessions, maintaining a seamless user experience.
- **Notification Service:** Integrating outputs from various services to enrich the notification conditions and options could lead to more tailored and relevant alerts.

Exploring methods to aggregate notifications into a single email based on set intervals or severity could reduce notification fatigue significantly.

Additionally, introducing daily summary emails and drawing more attention to the unread notifications within the user interface could ensure critical information is promptly noticed and acted upon.

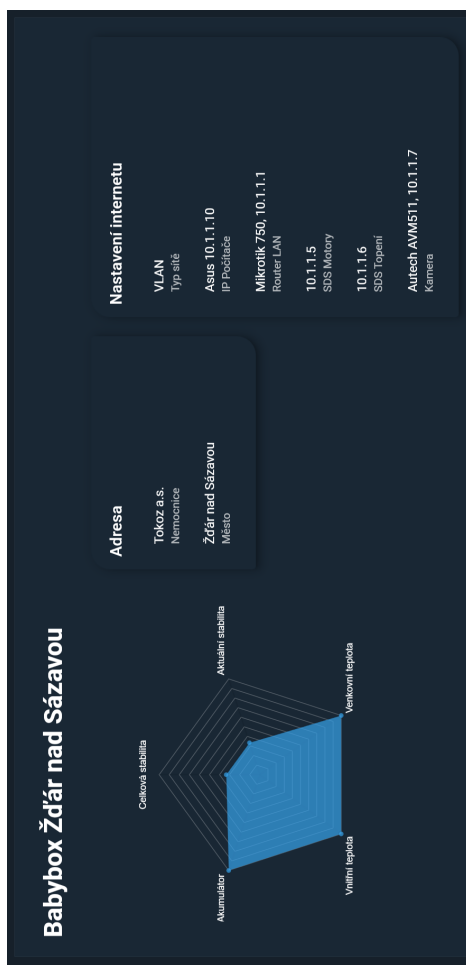
- **Front-End Enhancements:** The front-end could be improved by investigating alternative charting libraries or techniques that handle large datasets more efficiently. This would address performance issues currently experienced with ApexCharts and enhance the data visualization aspects of the application.

Another big feature to add to the front-end would be push notifications as part of the PWA functionality that would provide real-time updates to users, enhancing the responsiveness and interactivity of the application.

# Appendix A

## Previous Solution Screenshots

### A.1 Babybox Detail Page



## A.2 Babybox Page

The screenshot displays the Babybox web interface. The top section features a live camera feed of a BabyBoo device mounted on a wall. Below the feed, a data dashboard provides real-time information:

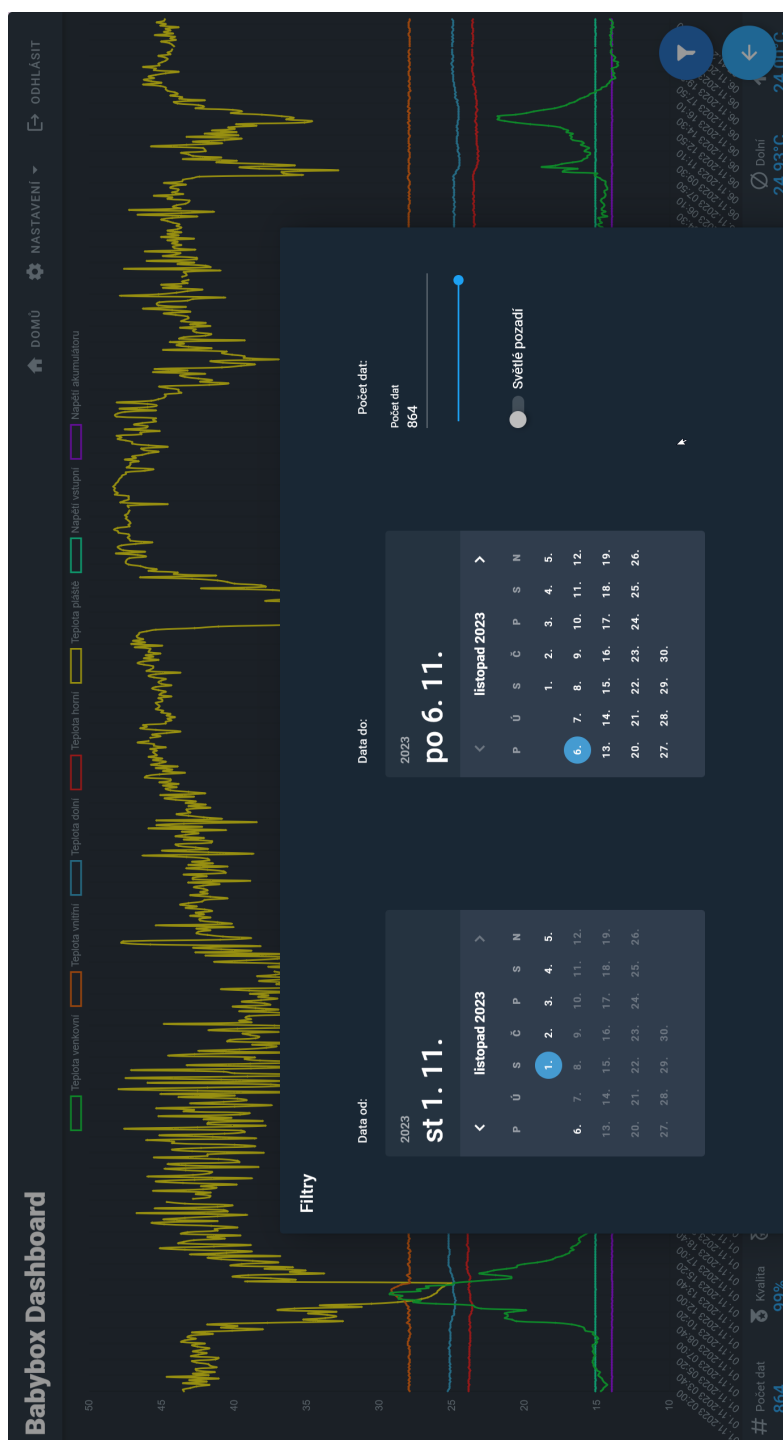
- Count:** 3
- Quality / Year:** 33%
- Battery:** 13.99V
- Internal Temperature:** 28.00°C
- External Temperature:** 17.81°C
- Internal Temperature (Right):** 28.00°C
- External Temperature (Right):** 17.81°C
- Bottom Temperature:** 25.37°C
- Top Temperature:** 24.31°C

Below the data, the interface includes the following sections:

- Babybox Ždár nad Sázavou**
- Adresa:** Tokoz a.s., Ždár nad Sázavou
- Možnosti (Options):**
  - VICE O BABYBOXU
  - UPRAVIT
  - VŠECHNA DATA
  - NAPĚTÍ
  - KONTAKTY
  - TEPLoty



## A.3 Time Filtering on Chart Page



## Appendix B

# New Solution Screenshots

### B.1 Babybox Detail Page

#### Babybox Benešov

##### 📍 Umístění

Nemocnice:  
Nemocnice Rudolfa a Stefanie  
Benešov

Město:  
Benešov

Ulice:  
Máchova 400

Směrovací číslo:  
256 30

📍 Souřadnice:

Zeměpisná šířka  
49.78734967954269

Zeměpisná délka  
14.679603221946026

[Google Maps](#)

[Mapy.cz](#)

##### 🔌 Síťová Konfigurace

Nemocnice:  
VLAN

Router:  
10.1.1.1/24

Motorová jednotka:  
10.1.1.5

Jednotka topení:  
10.1.1.6

Kamera:  
10.1.1.7

PC:  
10.1.1.10

Gateway:  
DHCP

##### 📞 Kontaktní Informace

Jméno	Příjmení	Role	Email	Telefon	Poznámka
John	Doe	Nurse	john.doe@hospital.com	123 456 789	Contact information for display purposes

## B.2 Babybox Table with Tooltips

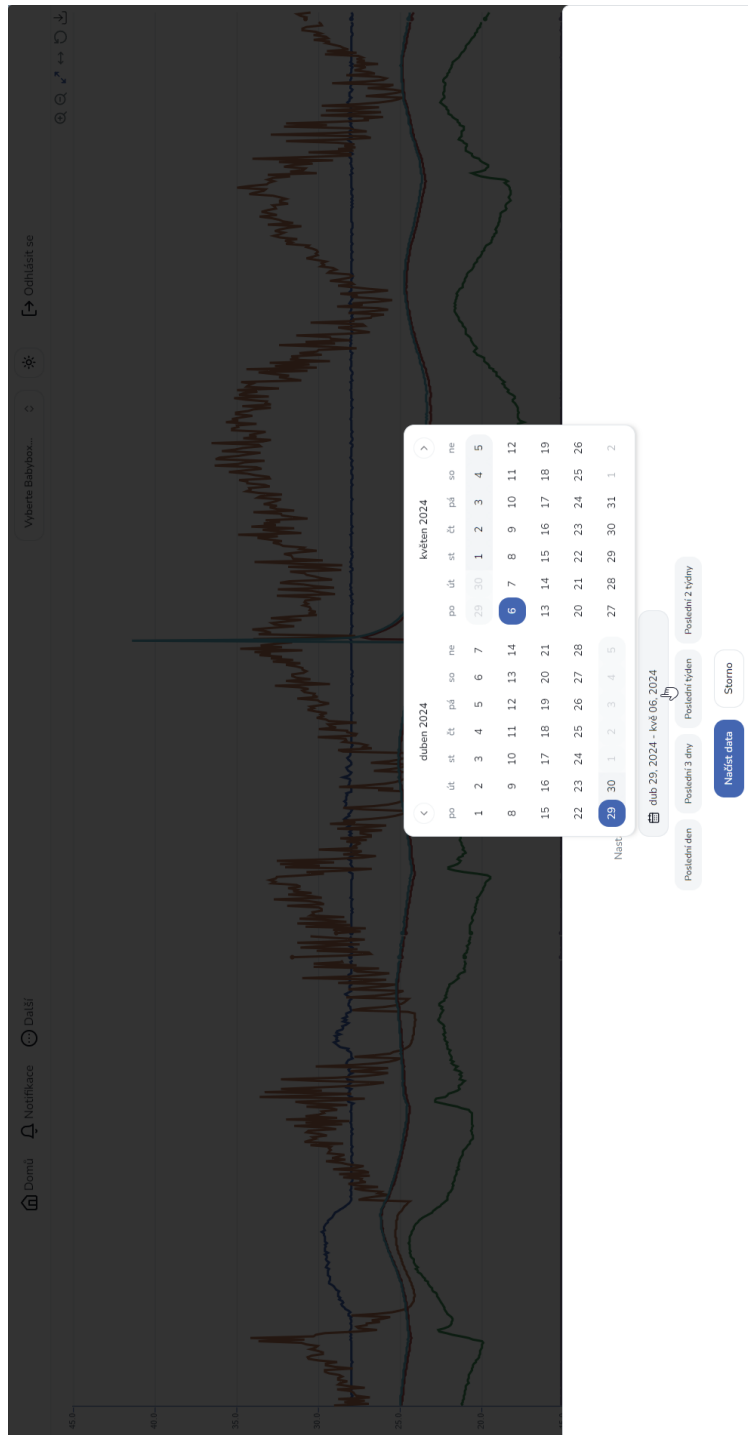
Babybox ↕	Vnitřní [°C]	Venkovní [°C]	Vstup [V]	Baterie [V]	Čas dat	Status
Benešov	28.00	19.75	15.16	14.11	00:00 06.05.2024	OK
blansko	28.18	20.93	15.19	13.99	00:00 06.05.2024	OK
breclav	28.00	22.25	15.22	14.11	23:59 05.05.2024	OK
brno	28.87	22.75	14.92	13.75	23:54 05.05.2024	OK
cbudejovice	27.93	22.18	15.16	14.05	00:00 06.05.2024	OK
ckrumlov	28.31	19.81	15.28	14.11	00:00 06.05.2024	OK
clpa	27.06	19.81	15.22	14.17	00:00 06.05.2024	OK
decin	27.50	18.56	14.89	13.81	00:00 06.05.2024	OK
Domažlice	27.12	18.00	15.25	14.08		
frkviňohradý	27.31	19.25	15.28	14.23	01:10 15.04.2024	Error

Data nepřišla - poslední záznam je více než 12 minut starý.

Babybox ↕	Vnitřní [°C]	Venkovní [°C]	Vstup [V]	Baterie [V]	Čas dat	Status
Benešov	28.00	19.75	15.16	14.11	00:00 06.05.2024	OK
blansko	28.18	20.93	15.19	13.99	00:00 06.05.2024	OK
breclav	28.00	22.25	15.22	14.11	23:59 05.05.2024	OK
brno	28.87	22.75	14.92	13.75	23:54 05.05.2024	OK
cbudejovice	27.93	22.18	15.16	14.05	00:00 06.05.2024	OK
ckrumlov	28.31	19.81	15.28	14.11	00:00 06.05.2024	OK
clpa	27.06	19.81	15.22	14.17	00:00 06.05.2024	OK
decin	27.50	18.56	14.89	13.81	00:00 06.05.2024	OK
Domažlice	27.12	18.00	15.25	14.08		
frkviňohradý	27.31	19.25	15.28	14.23	01:10 15.04.2024	Error

Data nepřišla - poslední záznam je více než 12 minut starý.

## B.3 Time Filtering on Chart Page



## B.4 Statistics and Tabular Data under the Chart

Kalendář

Zámky dat

Nastavení grafu

← Zpět

### Statistiky

Z vybraného období

Vnitřní teplota

28.13

↓ 27.87 29.81 ↑

Venkovní teplota

20.79

↓ 17.43 24.50 ↑

Teplota pláště

29.26

↓ 24.06 36.56 ↑

Horní teplota

24.49

↓ 20.18 27.25 ↑

Spodní teplota

24.65

↓ 23.37 41.43 ↑

Napětí vstupní

15.16

↓ 15.10 15.19 ↑

Napětí baterie

14.08

↓ 14.05 14.11 ↑

### Tabulka dat

Čas	Vnitřní	Venkovní	Pláště	Horní	Dolní	Vstupní	Baterie	Status
30.4.24.00:00	28.00	21.18	29.06	24.93	25.00	15.16	14.11	OK
30.4.24.00:10	28.00	21.25	26.93	24.93	25.00	15.16	14.08	OK
30.4.24.00:20	28.00	21.18	28.43	24.93	25.00	15.16	14.08	OK
30.4.24.00:30	28.00	21.18	27.18	24.87	25.00	15.16	14.11	OK
30.4.24.00:40	28.00	21.12	28.81	24.87	25.00	15.16	14.11	OK
30.4.24.00:50	28.00	21.12	26.87	24.87	25.00	15.16	14.08	OK
30.4.24.01:00	27.93	21.12	27.68	24.87	25.00	15.13	14.08	OK
30.4.24.01:10	28.00	21.06	27.81	24.81	24.93	15.16	14.11	OK
30.4.24.01:20	28.00	21.06	27.87	24.81	24.93	15.16	14.08	OK
30.4.24.01:30	28.00	21.06	28.06	24.81	24.93	15.16	14.11	OK

Strana 1/88
Předchozí
Další

# Bibliography

1. *BABYBOX Problematika babyboxů* [online] [visited on 2024-05-05]. Available from: <https://www.babybox.cz/?p=problematika>.
2. *Statistika babyboxů ke dni 28. dubna 2024* [<https://www.babybox.cz/media/pdf/statistika-babyboxu.pdf>]. 2024. Accessed: 2024-5-5.
3. *Babybox - Zdeněk Jurřica - MONTEL* [<https://jurřica-montel.cz/babybox/>]. [N.d.]. Accessed: 2024-5-5.
4. ASAI, Atsushi; ISHIMOTO, Hiroko. Should we maintain baby hatches in our society? *BMC Med. Ethics*. 2013, vol. 14.
5. COCHRANE, Joan; MING, Goh Lee. Abandoned babies: the Malaysian ‘baby hatch’. *infant*. 2013, vol. 9.
6. OLEJARZ, S. Ethical concerns relating to child abandonment and baby hatches: The case of Poland. *J. Philos. Ethics Health Care Med*. 2017, vol. 11.
7. MONGIELLO, Marina; NOCERA, Francesco; PARCHITELLI, Angelo; RICCARDI, Luca; AVENA, Leonardo; PATRONO, Luigi; SERGI, Ilaria; RAMETTA, Piercosimo. A Microservices-based IoT Monitoring System to Improve the Safety in Public Building. In: *2018 3rd International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE, 2018, pp. 1–6.
8. AYDIN, Sahin; NAFIZ AYDIN, Mehmet. Design and implementation of a smart beehive and its monitoring system using microservices in the context of IoT and open data. *Comput. Electron. Agric*. 2022, vol. 196, p. 106897.
9. DE IASIO, Antonio; FURNO, Angelo; GOGLIA, Lorenzo; ZIMEO, Eugenio. A Microservices Platform for Monitoring and Analysis of IoT Traffic Data in Smart Cities. In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 5223–5232.

10. SHABANI, Isak; BIBA, Tonit; ÇIÇO, Betim. Design of a Cattle-Health-Monitoring System Using Microservices and IoT Devices. *Computers*. 2022, vol. 11, no. 5, p. 79.
11. RAHMATULLOH, Alam; SARI, Dewi Wulan; SHOFA, Rahmi Nur; DARMAWAN, Irfan. Microservices-based IoT Monitoring Application with a Domain-driven Design Approach. In: *2021 International Conference Advancement in Data Science, E-learning and Information Systems*. IEEE, 2021, pp. 1–8.
12. SIMANJUNTAK, Eko; SURANTHA, Nico. Multiple time series database on microservice architecture for IoT-based sleep monitoring system. *Journal of Big Data*. 2022, vol. 9, no. 1, p. 108.
13. *Grafana Labs*. Grafana documentation [<https://grafana.com/docs/grafana/latest/>]. [N.d.]. Accessed: 2024-5-5.
14. *InfluxData*. InfluxDB [<https://www.influxdata.com/>]. 2017. Accessed: 2024-5-5.
15. SCRIPTBEES. *Medium*. What is TICK Stack, and why should we consider using it? [<https://medium.com/@scriptbees/what-is-tick-stack-and-why-should-we-consider-using-it-5a2bddcd7b10>]. 2020. Accessed: 2024-5-5.
16. REESE, George. Distributed Application Architecture. In: *Database Programming with JDBC & Java, Second Edition*. 2000, pp. 126–145.
17. LEWIS, James; FOWLER, Martin. *martinfowler.com*. Microservices [<https://martinfowler.com/articles/microservices.html>]. [N.d.]. Accessed: 2024-5-5.
18. RICHARDSON, Chris. *Microservice Architecture* [<https://microservices.io/>]. [N.d.]. Accessed: 2024-5-5.
19. RICHARDSON, Chris. *Microservices Patterns: With examples in Java*. First Edition. Manning, 2018.
20. NEWMAN, Sam. *Building Microservices: Designing Fine-Grained Systems*. 2nd ed. O'Reilly Media, 2021.
21. *What Is a Message Broker?* [<https://www.ibm.com/topics/message-brokers>]. 2024. Accessed: 2024-5-5.
22. *RabbitMQ Documentation* [<https://www.rabbitmq.com/docs>]. [N.d.]. Accessed: 2024-5-5.
23. *Traefik Labs: Say Goodbye to Connectivity Chaos*. API Gateway: Developer's #1 Choice [<https://traefik.io/solutions/api-gateway/>]. [N.d.]. Accessed: 2024-5-5.
24. KONG. *What is an API Gateway? Benefits and Use Cases* [<https://konghq.com/blog/learning-center/what-is-an-api-gateway>]. [N.d.]. Accessed: 2024-5-5.

25. *What is a REST API?* [<https://www.ibm.com/topics/rest-apis>]. 2024. Accessed: 2024-5-5.
26. EDWARDS, Alex. *Let's Go Further!* [N.d.].
27. ERICKSON, Jeffrey. *Here's Why JSON Rules the Web* [<https://www.oracle.com/database/what-is-json/>]. Oracle, 2024. Accessed: 2024-5-5.
28. *Rendering* [<https://nextjs.org/docs/pages/building-your-application/rendering>]. [N.d.]. Accessed: 2024-5-5.
29. SHAD, Sherry. *Medium*. SSR vs CSR: Server Side Rendering vs Client Side Rendering [<https://medium.com/@shararehaddev/ssr-vs-csr-server-side-rendering-vs-client-side-rendering-deb1d3855b77>]. 2023. Accessed: 2024-5-5.
30. *MDN Web Docs*. Progressive web apps [[https://developer.mozilla.org/en-US/docs/Web/Progressive\\_web\\_apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)]. [N.d.]. Accessed: 2024-5-5.
31. *web.dev*. Progressive Web Apps [<https://web.dev/explore/progressive-web-apps>]. [N.d.]. Accessed: 2024-5-5.
32. *React Hooks for Data Fetching – SWR* [<https://swr.vercel.app/>]. [N.d.]. Accessed: 2024-5-5.
33. *Google Cloud*. What are containers? [<https://cloud.google.com/learn/what-are-containers>]. [N.d.]. Accessed: 2024-5-5.
34. *Docker*. What is a Container? [<https://www.docker.com/resources/what-container/>]. [N.d.]. Accessed: 2024-5-5.
35. *Docker Documentation*. Docker Compose overview [<https://docs.docker.com/compose/>]. 2024. Accessed: 2024-5-5.
36. *IBM Developer* [<https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1/>]. [N.d.]. Accessed: 2024-5-5.
37. *MongoDB*. What Is NoSQL? NoSQL Databases Explained [<https://www.mongodb.com/nosql-explained>]. [N.d.]. Accessed: 2024-5-5.
38. *What is MongoDB?* [<https://www.mongodb.com/docs/manual/>]. [N.d.]. Accessed: 2024-5-5.
39. *InfluxDB OSS v2 Documentation* [<https://docs.influxdata.com/influxdb/v2/>]. [N.d.]. Accessed: 2024-5-5.
40. EXTIO TECHNOLOGY. *Medium*. Understanding JSON Web Tokens (JWT): A Secure Approach to Web Authentication [<https://medium.com/@extio/understanding-json-web-tokens-jwt-a-secure-approach-to-web-authentication-f551e8d66deb>]. 2023. Accessed: 2024-5-5.
41. *JSON Web Tokens - jwt.io* [<https://jwt.io/>]. Auth0, [n.d.]. Accessed: 2024-5-5.



42. *Best practices for REST API design - Stack Overflow* [<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/>]. 2020. Accessed: 2024-5-5.
43. OSSPH. *10 Programming Principles Every Software Developer Should Know* [<https://blog.ossph.org/programming-principles-every-developer-should-know/>]. [N.d.]. Accessed: 2024-5-5.
44. RAWSON, David. *The Startup*. Idiomatic Code - The Startup - Medium [<https://medium.com/swlh/idiomatic-code-a73f17f0f287>]. 2020. Accessed: 2024-5-5.
45. BORKAR, Pradnya. *Medium*. Functional programming principles - Pradnya Borkar [<https://medium.com/@kumbhar.pradnya/functional-programming-principles-6f59bc6764ff>]. 2020. Accessed: 2024-5-5.
46. *Docs* [<https://nextjs.org/docs>]. [N.d.]. Accessed: 2024-5-5.
47. GREIF, Sacha. *State of JavaScript 2022* [<https://2022.stateofjs.com/en-US/>]. [N.d.]. Accessed: 2024-5-8.
48. VAEZIAN, Vahid. *Towards Data Science*. Popularity Ranking of Programming Languages [<https://towardsdatascience.com/popularity-ranking-of-programming-languages-72bcf697ea20>]. 2020. Accessed: 2024-5-5.
49. O'GRADY, Stephen. *tecosystems*. The RedMonk Programming Language Rankings: January 2024 [<https://redmonk.com/sogrady/2024/03/08/language-rankings-1-24/>]. 2024. Accessed: 2024-5-5.
50. *Bun*. What is Bun? [<https://bun.sh/docs>]. [N.d.]. Accessed: 2024-5-5.
51. CHAKRABORTY, Sutirtha. *Goroutines in Golang - Golang Docs* [<https://golangdocs.com/goroutines-in-golang>]. [N.d.]. Accessed: 2024-5-5.
52. *Web Frameworks Benchmark* [<https://web-frameworks-benchmark.netlify.app/>]. [N.d.]. Accessed: 2024-5-5.
53. *www.techempower.com*. TechEmpower Web Framework Performance Comparison [<https://www.techempower.com/benchmarks/>]. [N.d.]. Accessed: 2024-5-5.
54. *ApexCharts.js*. JavaScript Line Chart with Annotations – [<https://apexcharts.com/javascript-chart-demos/line-charts/line-chart-annotations/>]. ApexCharts, 2018. Accessed: 2024-5-5.
55. JUŘICA, Zbyněk. *timeseries-benchmark* [<https://github.com/zbyju/timeseries-benchmark>]. 2023.
56. *What is middleware?* [<https://www.redhat.com/en/topics/middleware/what-is-middleware>]. [N.d.]. Accessed: 2024-5-5.

57. DOBBELAERE, Philippe; ESMAILI, Kyumars Sheykh. Kafka versus RabbitMQ: A comparative study of two industry reference publish/-subscribe implementations: Industry Paper. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 227–238. DEBS '17.
58. *Caddy Documentation* [<https://caddyserver.com/docs/getting-started>]. [N.d.]. Accessed: 2024-5-8.
59. KLYNE, G; NEWMAN, C. *Date and Time on the Internet: Timestamps*. Tech. rep.
60. XIA, Vincent. *Medium*. What is Mobile First Design? Why It's Important & How To Make It? [<https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00>]. 2017. Accessed: 2024-5-5.
61. FOWLER, Martin. *martinfowler.com*. Test Driven Development [<https://martinfowler.com/bliki/TestDrivenDevelopment.html>]. [N.d.]. Accessed: 2024-5-9.

# Contents of the Attached Media

<code>src</code>	
├ <code>impl</code> .....	implementation source codes
├ <code>thesis</code> .....	source code for the thesis in $\text{\LaTeX}$ format
└ <code>text</code> .....	text of the thesis
├ <code>thesis.pdf</code> .....	thesis in the PDF format