POLITECNICO DI TORINO

MASTER's Degree in Software Engineering

MASTER's Degree Thesis

Transforming IoT Prototypes into Scalable and Maintainable Industrial Software Solutions

Supervisors

Prof. STEFANO QUER

Candidate

EDOARDO MICCONO

July 2024

Summary

The modern software development landscape offers multiple ways to write small code snippets in a really fast and efficient way, thanks to a variety of AI-powered tools that greatly simplify the creation of small to medium-sized projects. However, transitioning from a small proof-of-concept project to a scalable and maintainable industrial production code is a challenging endeavor.

The thesis analyzes the difference between these two phases, emphasizing the transformation from a code prototype, focused on core functionalities and feasibility of a project, into a robust, scalable, and maintainable market-ready software solution. This transformation process involves multiple crucial steps, that will greatly enhance the overall quality of the product thanks to CI/CD methodologies and best practices while also adding new features and functionalities.

Extensive testing will be implemented ensuring the software works as intended and meets the requirements, also identifying errors that may go unnoticed otherwise. The test suite is implemented using Jest and Supertest, leveraging new technologies such as Test containers instead of relying on mocking services and databases. The suite will then be integrated into a pipeline, further improving the quality of the code merged into the main repository. This will be paired with precise documentation, crucial for maintenance, general system's understanding and future development of the application.

An Object relational mapping (ORM) library and a validation library will be introduced (Drizzle ORM and ZOD in this case) with the purpose of further improving the software reliability, security and scalability.

Additionally, this thesis leverages this new infrastructure, making the integration of Quality of Life (QoL) and management features easier to test and implement, refactoring huge parts of the existing code, and enhancing the overall application.

Through examination of methodologies, tools and best practices, this thesis provides a guide transitioning from prototype to production code, demonstrating the effectiveness of these practices directly with the implementation of new useful features.

Table of Contents

List of Tables

List of Figures

Acronyms

DB

Data Base

API

Application Programming Interface

ORM

Object Relational Mapper

CI/CD

Continuous Integration / Continuous Development

I/O

Input Output

HTTP

HyperText Transfer Protocol

UI

User Interface

UX

User eXperience

SQL

Structured Query Language

IoT	Internet of Things
IIoT	Industrial Internet of Things
QoL	Quality of Life
AI	Artificial Intelligence
IDE	Integrated Development Environment
ECMA	European Computer Manufacturers Association
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model
RDBMS	Relational Database System
AWS	Amazon Web Services
JSON	JavaScript Object Notation

JWT

JSON Web Token

DevOps

Development and Operations

YAML

Yet Another Markup Language

VM

Virtual Machine

FE

FrontEnd

BE

BackEnd

PK

Primary Key

FK

Foreign Key

Chapter 1

Introduction

The Internet of Things (IoT) and Artificial Intelligence (AI) have been rapidly evolving in recent years, transforming various industries and significantly impacting daily life.

IoT refers to an interconnected network of physical devices, that collects and exchange data, while AI involves systems capable of performing tasks that typically require human intelligence.

The proliferation and growth of IoT has been remarkable in the recent years, and it has influenced the whole world, from normal people living their daily life, with "smart" accessories such as light bulbs or thermostats, to companies that leverage sensors and data to optimize their industrial processes.

At the same time AI had huge breakthroughs, in many of its fields, particularly in machine learning, deep learning and natural language processing. These advancements made advanced algorithms, capable of analyzing large datasets, recognizing patterns, making high accuracy predictions and even program in every programming language with high proficiency.

The high amount of data provided by IoT devices, can now be analyzed by AI to improve products and processes. This synergy will eventually lead to smarter systems, capable of predictive maintenance and autonomous decision making.

In the modern era of software development, thanks to AI powered tools such as ChatGPT, CodeCopilot and others, is really easy to create small to medium sized projects. Nowadays any reasonably motivated individual could create some small applications able to perform simple tasks.

As many may see this as the end of the developer's job, this thesis will deeply analyze the differences between a small sized project and a professional, sellable software product.

These small medium projects are usually really effective to test functionalities, ideas or proof of concepts, and they are critical to understand if a software product may work or not, but transitioning a project from a nascent code prototype to a robust and scalable industrial production code base is an important and challenging endeavor. This transition is crucial for properly transforming new innovative ideas and solutions into reliable and maintainable code, able to withstand the real-world applications.

A code prototype typically implements all the core functionalities and demonstrates the feasibility of a concept, and it is always focused on proving the viability of ideas rather than adhering to best practices. This approach is optimal for exploring and validating new concepts but it lacks documentation and robustness, required for industrial production.

Industrial production code on the other hand demands a more careful and comprehensive approach, that takes into account multiple factors obviously left out when prototyping. It is required to adhere to some code standards, to take into account the scale (and the potential scalability) of a project, as well as security and maintainability. This transition requires a meticulous approach, involving code refactoring, extensive testing, documentation, optimization and adherence to best coding practices and standards.

This thesis will delve into this process of transforming a project from its prototype phase into a large-scale production code, not only from a purely technical standpoint, but also implementing into the existing app useful management functionality and Quality of Life (QoL) features, to leverage the infrastructure created that made this addictions easier to test and implement.

We will be exploring critical steps, methodologies and tools used to make this transformation possible.

Chapter 2

Presentation: IoT, Architecture and Context

This chapter explores the impact of IoT and Industry 4.0 , and introduces AROL IoT architecture. We examine principles, implementation and benefits of this architecture, showcasing its role in the modern industry, and presenting some core concepts that will be useful to understand the future chapters.

2.1 IoT

IoT represents a paradigm shift in how devices and systems connect and interact with each others. Embedding sensors and softwares into traditionally analogical everyday objects, we obtain a huge network of interconnected devices that can communicate seamlessly.

The IoT concept dates back to 1980, where a university equipped a Coke machine with sensors to report its inventory and check if they were cold, however it isn't until the early-mid 2000s that IoT began to look like a new system that can shape the future, where technological innovation in wireless connectivity, sensors and data analytics made this concept more real and feasible.

IoT encompasses a broad variety of devices, from wearable technology to household appliances, but they all share 4 key components:

1. Devices/Sensors - literally the "Things" in IoT, equipped with sensors to perform some kind of checks.

2. Connectivity - every device needs some kind of connectivity unit able to transmit data to other devices or central systems.
3. Data Processing - the data collected by the devices must be properly processed, this can happen directly on the device (edge computing) or directly at a central server (cloud computing).
4. User Interface - everyday users must be able to interact with the IoT system through mobile apps or web applications, to check and receive notification about the status of their devices.

As example any IoT device could be programmed to improve performance or energy consumption depending on the situation and with very little computing effort, while also providing real time feedback to the user.

The primary aim of IoT is to enhance every aspect of everyday life, improving convenience, efficiency and decision making. Its versatility allows it to be applied to numerous different use cases: from smart homes, to healthcare, to agriculture or smart cities.

The IoT term is really wide and it encompasses so many sectors that we must narrow our focus, to the one we are more interested in: Industrial Internet of Things (IIoT).

2.1.1 IIoT

While IoT focuses on consumer application, IIoT (Industrial Internet Of Things) targets the industrial sector, deploying smart sensors on industrial machines, allowing data gathering and analysis, improving efficiency, productivity and safety.

The general idea is basically the same as IoT, but given the industrial impact on economy, production ad safety, The stakes in the industrial field are even higher.

Storing and analyzing this huge amount of data allows to extend the machineries lifespan by predicting potential failures and doing the needed maintenance before accidents happen. Also processes like supply chain management are enhanced through real time tracking of goods, which makes the whole process simpler, not to mention the fact that having all of these sensors on (usually) very expensive machines, allows the company to take data-driven decision, minimizing the wastes and maximizing production and performances.

All of these upsides granted the IIoT the name of Industry 4.0, representing a substantial shift in how the industries operate, driven by technological innovations.

Integrating IoT, cloud computing and AI to create smart factories and intelligent production systems, leads the world into unprecedented levels of efficiency and innovation.

The number 4.0 derives from the fact that historically there has been three industrial revolutions, and that's why it is also usually referred as the Fourth Industrial Revolution:

1. First Industrial Revolution - late 18th century, brings steam power into industry production.
2. Second Industrial Revolution - early 20th century, brings mass production and electricity into assembly lines.
3. Third Industrial Revolution - mid 20th century, brings automation, computers and electronics into industries.

While it brings to the table lots of benefits, it also raises some concerns, since it changes not only the way people interact with machines but also the whole workforce of industries has to be redistributed in the newest areas like AI, data analytics, robotics and so on. Also the great number of devices connected one another may raise concerns due security and interoperability, since integrating two different IIoT systems together may be quite complex, and it needs establishing of some common standards. Not to mention the fact that mid to small industries may find hard to perform this transition since it requires a huge investment in infrastructure, technology and training.

Even knowing some problems that may arise, the future of Industry 4.0 looks promising, and the constant digital innovations and advancements are bound to improve and enhance this sector even more.

2.2 AROL Group

AROL Group includes 4 different companies : AROL Closure System, MACA, Tirelli and UNIMAC-GHERRI. Thanks to 40+ years of experience, AROL Closure System is a global point of reference and world's leading manufacturer for the design, production and distribution of capping, corking, crowning, plugging, and closing equipment.

AROL's on-site vast manufacturing capability and lean commitment ensure flexibility, the highest quality standards, and reliable delivery times. It manufactures over

700 machines a year and to date, AROL GROUP has installed more than 30.000 machines worldwide [1] .

2.3 AROL Layered modular architecture

We now need to take a step back and discuss the 4 key components of the IoT structure.

IIoT systems are usually designed following the Layered Modular Architecture, where each layer represent a different technological stack.

AROL, being an advanced and successful industry, already adopts an IIoT layered modular architecture, and we need to discuss it in detail.

AROL Layered modular architecture	
Device or Perception layer	Capping machineries and their built-in sensors
Network or Transport layer	Communication and network infrastructure like : Modbus TCP/IP protocol, PackML, Bluetooth
Service or Processing layer	Software in the cloud that collects and processes data arriving from the network layer
Content or Application Layer	People interface devices, like computer screens or tablets

Table 2.1: Summary of AROL's layered architecture

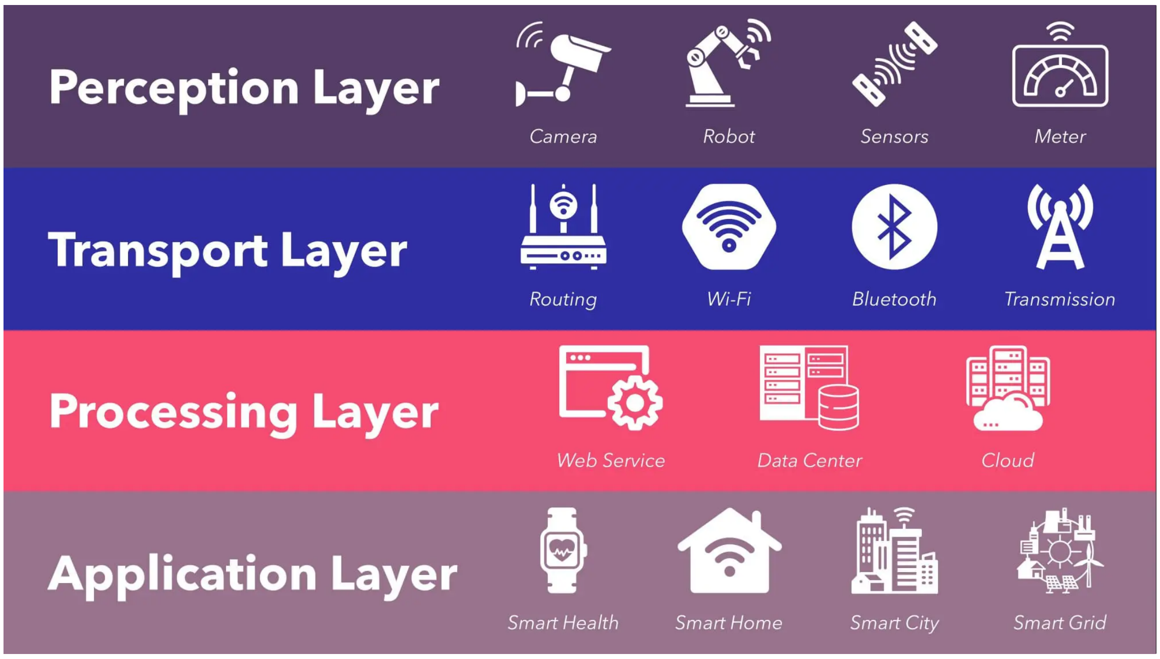
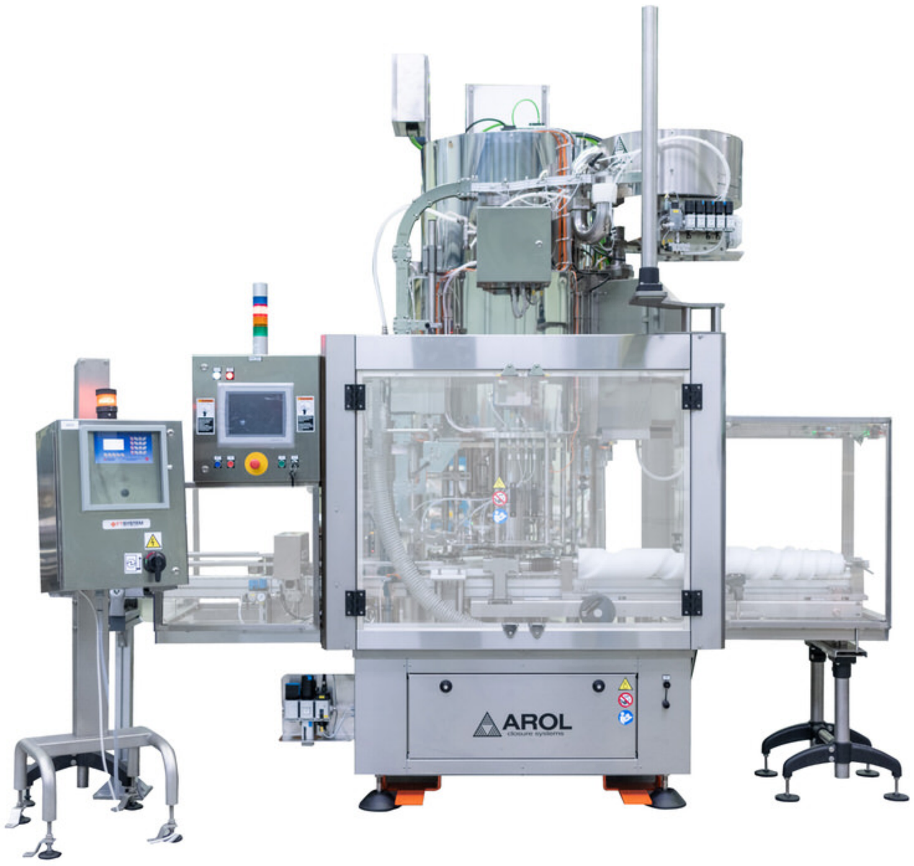


Figure 2.1: Image IIoT architecture [2];

2.3.1 Device or Perception layer

The Perception layer is represented by the machinery itself, in our case the capping machinery. Capping machines are huge industrial equipment used to apply a secure closure to some kind of container. The containers may vary in size and purposes, ranging from any kind of bottles to jars. Obviously different kinds of containers will have different types of capping machines.

Capping machines are widely used in the packaging industry to secure the contents of a container and maintain the quality and freshness of the product, protecting it from external contamination. Some machines are designed for high speed, high volume of operations, while others are designed for lower volume or more specialized applications. All the sensors are installed directly onto the machine to provide trustable real time data.



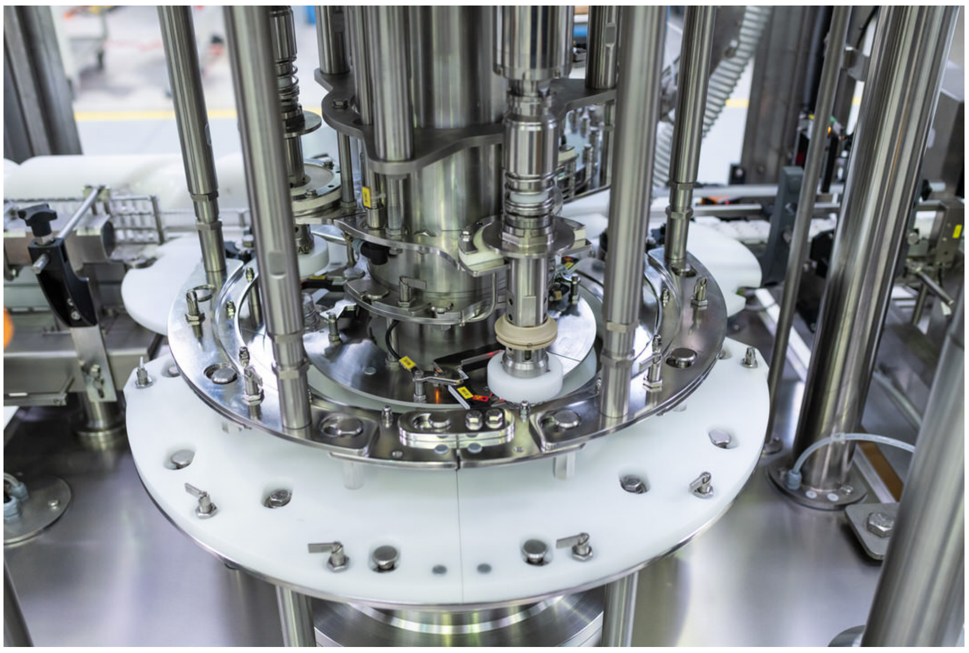


Figure 2.2: Images of AROL Equatorque capping machine [3]

2.3.2 Network or Transport layer

The AROL capping machines are equipped with different kinds of sensors, mounted in different positions. Sensors can be in the central body of the machine or inside the capping heads (the piece of the machine that will actually perform the closure of the container). Depending on the position and the need of every sensor, they may have different kinds of communications, body mounted sensors are directly connected to the machine's PLC through Modbus TCP/IP protocol (standard IEEE Ethernet 802.3) while the head mounted sensors may also be polled via Bluetooth(or the same Modbus protocol).

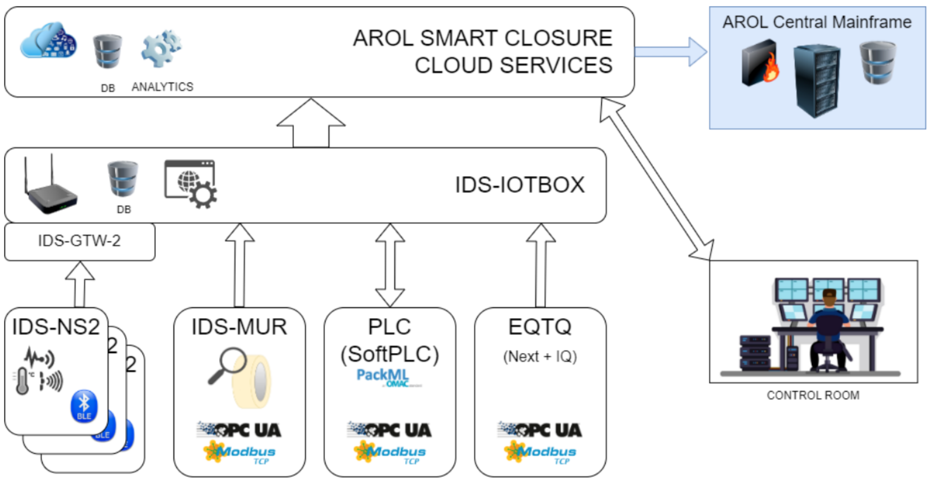


Figure 2.3: AROL current IIoT architecture and Infrastructure

AROL in recent years have also expanded and modernized its technology, to better adapt to the rapid changes of Industry 4.0. Combining edge computing devices and Node-RED scripts, sensor data processing is brought closer to the device where data is actually generated, reducing the amount of data sent through the network and limiting the amount of data processing that needs to be done later on. These edge computing devices have processing and short term storage capabilities, limiting the amount of data that needs to be stored in the cloud server since the quantity of data stored in the cloud is directly proportional with the cloud provider's bills. The data ingestion in the cloud is supported by protocols like MQTT (Message Queuing Telemetry Transport), a lightweight publish-subscribe network protocol that grants low bandwidth usage and power consumption.

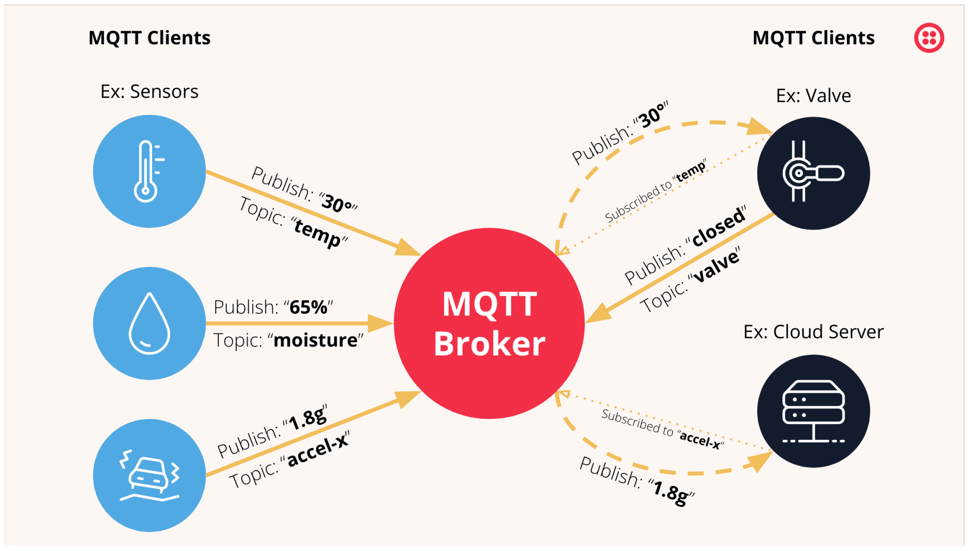


Figure 2.4: Example of MQTT protocol [4]

2.3.3 Service or Processing layer

This layer represents the software run in the cloud to analyze and transform the devices data into valuable and insightful information. This is where AI can be applied to the data to obtain some useful results, such as preemptive maintenance. The project object of this thesis just presents the information processed and analyzed by this layer.

2.3.4 Content or Application layer

The existing web application presents the data on this layer, visualizing it on a computer screens, miles away from where the machine is actually located. This layer also comprehends the built-in screen of capping machines, that presents the data in real time as it is sent into the cloud.

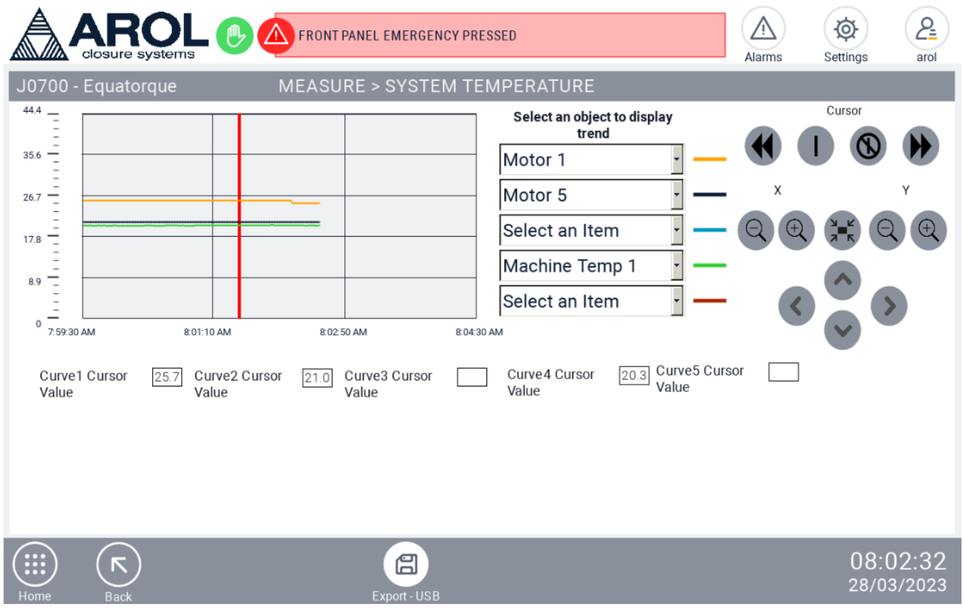


Figure 2.5: Images of built-in software

2.4 Objective of the thesis

This thesis aims to develop and expand an application designed to interact with IIoT devices. We will start from the excellent initial project, developed by Mario Deda, and we will delve into a new journey, understanding the work, adapting it to new technological standards, adding more functionalities, making it more scalable and reliable through the implementation of test suites, pipelines, linters and CI/CD best practices.

Chapter 3

Tech Stack

In this section we will briefly discuss the local development environment, as well as the principal tools used and the existing web application, which is the starting point for all of our future discussions and work. Designed to allow AROL company to provide software support for their products, the application aims to be an easy and intuitive way to monitor productivity and efficiency of the capping machines sold, leveraging this IIoT architecture opportunity.

This chapter will focus on the initial state of the application, which was also our primary constraint. Understanding the starting conditions is the first step to appreciate the challenges faced and the decisions made to overcome them. This insight will provide a comprehensive foundation for understanding the evolution and growth of this project.

3.1 Objectives of the app

The app has the primary objective of being able to scan, fuse, process and visualize in a human-friendly manner all the machinery data stored in the cloud. Users must be able to visualize all this data through a completely customizable and dynamic dashboard solution, as well as store data and documents for every machinery they possess. This way the app will be the central point to monitor, manage and check all machines sold by AROL. Of course most of the data managed by the app is sensible or reserved, so adequate authentication methods and role based access must be implemented accordingly.

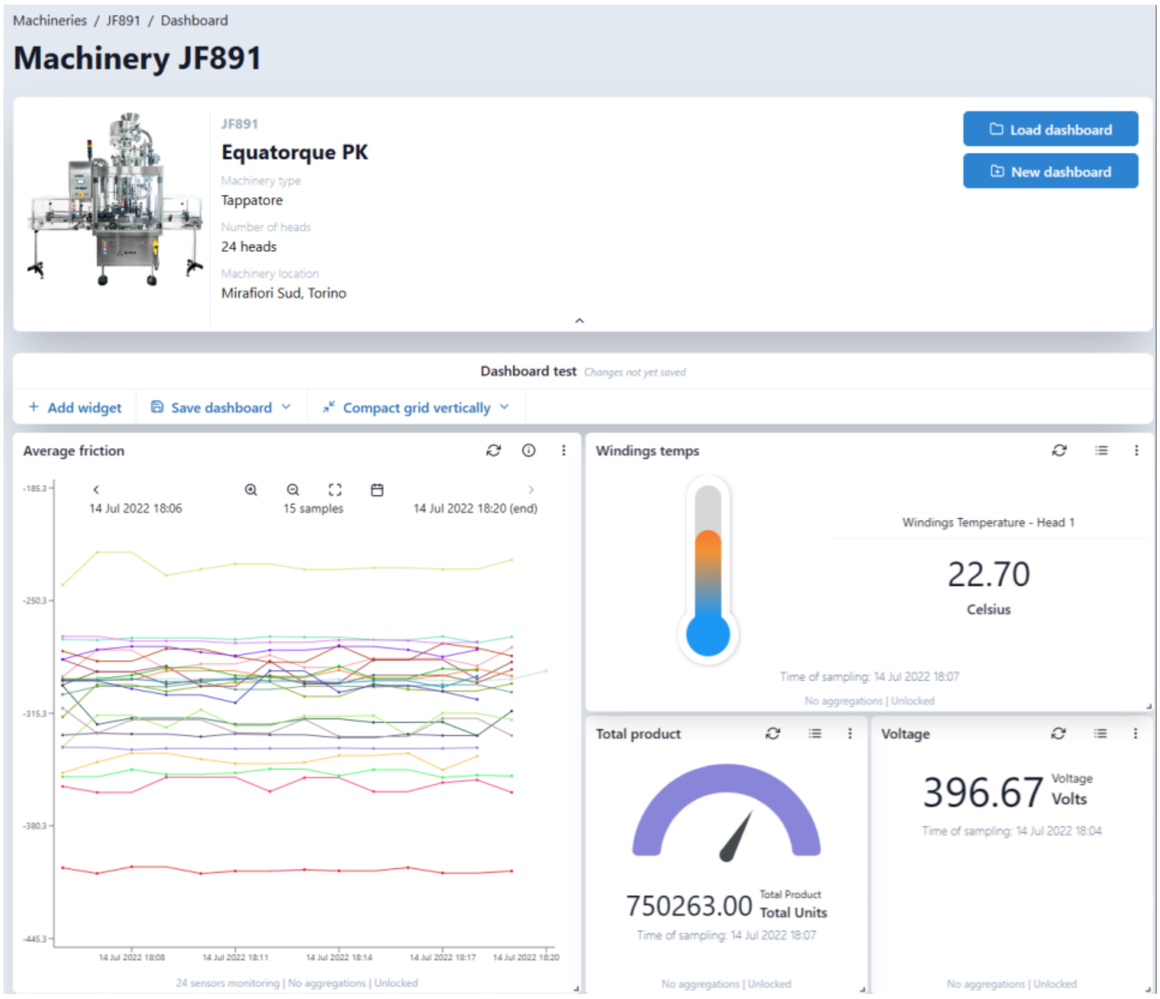


Figure 3.1: Dynamic dashboard solution

We will now talk about the initial Tech Stack and its Architecture, and all the following chapters will be dedicated to how this evolved to allow the growth of functionality and reliability of the app.

3.2 Local development environment and Tools

Integrated Development Environments (IDEs) are used daily by millions of developers and programmers all over the world, and VS Code and Intellij IDEA are two of the most popular ones. Each one of them has a huge numbers of supporters due to its uniqueness, since they represent two sides of the same coin. VS Code it's the open source side, being free and completely customizable, Intellij IDEA is paid software, that offers hundreds of excellent tools out of the box, that most of the users will never even appreciate properly since no one really needs all of them.

3.2.1 VS Code

VS Code stands for Visual Studio Code, an IDE developed by Microsoft, which is a lightweight and open source editor that has become the favourite of many developers.

Being lightweight also means that it's really fast to install and setup, it only requires a few clicks and offers a limited set of functionalities. The idea here is that every developer can choose from the huge marketplace of extension to customize his workflow only with the specific tools that he needs, without being forced to download and install a whole package of features that will probably never be used.

One important extension of VS Code that needs to be mentioned is Copilot, an AI-powered code completion tool developed by GitHub in collaboration with OpenAI. Copilot assist developers as they write code, it understands the context and the pattern of the code, suggesting whole lines or blocks of code, greatly accelerating the development process, enhancing productivity and code quality.

This with many other useful plugins and functionalities like

the integrated terminal, the really well done Version Control integration with Git and the Live Share features that allows distant team members to work together remotely, made VS Code a really simple, free, yet powerful product, with almost no limits to its capabilities [5].

3.2.2 Intellij IDEA

Intellij IDEA on the other hand, developed by JetBrains, is a paid premium software, it's a fully fledged full featured IDE with all the functionalities that any developer may ever need. Primarily aimed at Java development, it also supports a wide array of languages, each equipped with intelligent coding assistance and deep integration with developers tools. It offers a large sets of features, but some of the most notable ones are:

1. Intelligent code completion - Intellij can refactor your code in a simple click and perform real-time code analysis

2. Database tools - really handy tools for database management such as SQL databases, allowing to interact with them directly within the IDE.
3. Framework support - extensive support for a wide variety of frameworks, making it particularly indicated for almost any kind of environment.
4. Version Control Integration - excellent version control support, allows to take a quick look at the state of the repository you are working in and to perform operations in a few clicks that may be hard to do manually such as merge or rebase.

These feature singularly looks like they aren't a big deal, but all together they make navigating the codebase and interacting with database or version control system a seamless activity, really fast and efficient once you learn how to use it. This rich feature set and its advanced tools makes IntelliJ IDEA a preferred choice for professional use and enterprise environments [6].

3.3 Programming languages

JavaScript is the clear leader of front-end (FE) development and it has been the go to choice for many years. Its versatility and ease of use made it a staple in web development. However application are grown in complexity and often a simple crash of the app (for whatever reason) may translate in a huge loss of money for the majority of companies, making the limitation of JavaScript much more apparent.

3.3.1 JavaScript

JavaScript is an interpreted programming language created in 1995 by Brendan Eich in just 10 days. It was named "Java"script just for marketing purposes to capitalize on the Java popularity at that time, but it has nothing in common with it. It was intended to enable web designers and part time programmers to create dynamic web pages, but its simplicity and cross-platform capabilities made JavaScript much bigger and largely used than what its creator had ever envisioned.

JavaScript was submitted to ECMA (European Computer Manufacturers Association) for standardization, and it was updated over the years, continuously evolving and improving.

JavaScript is integrated with HTML (HyperText Markup Language) and CSS (Cascading Style Sheets), and it usually interacts with the DOM (Document Model

Object) of a webpage. The DOM can be visualized as a tree of HTML objects, managed in fact through JavaScript, which has the capabilities of adding, removing and modify elements of the DOM, as well as handling events like clicks and form submissions or fetch data through HTTPs (HyperText Transfer Protocol) requests.

In 2009 Ryan Dahl created Node.js, a runtime environment that allows JavaScript to run on the server-side, significantly expanding its capabilities beyond the browser. Since the creation of Node.js, JavaScript landed in the server side of application, offering an event-driven, non-blocking I/O (Input/Output), and being perfectly capable to handle HTTP requests and responses, database or file system interactions [7].

3.3.2 Typescript

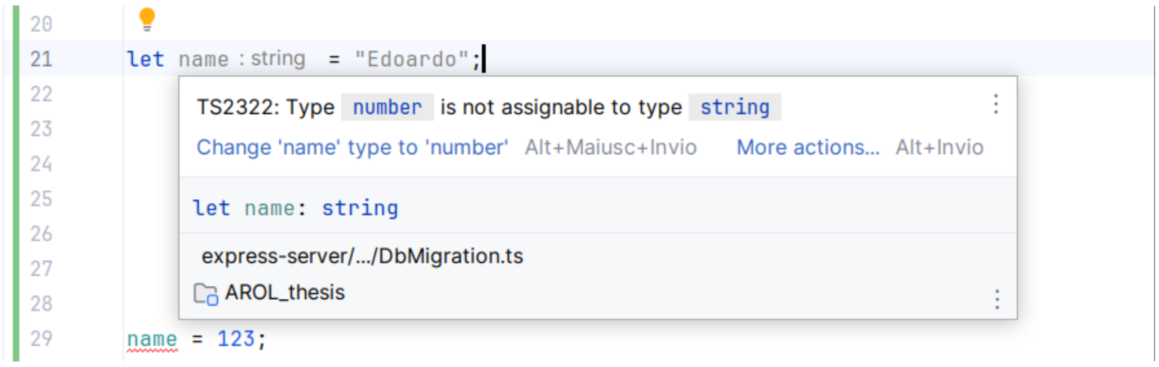
As JavaScript became more spread worldwide, its limits became much more apparent. JavaScript was created to carry on simple scripting tasks in browsers, but overtime it evolved so much that is used both in client and server side. The simplicity that made him so popular started to become a problem in large codebases.

To address these issues, Microsoft released Typescript in 2012, positioning it as a tool for building large-scale JavaScript application with robust and safe code. Typescript is a superset of JavaScript that introduces static typing, type inference and enhanced IDE support.

Being a super-set means that any JavaScript code will be valid in Typescript, allowing developers to gradually migrate from the former to the latter without completely rewriting their code or libraries. Also, being a superset means that any Typescript code is effectively compiled to JavaScript code before being executed, since it only adds features like type safety, to avoid many of JavaScript runtime problems.

Static typing is probably the most important Typescript feature. Typescript allows developers to define types for variables, parameters and return values, greatly helping catch type related errors at compile time rather than at runtime, reducing the likelihood of bugs in production and making the code more robust and maintainable.

Type inference is also really important, since it allows to keep the repetitive type annotations at minimum and the code short and clean, while still providing type safety.



```
let name : string  = "Edoardo";
```

TS2322: Type `number` is not assignable to type `string`

Change 'name' type to 'number' Alt+Maiusc+Invio
More actions... Alt+Invio

```
let name: string
```

express-server/.../DbMigration.ts

AROL_thesis

```
name = 123;
```

Figure 3.2: Typescript infers the 'string' type even if it's not declared and the IDE throws an error

Explicit type declaration means that the code is more readable and self-documenting, also allowing the IDE of your choice to do real time error checking [8].

For a more in depth comparison between JavaScript and Typescript check the table below.

Feature	**JavaScript**	**Typescript**
Typing	Dynamically typed	Statically typed
Compilation	Interpreted	Compiled to JavaScript
Type Safety	No type safety	Provides type safety
Learning Curve	Easier for beginners	Steeper due to type system
Tooling Support	Limited to JavaScript tools	Excellent tooling support with modern IDEs
Code Scalability	Less suitable for large codebases	Better suited for large-scale applications
Syntax	ES5/ES6 and later versions	Superset of JavaScript (ES5/ES6 and later versions)
Tooling	Basic tools (linters debuggers)	Advanced tooling and IDE support
Error Detection	Errors found at runtime	Errors are caught at compile-time
Backward Compatibility	Fully compatible with all browsers	Requires transpilation to JavaScript
Development Speed	Faster prototyping	Slower due to type checking
Code maintainability	Harder to maintain large codebases	Easier to maintain codebases due to type checking
Popular Use Cases	Web development, server-side scripting, game development	Enterprise-level applications, large-scale JavaScript projects

Table 3.1: Comparison of JavaScript and Typescript

3.4 Application's Architecture

Here we will discuss the initial architecture, laying the basis for all future discussions and modifications. The architecture is subdivided in three layers: client, server and data tier. The whole client and server tier are programmed in Typescript. We will now proceed to analyze every one of them.

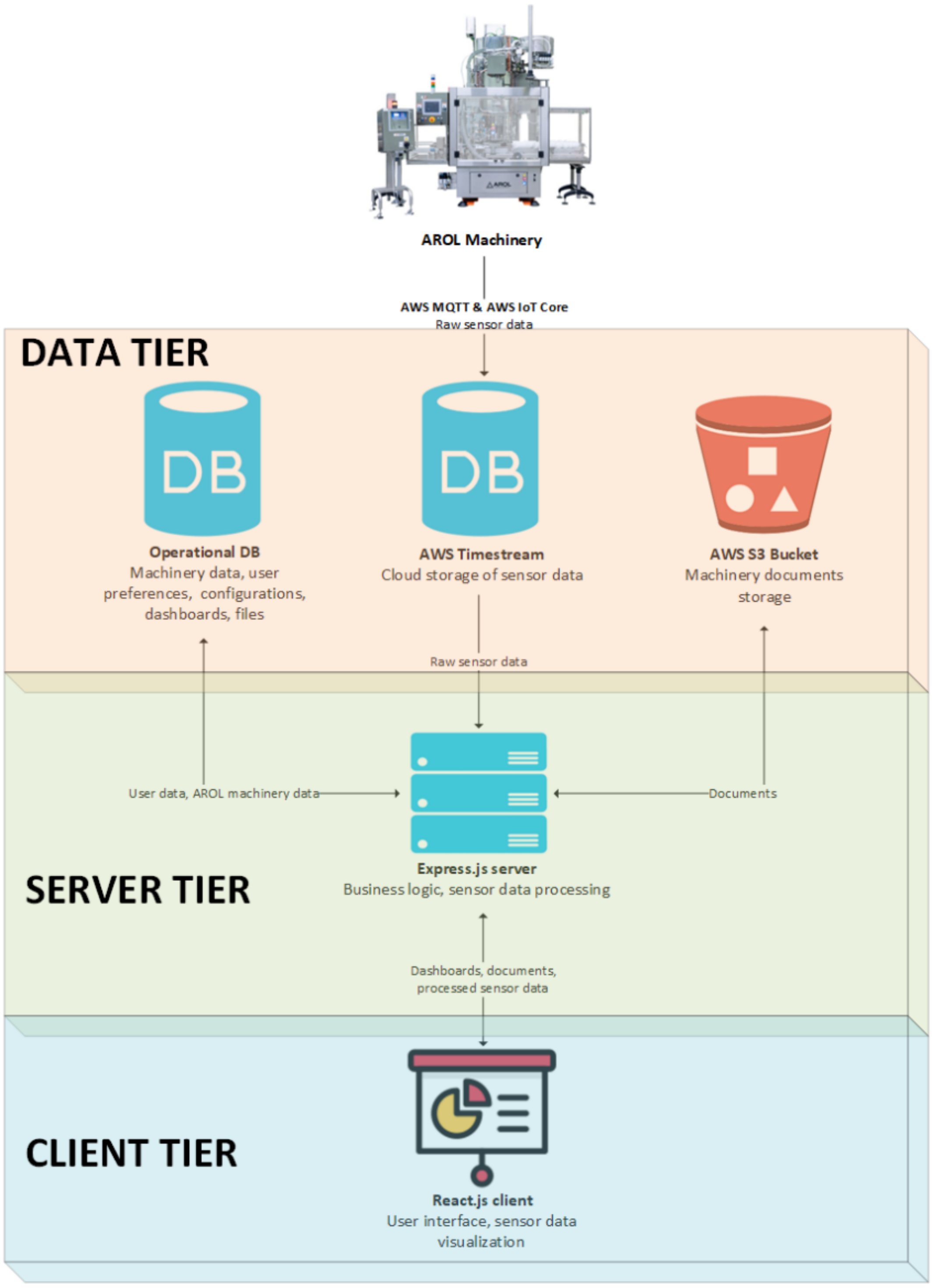

AROL Machinery
AWS MQTT & AWS IoT Core
Raw sensor data
DATA TIER
DB
Operational DB
Machinery data, user preferences, configurations, dashboards, files
DB
AWS Timestream
Cloud storage of sensor data
AWS S3 Bucket
Machinery documents storage
Raw sensor data
User data, AROL machinery data
Documents
SERVER TIER
Express.js server
Business logic, sensor data processing
Dashboards, documents, processed sensor data
CLIENT TIER
React.js client
User interface, sensor data visualization

Figure 3.3: Images of Client Server and Data tier

3.5 Client tier

The client tier of the app is developed in React.js which is a Javascript framework responsible of the whole user interface and user experience. This layer runs in the user browser and interacts with the server side of the application through API calls.

This tier is a presentation of the data analyzed and processed by the server side, but at the same time it's a sophisticated and highly responsive part of the overall architecture, which has to call the APIs properly.

User interface and sensor data visualization (achieved through dashboards and widgets) are presented here. Programmed in Typescript, the purpose of this layer is to provide clear info and friendly user interaction.

3.5.1 React.js

React.js is an open-source JavaScript library for building UI components and interfaces. It was created by Jordan Walke, a software engineer at Facebook in 2011, that wanted to offer a solution to the big challenge that its company was facing at the time: offering a rich and flawless UX, building a dynamic and responsive UI without giving up on performance.

React was later in 2013 open sourced and released to the public under MIT license.

The resulting features of React are specifically designed to solve this issue, starting from the component-based architecture, that encourages the realization of reusable UI components which can be nested, managed and handled independently.

Another key feature in React is the virtual DOM (Document Object Model) since it keeps track of the state of components and it handles the changes updating only the part of the UI that changed rather than re-rendering the entire page. Instead of directly manipulating the browser's DOM, React creates a new virtual DOM, compares it with the previous version, and only updates the necessary parts of the real DOM, reducing useless rendering, minimizing expensive DOM operations and enhancing performance.

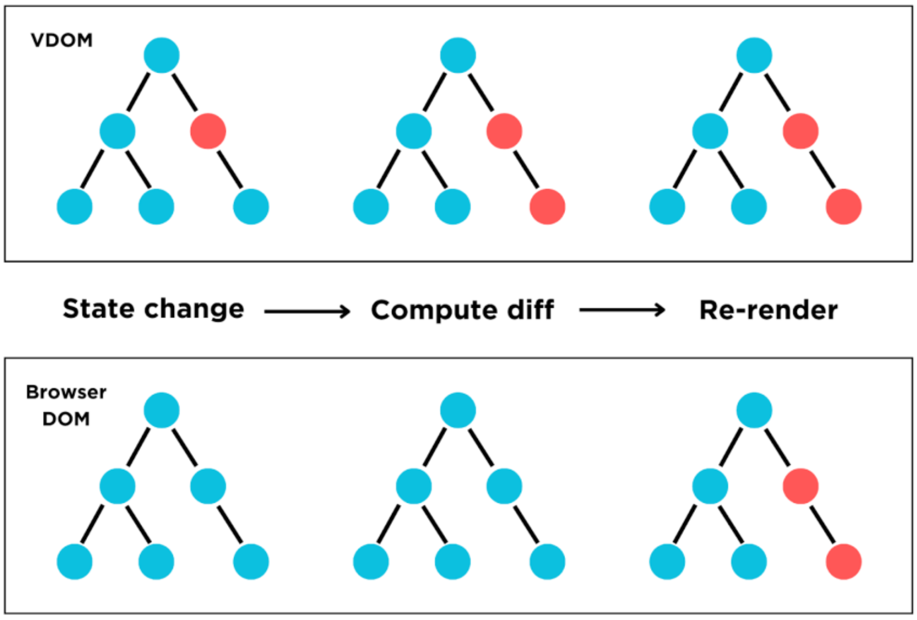


Figure 3.4: React Virtual DOM and browser DOM comparison [9]

Another advantage of React is the JSX (TSX in our case since we are using Typescript) syntax extension, which allows developers to write HTML-like elements, which can then be controlled through JavaScript code, making the structure management of the page intuitive and convenient.

One last cornerstone of React is the unidirectional data flow, that may look like a limitation initially, but having data flow in a single direction through the whole application makes it more predictable and easy to understand, especially as it grows in complexity.

React has 4 principal core concepts that need to be understood and respected to proficiently work with it:

1. Components - subdivided in functional and class components. Functional components are stateless and take props as inputs and return JSX.
2. Props - short for proprieties, are read-only inputs passed only from a parent component to a child component, enforcing this way the unidirectional data flow.
3. State - a variable inside a component, with dynamic data. States are mutable

and they change over time due to user interactions or network responses. State is used to control a component's rendering and behavior.

4. Life-cycle methods - methods bound to execute code at specific points during a component's life-cycle.

Nowadays React.js is really popular and diffused, well supported and with good documentation, making it particularly well-suited for creating dynamic and responsive single-page (or multiple-pages) applications.

Additionally, since its success and popularity, React offers a large and active community with a large selection of libraries and tools, that are generally pretty handy and easy to pick up.

The whole React architecture is designed to offer extreme control to the programmer offering him ways to control components rendering, appearance (through CSS), life-cycle, how data is passed between them, states of variables, routing and much more [10].

3.6 Server tier

At this layer, the technologies used to implement it are specified. We will use dedicated frameworks and libraries, containing ready-made code that helps the developer to prevent security issues and reduce the potential of critical bugs.

3.6.1 Node.js

Node.js an open source cross-platform runtime environment that executes JavaScript on the server side, effectively outside the browser, opening new possibilities and broadening developers limits, since JavaScript was initially developed to be run in browsers.

It is efficient, lightweight, and uses and event-driven, non-blocking, single threaded I/O model; making it optimal for real time services that need to handle a large number of concurrent connections.

Node.js operates in a single-threaded event loop, handling concurrent operations using promises, callbacks and async/await calls, so that operations like reading from a database or file system can be performed asynchronously, leaving the main thread free to perform other tasks.

Node.js also includes the **npm** package manager, which makes easier to manage and install libraries and modules in any Node applications. Its abundance of modules (it's the world's largest software registry) and well maintained plugins makes it a great choice for small and medium sized projects.

Some of the most important Node.js features are:

1. Middlewares - functions able to access the request(req) and response(res) objects. Used to perform logging and authentication
2. Routing - allows developers to define routes for different HTTP methods and URL paths, organizing application logic
3. Database integration - all the most widely used databases, from MongoDB to PostgreSQL, offers libraries able to help managing database operations.
4. Testing - Node.js application can be tested using frameworks like Mocha or Jest, providing great support for unit and integration testing.

Node.js has the merit of transforming the server side of development bringing JavaScript to the back-end. Its architecture is high performance and highly scalable, with a great offer of libraries and plugins, making it one of the most popular choices on the market, given that either you are building a simple web server or a complex enterprise application, Node.js has the tools to suit your needs [11].

3.6.2 Express.js - a Node.js framework

Express.js is a minimalistic and flexible Node.js web application framework, easy to use and well supported. The very same existence of Express is one of the reason of Node.js success and vice versa.

Express is open source and can claim a large community of contributors. This makes it a popular choice among developers for creating server side applications, since its built in functions and middle-wares, make the management of the server side a breeze.

Express.js also provides an efficient routing mechanism that allows developers to map specific URLs to different function or controllers (that handles requests with different HTTP methods such as GET, POST, PUT, DELETE), as well as a mechanism that specifies pieces of code that must be executed before or after certain API calls, allowing for a precise execution flow management.

Express.js offers libraries used to augment existing features, a few of the most relevant ones are the following:

1. Multer - middleware used to handle multipart/form-data and uploading files.
2. PG-Promise - a PostgreSQL interface for Node.
3. AWS SDK for JS - a set of APIs useful to easily interact with AWS services.

To recap, Express.js is built on top of Node.js, it's simple and minimal, it does not come with a lot of built-in functionalities that would force the structure or the architectural pattern of the project, allowing each project to adopt the strategy that suits it more, making it versatile and used for a wide range of application, from web servers to microservices to real time IoT applications [12].

3.7 Data tier

All the complexity is on condensed on the server tier, since the data tier has only to provide the data when requested. This tier is divided in 3 different parts with 3 different purposes: a PostgreSQL DB as our operational DB, an AWS Timestream DB, an AWS3 DB.

3.7.1 PostgreSQL - operational DB

This is our operational db, where most of the data created and needed by the application is stored. As example Here we have all of our user's and company's data.

PostgreSQL (or simply Postgres) is an open-source RDBMS (Relational Database Management System), known for its robustness and reliability. Born in 1989 by Michael Stonebraker as POSTGRES project, it was later in 1996 transformed into PostgreSQL, to highlight its SQL capabilities, and released under that same name.

Given its long history, and the fact that is a system still used and still standing, Postgres has evolved over decades into a leading database system used worldwide, with a wide and rich feature set :

1. ACID compliant - Atomicity, Consistency, Isolation and Durability are 4 key properties of a database, that together ensure data integrity.

2. MVCC (Multi-Version Concurrency Control) - allows multiple transaction to occur concurrently without blocking the database, greatly improving performance

3. Extensible data types - beyond standard types like integers and strings Postgres supports JSON, XML and key-value pairs, also letting the user define their own data types.

4. Replication and Partitioning - both synchronous and asynchronous replication is supported, as well as table partitioning to improve performances on large tables.

PostgreSQL follows a client-server model; the server, known as "postgres" manages the database files, accepts connections from the client and performs operations for them. This allows opportunity for performance and query optimization to be done directly by the server, leveraging shared buffers, that reduces the needs of disk I/O, WAL (Write-Ahead Logging) that ensures data integrity logging all changes before applying them.

All of these features, perks and options make PostgreSQL viable for a large variety of use cases, that go from web application to geospatial applications or data warehousing. Its wide range of applications, combined with its open-source nature, its strong and vibrant community and the continuous innovation, PostgreSQL offers robustness and flexibility, making it one of the most beloved RDBMS [13].

3.7.2 AWS Timestream

AWS (Amazon Web Services) Timestream is where the Amazon cloud is used to store sensor data. Exactly for this reason it is a non traditional database, especially designed to handle big data in input and store it in a proper way.

Timestream is a a fully managed time series database service offered by AWS (Amazon Web Services), designed to handle the storage, the ingestion and the querying of time series data. For this reason it's a No-SQL database, specifically designed for handling timestamped data, such as monitoring of industrial equipment (and this is our case), logs of user behaviours or managing of IoT devices.

The storage is efficient, and it separates the data based on the entry age. The older entries are stored in a slower magnetic storage, and eventually cancelled after a certain period of time. The newest and freshest data are stored in faster devices, because it is supposed those are the most accessed and desirable ones.

Timestream is considered a schema-less database since there's no enforced schema, but it supports SQL-like queries to provide built-in analytics, able to smooth and interpolate the high throughput data ingested.

Of course, being an Amazon product, it is well integrated with the whole Amazon cloud ecosystem [14].

3.7.3 AWS S3

AWS S3 Bucket (Amazon Simple Storage Service or simply Amazon S3 since it has 3 'S' in the name) is used for the storage of machinery documents.

It's yet another service offered by Amazon, designed for data storage and retrieval at any scale. It is very durable, reliable, scalable, flexible and integrated with the whole AWS ecosystem.

One of the most important and impressive statistic is that S3 is designed for 99.999999999% (11 9's) of durability and 99.99% availability of objects over a given year. The data is stored redundantly across multiple devices and facilities to achieve this values [15].

3.8 Docker

Docker is an open-source platform developed by Docker.inc, and today plays a crucial role in automating the deployment of application inside portable and lightweight virtual containers.

Docker was released in 2013 by dotCloud and it was specifically designed to solve the application's dependency problem. Programming an applications that runs into every computer system available on the market is quite complex, and the developers effort to develop and maintain a specific version leads inevitably to abandoning some systems to favor some of the most diffused ones. The Docker team proposed a solution to this "dependency hell" problem, giving birth to containerization, crucial technology in DevOps (Development and Operations) and CI/CD pipelines.

Docker allows developers to package applications and their dependencies into a unit known as a container. The unit will be equal in all the environments it will be run, so if it runs on your personal PC, it will run also in every other system launching that container. A container is similar to a virtual machine, but while isolating the application processes it also shares the host system kernel, making it in fact lightweight and efficient.

3.8.1 Docker architecture

The core of Docker is the Docker Engine, a client-server application made of 3 things: The Docker Daemon, the background service responsible for container management, The Docker Client, needed to interact with the Docker Daemon and the REST API which allows access to some functionality of Docker through code.

3.8.2 Docker images and containers

The architecture described is used to run containers, that are basically composed by 2 parts : Docker images and Docker containers.

Docker images are immutable read-only files, also known as Dockerfiles. They include everything that is needed from the program to run, just like a recipe, they specify everything that is needed for the correct program execution, from the very same application code to environment variables and libraries. They serve ad a blueprint for creating containers. When these instruction are executed by the Docker Engine, the Docker container is created.

Docker containers are simply instances of Docker images, their greatest perk being the fact that they are isolated from the outside system, while encapsulating everything needed to execute the software [16].

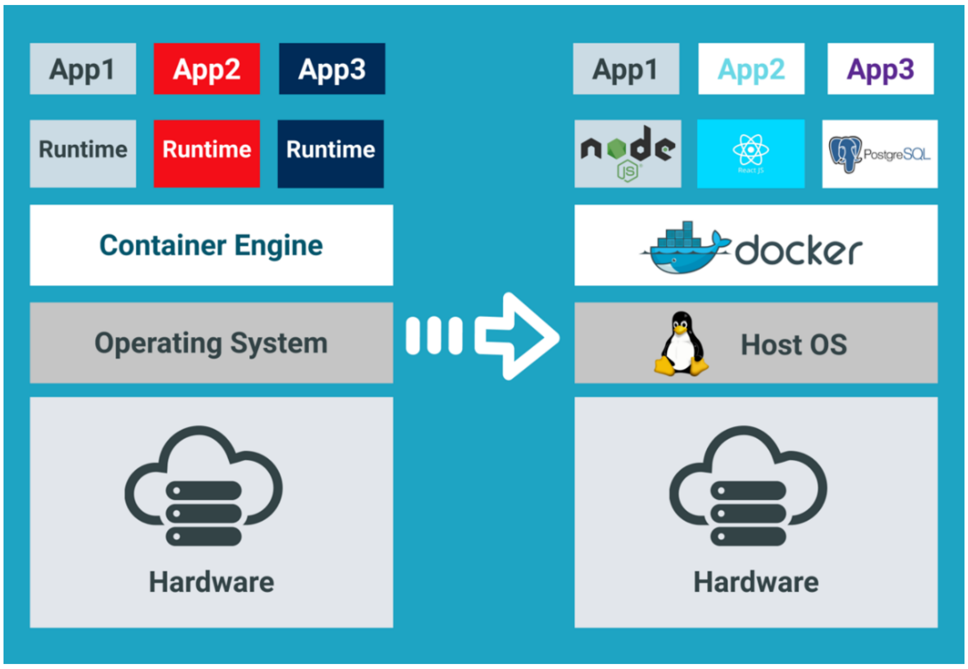


Figure 3.5: Docker Containerization

3.8.3 Docker Compose

Docker Compose is a tool used to create multi-container applications. A YAML file (Yet Another Markup Language) is read by Compose, and multiple containers will be run as a single service. Each one of them will be run independently and isolated, but is able to interact with the others when needed.

This whole infrastructure, is highly portable, well isolated, with good performances since the VMs (Virtual Machines) share the host's kernel, thus making it all highly horizontally scalable. This is why Docker has become the standard for Microservices architectures and Cloud deployments. The independence of containers is also well suited for DevOps practices, granting environment consistency between different working teams.

In our case we will use docker Compose to launch 3 different containers, one for the frontend, one for the backend and one for the application database (PostgreSQL). When launched locally the application needs 4 containers, the extra one for localstack to mock AWS3 services.

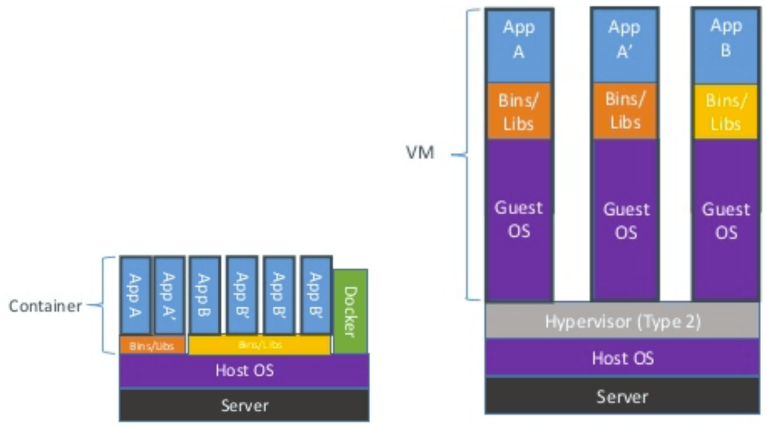
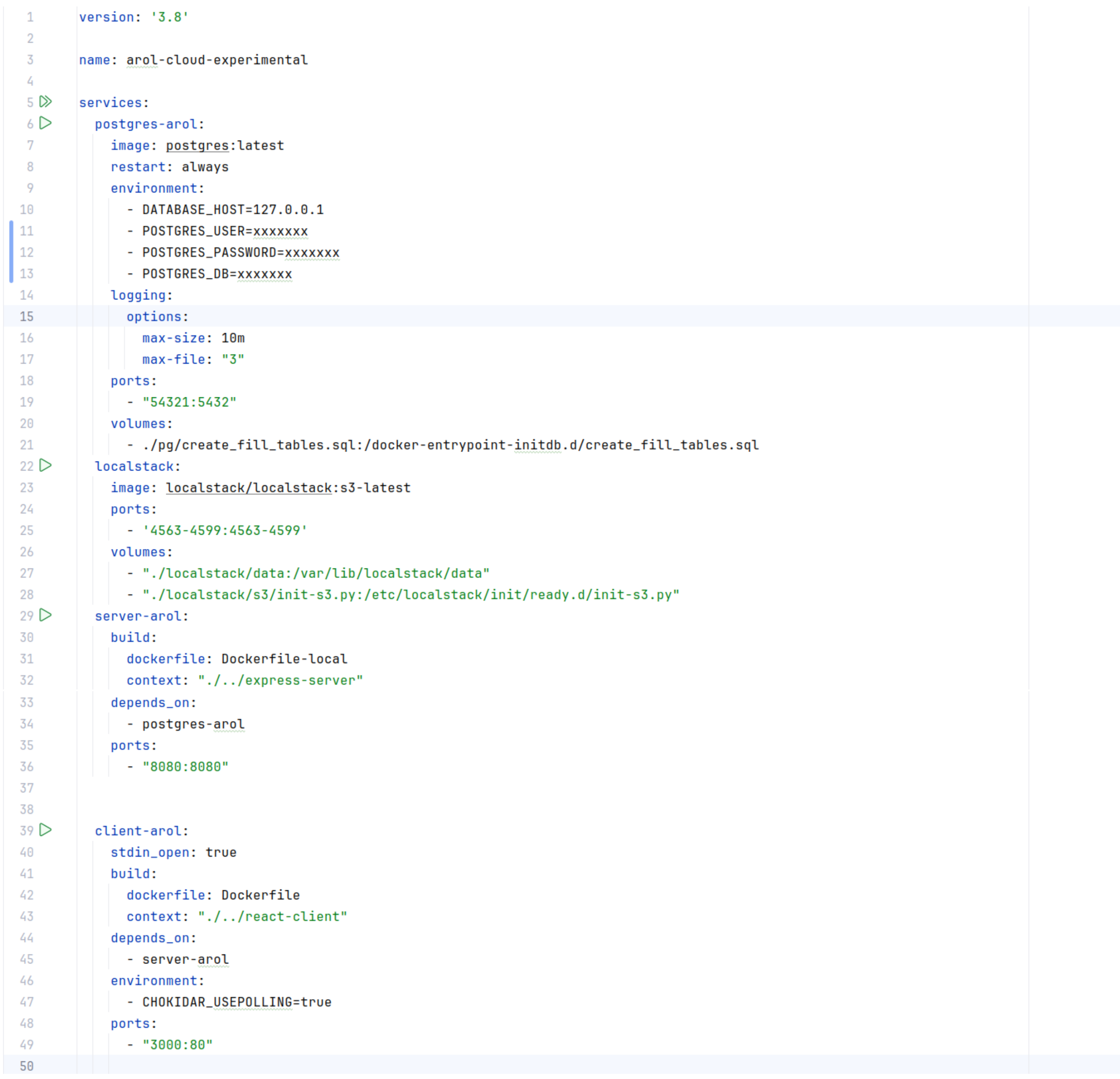


Figure 3.6: Differences between a VM and a container



```yaml
version: '3.8'

name: arol-cloud-experimental

services:
  postgres-arol:
    image: postgres:latest
    restart: always
    environment:
      - DATABASE_HOST=127.0.0.1
      - POSTGRES_USER=xxxxxxx
      - POSTGRES_PASSWORD=xxxxxxx
      - POSTGRES_DB=xxxxxxx
    logging:
      options:
        max-size: 10m
        max-file: "3"
    ports:
      - "54321:5432"
    volumes:
      - ./pg/create_fill_tables.sql:/docker-entrypoint-initdb.d/create_fill_tables.sql
  localstack:
    image: localstack/localstack:s3-latest
    ports:
      - '4563-4599:4563-4599'
    volumes:
      - "./localstack/data:/var/lib/localstack/data"
      - "./localstack/s3/init-s3.py:/etc/localstack/init/ready.d/init-s3.py"
  server-arol:
    build:
      dockerfile: Dockerfile-local
      context: "./../express-server"
    depends_on:
      - postgres-arol
    ports:
      - "8080:8080"


  client-arol:
    stdin_open: true
    build:
      dockerfile: Dockerfile
      context: "./../react-client"
    depends_on:
      - server-arol
    environment:
      - CHOKIDAR_USEPOLLING=true
    ports:
      - "3000:80"

```

Figure 3.7: Docker Compose Example

3.9 Localstack

As the times goes on Cloud services are becoming much more persistent in modern architectures. AWS (Amazon Web Services) is constantly growing and every year offers more option and more services to support need an application may have. As the number of cloud services grows over time, the applications dependency to these services also raises. Unfortunately, testing these services on the cloud is not an option, since the bills are calculated on the data used, running test suites that may insert or remove data just for testing purposes is economically unfeasible.

This is where Localstack enters in the game, an open-source tool that emulates locally AWS services, an efficient and cost-effective solution to test apps cloud functionalities. Localstack supports a wide range of AWS services, like AWS S3 (Simple Storage Service), Dynamo DB, SNS, SQS, Elasticsearch and many others. The service provided are also customizable, making the testing as close as possible to a real world scenario [17].

In order to communicate with Localstack, since a consistent environment is needed, a Docker containers needs to be up and running, in order to make the AWS SDK (Software Development Kit) able to contact the fake cloud service (the Docker container) through the 4566 port.

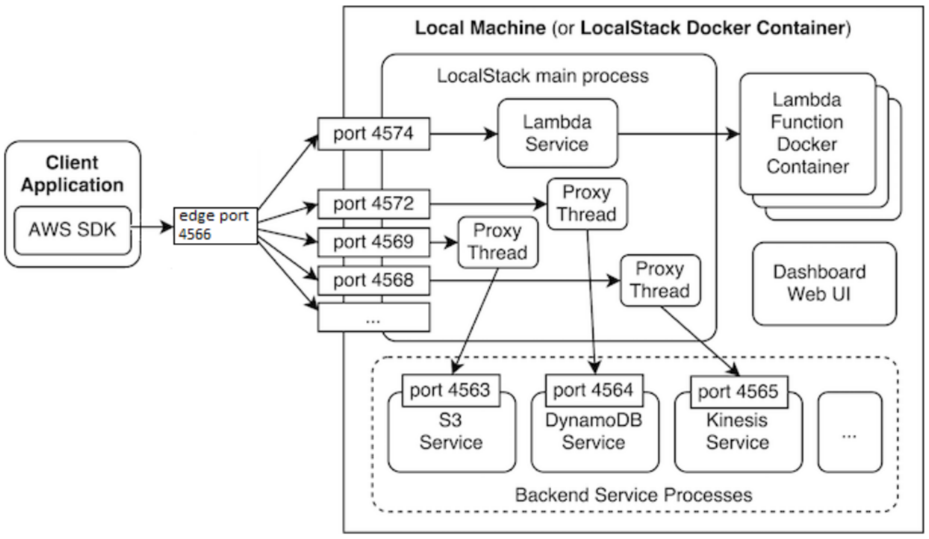


Figure 3.8: Localstack architecture

3.10 Security

Due to the wide range of web application and their exposure to the Internet, the need for robust security measures is critical to protect sensitive data, maintain users' trust and ensure the overall integrity of the whole system. Mishandled data can lead to severe consequences, not only like financial loss but also can have legal repercussions on the company.

With the word "security" in this case we refer not only to hacker attacks like man-in-the-middle schemes or SQL injections, but we encompass the overall user system safety too, since no user in the system should be able to meaningfully modify data or give inputs that could harm the system or lead to malfunctions.

One of the first step towards a robust and secure web application is a reliable authentication system.

3.10.1 Authentication

Authentication is the process of verifying the identity of a user before granting them access to resources, ensuring that only authorized individuals can interact with sensitive information and systems.

There are different systems to implement the authentication mechanism, compared and discussed below, but in our web application we opted for a Token Based Authentication, using JWT (JSON Web Token).

1. Password based authentication - Basic security level, very vulnerable to brute force attacks and phishing, easy to implement but difficult to scale. Simple and widely used but less secure. Best for small applications where security is not a primary concern.

2. Token based authentication - Good security, tokens are short lived and can be encrypted, it’s highly scalable with moderate vulnerabilities since token can also be intercepted, stateless. Provides higher security and scalability, suitable for APIs. Reduces the need for session management.

3. Multi-factor authentication (MFA) - Very high security combining multiple factors, really complex to implement and cumbersome for the users since it requires multiple steps, really high implementation complexity, better suited for high security environments

4. Biometric authentication - High security, biometric information are really unique and hard to replicate, with moderate complexity and easily scalable. Provides high security through unique user traits. Best for applications requiring stringent security measures, though it requires specific hardware.

3.10.2 JWT - JSON Web Token

JWT is a proposed Internet standard (RFC 7519) for transmitting information between two parties as a JSON object. These tokens are signed to ensure the integrity and authenticity they claim to contain. Unlike session based authentication, JWT allows for stateless authentication, which comes in really handy when dealing with distributed systems or microservices architectures.

A JWT is composed of three parts: Header, Payload and Signature, opportunely concatenated with dots(.) to form the whole token.

1. Header - typically contains two parts: the type of token used (JWT in our case) and the signing algorithm used such as HMAC, RSA or SHA256

2. Payload - The payload contains the claims, the statements about the entity who want to access our services

3. Signature - to create this part we need to take the encoded header, the encoded payload, a secret and sign that all with the encryption algorithm specified in the header. this is the part that ensures that our messages and infos weren't tampered along the way.

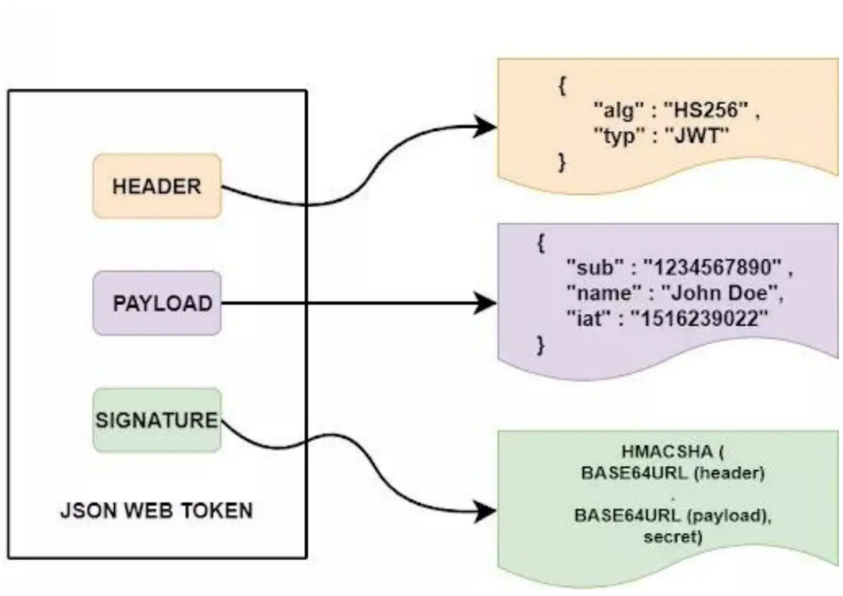


Figure 3.9: JSON JWT object [18]

When a user successfully logs in, the server creates a JWT and sends it back to the user. This token contains all the claims needed for security and is signed to prevent tampering. The user will store the token client side or in the local storage. For every subsequent request, the client need to include the JWT in the Authorization Header. The server will then proceed to verify the token's signature, check the claims and authenticate the user.

Since JWTs are stateless and expire after some predefined time, handling token refresh is crucial for maintaining user sessions without asking the user again for authentication. Usually when the user logs into the system 2 different types of tokens are generated : an access token, used to perform API requests, and a refresh token, used to obtain a new one when the current one expires. The access token has usually a short lifespan (usually 10 minutes but can be defined by the programmer) to minimize the risk of someone snuffing it out. When this happens the client sends a request to the server refresh token endpoint with its own refresh token. The server will proceed to verify the refresh token and issue a new access token, so that the client can update its old one.

In the end JWT is a good, reliable and safe approach to authentication, nonetheless it has some pros as well as some issues.

Potential issues that may arise are due to inconsistency in its implementation or in the chosen algorithm. Stronger algorithm are recommended like RSA (Rivest-Shamir-Adleman algorithm). The secret key used to validate the tokens is also a point of potential failure, because if compromised, then tokens can be forged by anyone. Moreover when a token is issued is difficult to revoke, posing yet another security risk.

The advantages compared to other strategies are not negligible too, like the fact that they can be safely sent via URL POST parameters or in HTTP headers, or the fact that they contain all the required information about the user, eliminating the need of eventual database lookups.

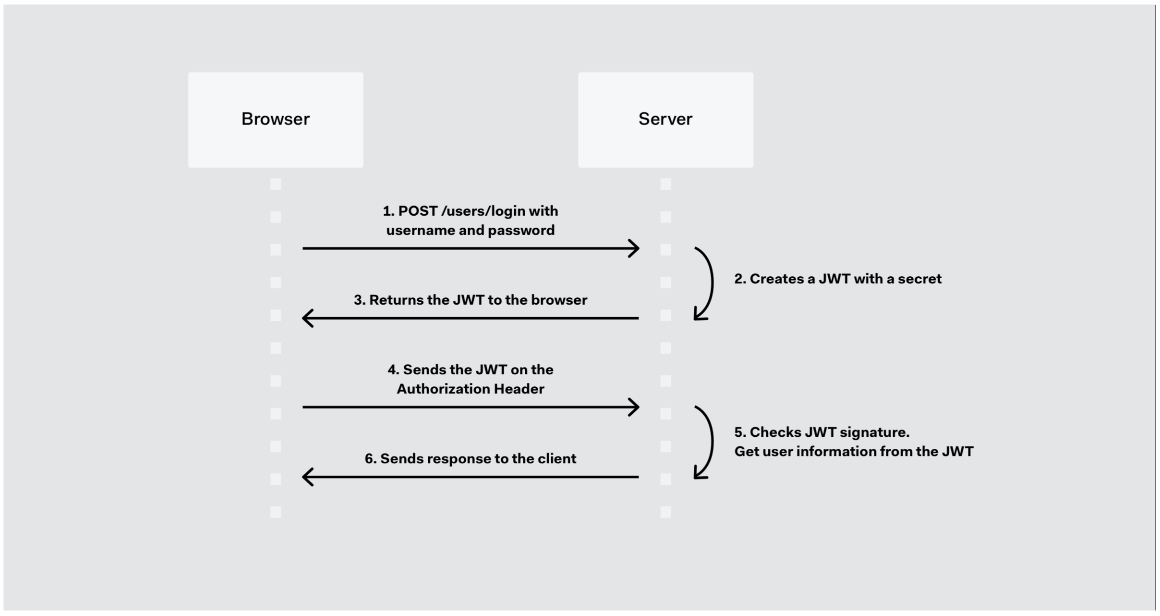

Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the browser
4. Sends the JWT on the Authorization Header
5. Checks JWT signature. Get user information from the JWT
6. Sends response to the client

Figure 3.10: Schema explaining the normal functioning of a JWT token [19]

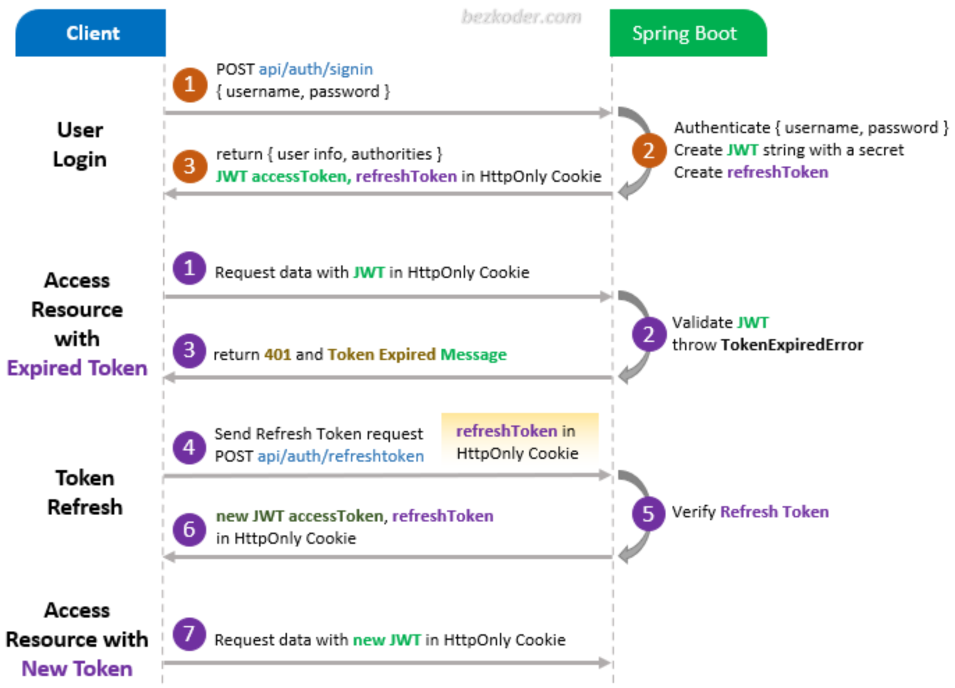

Client
bezkoder.com
Spring Boot
User Login
1 POST api/auth/signin { username, password }
2 Authenticate { username, password } Create JWT string with a secret Create refreshToken
3 return { user info, authorities } JWT accessToken, refreshToken in HttpOnly Cookie
Access Resource with Expired Token
1 Request data with JWT in HttpOnly Cookie
2 Validate JWT throw TokenExpiredError
3 return 401 and Token Expired Message
Token Refresh
4 Send Refresh Token request POST api/auth/refreshtoken
refreshToken in HttpOnly Cookie
5 Verify Refresh Token
6 new JWT accessToken, refreshToken in HttpOnly Cookie
Access Resource with New Token
7 Request data with new JWT in HttpOnly Cookie

Figure 3.11: Schema explaining the refreshing of a JWT token [20]

3.10.3 Role based access

Given the huge number of different kinds of data present in the application and the huge number of users that must be able to use it concurrently, the access to resources is controlled through role based access, which gives users of a specific roles only certain permissions. Since this roles are unchanged we will briefly discuss them, giving them context and purpose. The roles are the following: ADMIN, MANAGER, WORKER.

1. Administrator: Has the highest authority within the company. It is granted permission to every entity that belongs to his company, overseeing all aspects of machinery and user administration.
2. Manager: acts as a delegate of the administrator, possessing similar permission, but only on a subset of machineries that were granted to him by the administrator. While he is able to modify user profiles, managers cannot delete users.
3. Worker: He occupies the lowest tier of the hierarchy. Most of his actions are limited to interacting with dashboards and documents. He cannot manage users, cannot modify them and he is involved in daily routine operations, consulting documents and dashboards for his task execution.

3.11 Starting point

This was a brief resume of the starting point of the app, which was expanded and improved mainly to achieve its original objectives and improve on the industrial and professional development aspect, introducing new frameworks and CI/CD integration tools, with the intention to provide it in the future to all AROL customers, also laying the basis for its potential scalability and adaptation in other contexts. Through this examination, this research seeks to offer valuable insights into best practices for web application development and the future trends shaping the industry.

Chapter 4

New functionalities

In this chapter we will discuss about the new functionalities that have been added to the application to make it more compliant to the requirements defined more than an year ago by AROL. All the core functionalities were already in place, but management of users, companies and templates has been added to grant AROL a better control on the whole system, making a single trusted user able able to interact and manage the whole system through an intuitive user interface, without any technological skill required other than understanding the interface. We will now briefly present the new features as we will discuss them more in depth in the chapter

- Superuser : Complete control over the application. This user has to be in the AROL company.

- Company Management features : The superuser can manage all other companies, creating new ones and deleting old ones.

- User Management features : Users can now be created, updated, disabled and deleted. User's passwords can be reset, permissions can be modified, users can be filtered and searched.

- Machinery Management features : Machines can now be created, updated, deleted, modified, filtered and searched.

- Template creation and management : Now AROL can create dashboard templates, so that users don't have to create them from scratch.

4.1 Superuser

A new role has been added. The Superuser will the user entrusted with the control of the whole application, who can give or revoke access and permissions to data to any other user in the system. The Superuser is basically the most powerful user of the application and it can be really useful even simply for debugging purposes since he can access every resource and every functionality ever implemented.

The addition of the Superuser role is critical for two reasons. First and foremost it allows the complete control of the system directly to AROL, where even a non-programmer can be in charge of managing and monitoring every company everywhere in the world. Second, but not less important, it allows for excellent customer support. For any problem with the application the superuser can login and verify the issues in a matter of seconds, as well as provide documentation or realize specific dashboards templates tailored on the customer's needs.

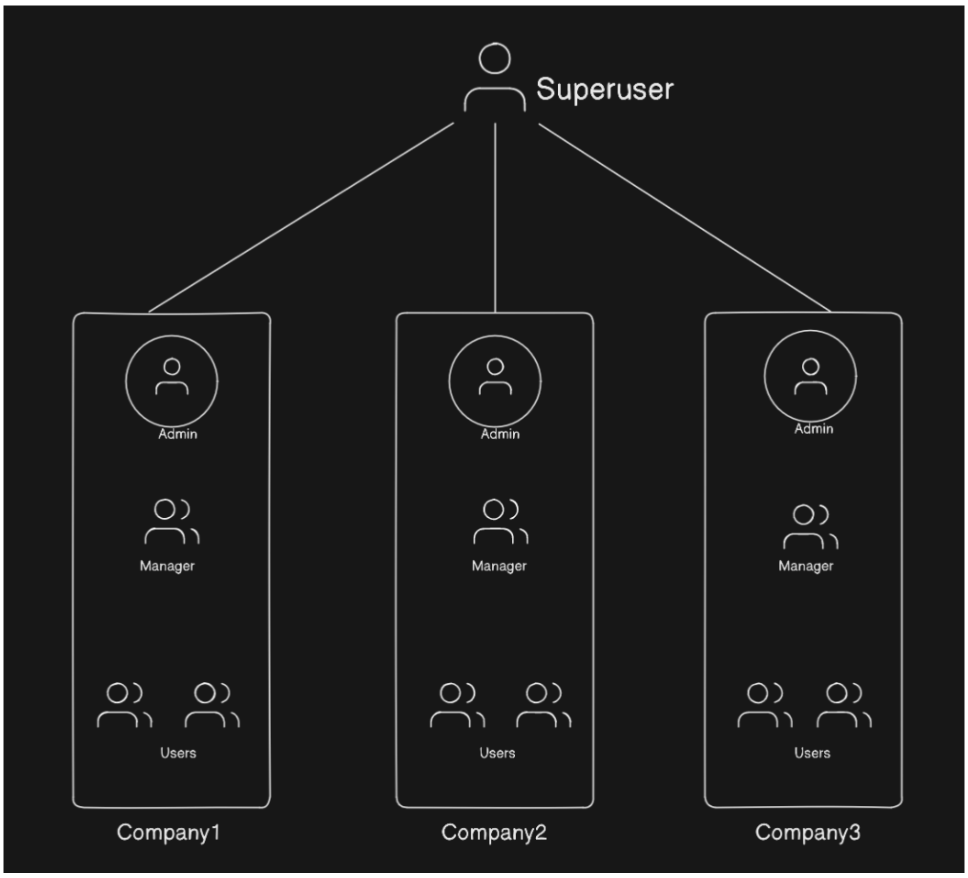


Figure 4.1: Screenshots of Superuser schema

AROL is the only company allowed to have Superuser roles, so the superuser will be able to see (and eventually test) all the functionality in the application, for internal tests and quality assurance, so that AROL itself can use its own application to monitor its own machines. This means that the superuser belongs to a company like all the other users, can use and test every functionality, and can manage and control every other user of the other companies.

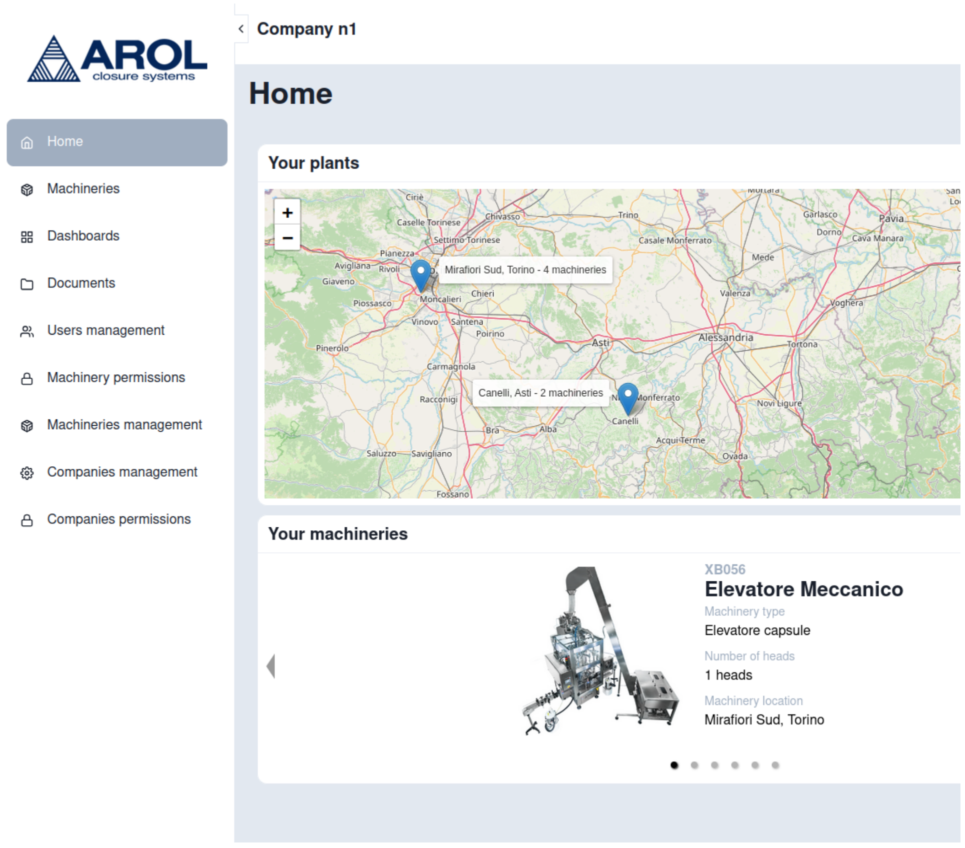


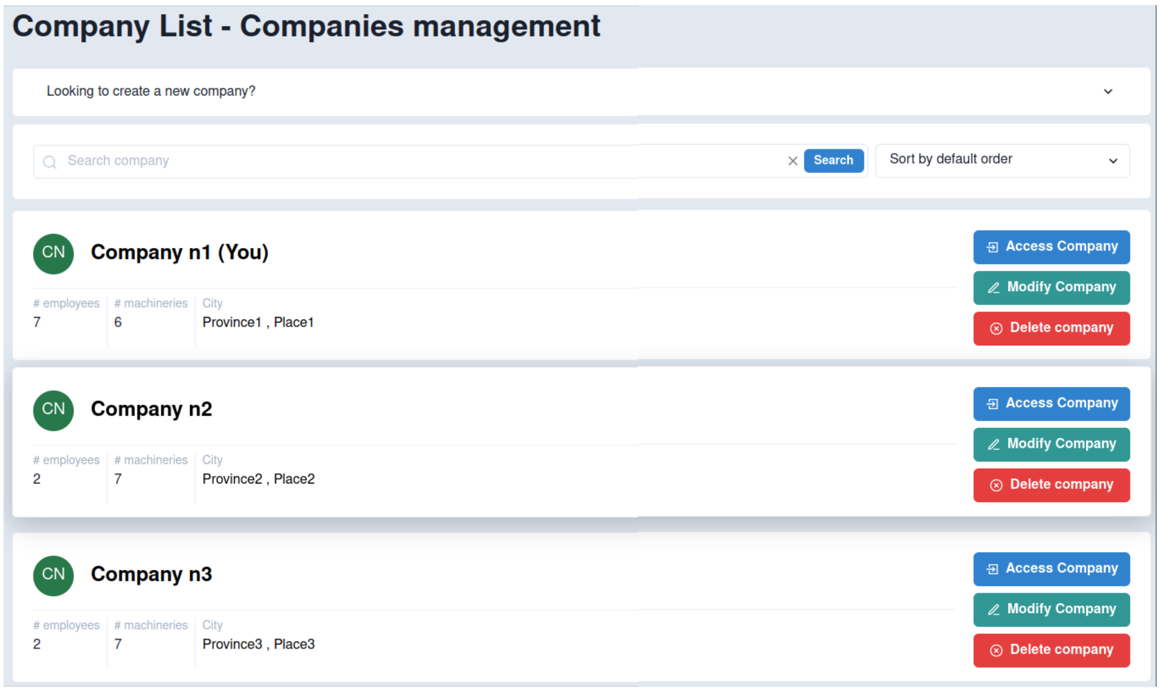
Figure 4.2: Screenshots of Superuser interface

4.2 Company management

One of the key features of this work, strictly tied to the Superuser.

A company is in fact a company, a customer of AROL who bought one of more of its machines. A single company basically containerize the whole package of machines, documents, users and templates, making the management of all of it much easier. Another advantage is that each company is independent and cannot see any data belonging to other companies. Each company is uniquely identified by an ID, a name and a city. Company management allows the creation of new ones, making the number of potential AROL customers virtually unlimited.

While the old version of the application had basically the scope of a single company, creating multiple ones means a better partition of users, machineries, and allows for a better and faster management, as well as an easy and convenient way to make some data available only to a single customer. Companies can be created and deleted by the superuser as he sees fit, and ideally they should coincide with real companies, AROL's customers.


Company List - Companies management

Looking to create a new company?

CN Company n1 (You)

# employees	# machineries	City
7	6	Province1 , Place1

CN Company n2

# employees	# machineries	City
2	7	Province2 , Place2

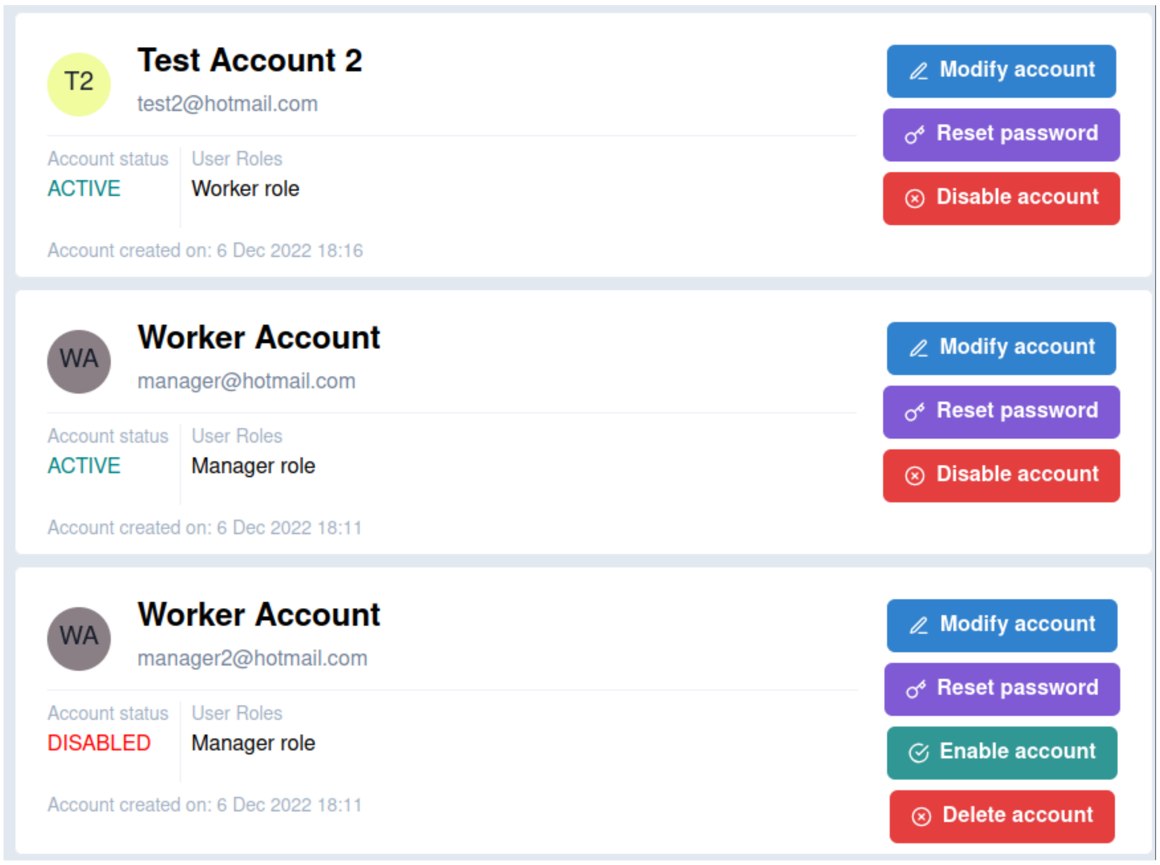
CN Company n3

# employees	# machineries	City
2	7	Province3 , Place3

Figure 4.3: Images of Company Management

4.3 User management features

A new module has been developed with the objective of facilitating the creation of new users within the AROL system. This module existed previously, and it is now accessible in multiple parts of the application, like when accessing a company management page or when managing user's permissions. It is also possible to update an existing user account with the added feature of enabling or disabling a certain account, or simply reset their password, so that every user can chose the one they prefer to access the system



Test Account 2
test2@hotmail.com
Account status: ACTIVE
User Roles: Worker role
Account created on: 6 Dec 2022 18:16

Worker Account
manager@hotmail.com
Account status: ACTIVE
User Roles: Manager role
Account created on: 6 Dec 2022 18:11

Worker Account
manager2@hotmail.com
Account status: DISABLED
User Roles: Manager role
Account created on: 6 Dec 2022 18:11

Figure 4.4: Users management menu

Also, users can be disabled or enabled if user has at least admin permission. Disabled users won't be able to login into the systems until they are enabled again. Disabled users are eligible to be cancelled. Cancelling a users requires a two-step confirmation, that ensures careful consideration, with the option to confirm the deletion or revert the action and enabling the account. The deletion process under the hood is managed by database triggers, that allows the database to remain consistent and remove every piece of data regarding the old user like permissions on machines or documents.

A filtering and search component has been developed since as the number of user using the system get bigger, the need for user searching and filtering becomes more crucial. Users can now be filtered by status, creation date and role, and can be filtered only by logged in users with user management permissions like admins and managers.

4.4 Machinery Management

Each machinery has now its own drop-down menu where all kinds of operations can be performed. Machinery can now be created, updated, deleted and searched through filters. The filter and search component for machineries operates in an analogous manner to the user filter. The search term is intersected with the applied filters for a more refined and customized search result. Machines can be filtered for model and type, number of heard dashboards and documents.

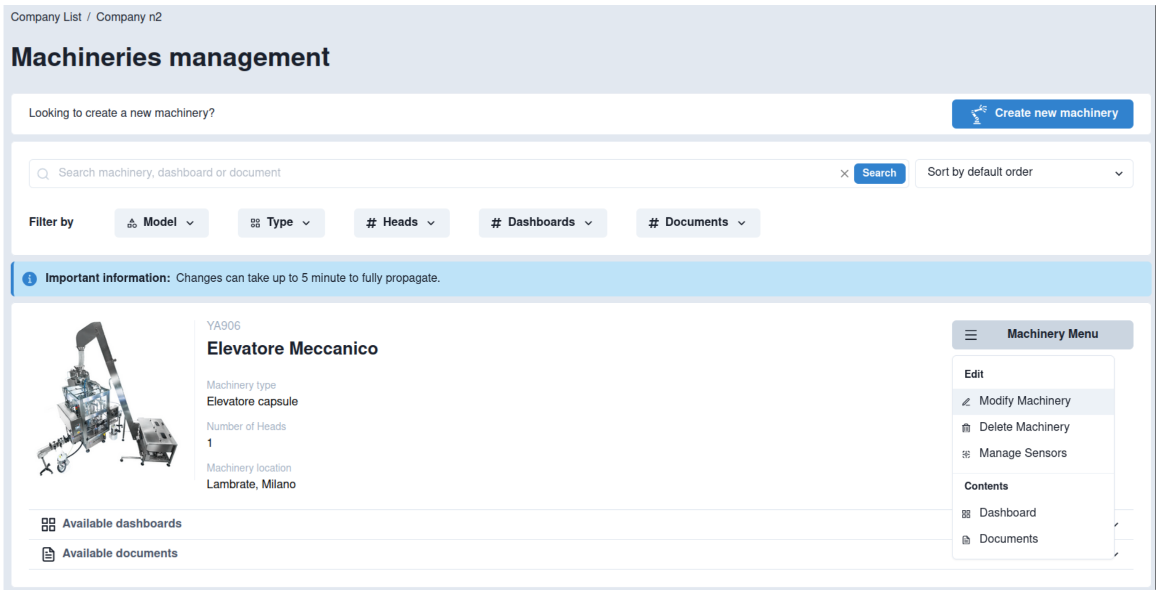
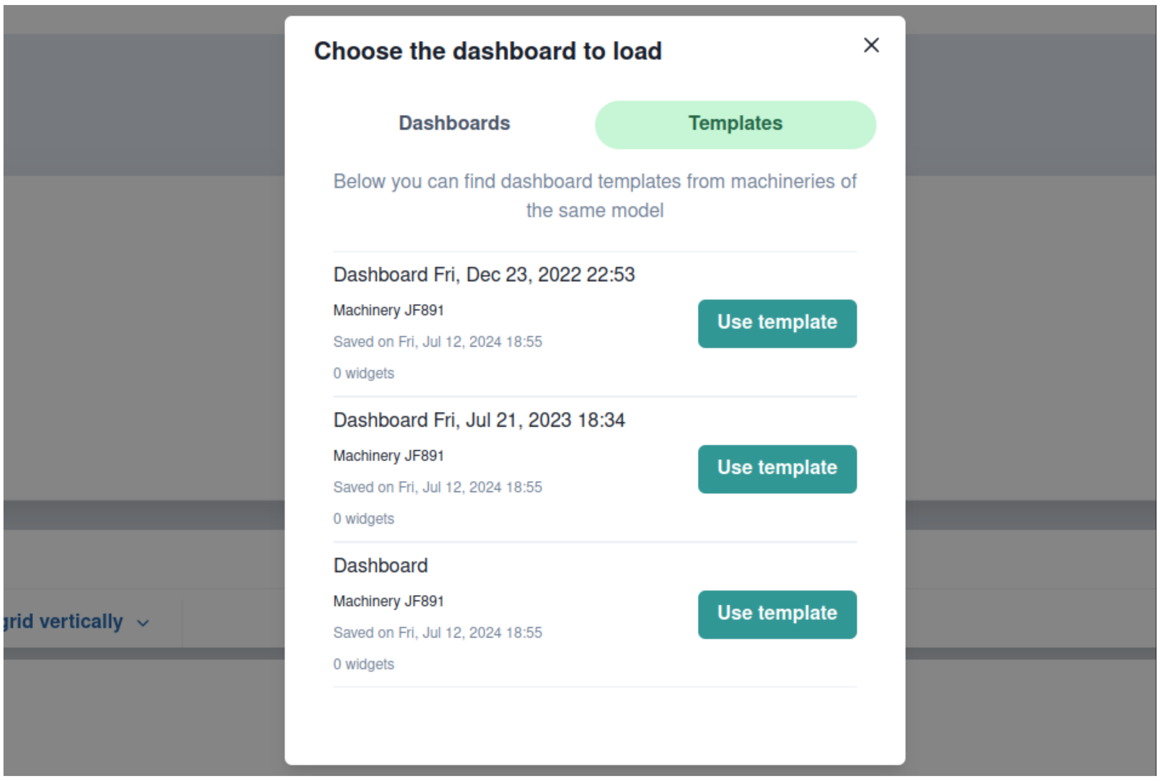


Figure 4.5: Machinery management options and filters

A filtering and search component has been developed since as the number of machines in the system gets bigger, the need for searching and filtering becomes more crucial. Machineries can now be filtered by model, type, number of heads, dashboards and documents.

4.5 Template creation and management

Probably the most useful addiction to the application. A template in this context refers to a predefined dashboard structure, designed to simplify the monitoring and management of dashboards, particularly for users who may not have the technical expertise needed or just are not sure about what widgets to use and in which scenario. The possibility of creating a dashboard for scratch is useful and offers endless possibilities of customization, but it is a complex and technical task that many user may want to avoid. The templates solve this issue, giving customers a predefined dashboards, tailored on the machine that users are working with, and still giving them the power to modify and update it with the widgets and data they may like.



Below you can find dashboard templates from machineries of the same model

Dashboard Fri, Dec 23, 2022 22:53
Machinery JF891
Saved on Fri, Jul 12, 2024 18:55
0 widgets

Dashboard Fri, Jul 21, 2023 18:34
Machinery JF891
Saved on Fri, Jul 12, 2024 18:55
0 widgets

Dashboard
Machinery JF891
Saved on Fri, Jul 12, 2024 18:55
0 widgets

Figure 4.6: Templates selection example

4.6 Notification System Design

In today's digital age an effective communication with customers is critical for the success of every application, and in AROL's case, notification could be used to perform some kind of interaction with the customer, whether it is for marketing purposes, or just to alert that the system may be down due to an update or even for signaling that maintenance on some kind of machine is due.

4.6.1 Requirements

Before starting to design a system, it is important to have a clear idea of what the program needs to achieve.

This design is focused on providing the most functionalities possible, in the context of what functionality may really be useful, with minimal changes to the original code and keeping in mind the potential scalability of the system.

Given the fact that our software isn't a social network the complexity of this system greatly diminishes, since there are no followers to keep track, and the communication is almost only in one direction: from AROL to customers.

The main goals were the following :

1. Minimal changes to the existing infrastructure
2. Most flexible and most useful functionalities given the context
3. Different types of possible notifications
4. Possibility of sending notifications only to a certain role or company
5. Scalability to handle high volume of notifications
6. Security and privacy

4.6.2 System Design

Given the fact that our software isn't a social network the complexity of this system greatly diminishes, since there are no followers to keep track, and the communication is almost only in one direction: from AROL to customers.

One initial solution to save the notifications in a database was something like this:

NOTIFICATIONS:

id (PK)	**userID**	**type**	**title**	**message**	**date**	**isRead**
integer	integer	string	string	string	string	boolean

Table 4.1: Initial Notification Table Structure

But this solution has a fatal flaw. In the context of this application and with the volumes that AROL may expect this solution could work, but it has one big problem.

Title, message and data this way are repeated for each user in the system, meaning that if we want to notify something to a thousand of users in our system, we would have of course a thousand entries in the notification table, but those three properties would be the same for everyone, because it's the same notification for every user.

For this reason a new better design was studied, and the database table has been split in two:

NOTIFICATIONS TABLE:

id (PK)	**userID**	**type**	**notification-content-id** (FK)	**isRead**
integer	integer	string	integer	boolean

Table 4.2: Notification Table Structure

NOTIFICATION CONTENTS:

notification-content-id (PK)	**title**	**message**	**data**
integer	string	string	JSON

Table 4.3: Notification Content Table Structure

Please note that in the tables "PK" stands for Primary Key and "FK" stands for Foreign Key. This way we solve the problem of duplicated messages and titles, relying on a external key to retrieve the content of the notification, that will be stored in the database only once.

The type and isRead values are left in the original table, since they are useful to the FE rendering. To be specific, the isRead value is a boolean value used to determine whether we need to render the notification icon or not, the type value specifies the various kinds of notification that may be sent or received.

4.6.3 Implementation

To properly use and leverage the two tables proposed in the design, some new APIs need to be added to the BE:

1. getUsersByRole() - to get every user of the system of a certain role (only for Superuser)
2. getNotifications() - to be called by every user to get its notifications

3. getNotificationContent() - to be called when notification data is required

4. setNotificationIsRead() - to update a single notification and set its isRead field

The first function is useful for the superuser to send notification only to users of a certain role. The second function is important to get the number of notifications that still has to be read by the user as well as the respective notification-content-id. The third function is really important, since it exist only because we divided the notification table into two. One could argue that it makes the system overly complex, since we have to do one extra call to the server (and consequentially the DB) just to get the content of our communication, but the added benefits of not repeating hundreds of times the same notification content greatly surpasses any downsides that one may come up with, not to mention that doing things this way makes it a lot more scalable. The fourth function is useful to properly visualize pending notifications, since the "isRead" field will primarily be used by the FE.

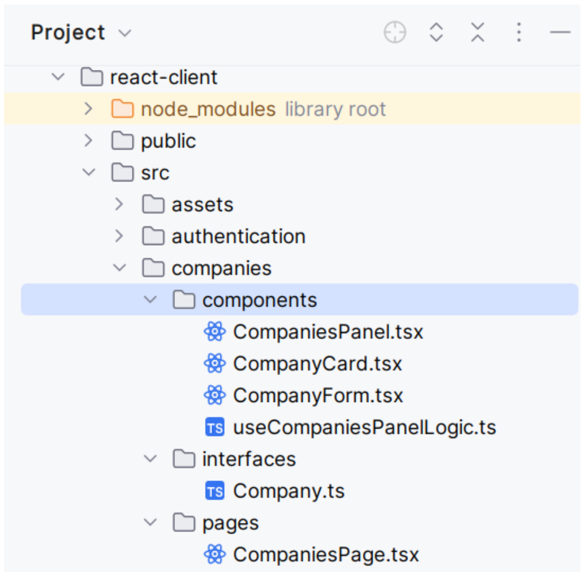
4.6.4 Optimizing Frontend folder structure and organization

Having a well-organized FE folder structure is crucial for maintainability and scalability, and it ensures the project remains clean and easy to navigate, making it simpler for developers to collaborate and manage the project over time.

The source directory (/src folder) is the core of the project, containing all of the source code. There are several viable strategies for organizing the code, in this case the chosen approach is an hybrid between grouping files by route and grouping them by type, with the idea of having the best of both worlds.

Grouping files by route makes the files and folders directly communicate the application features, but confusion arises when a component needs to be shared between different routes, inevitably leading to a future refactoring.

Grouping files by type on the other hand has the advantage of having a flatter file and folder structure, but finding out which files are used by which features gets harder as the project gets bigger, and integrating new people into the project might be tricky since the structure does not communicate the application features.



```
Project
react-client
  node_modules library root
  public
  src
    assets
    authentication
    companies
      components
        CompaniesPanel.tsx
        CompanyCard.tsx
        CompanyForm.tsx
        useCompaniesPanelLogic.ts
      interfaces
        Company.ts
      pages
        CompaniesPage.tsx
```

Figure 4.7: FE folders structure

The obvious advantage is that we can be sure that everything concerning the "companies" components is in fact in that folder. Moreover this folder is further subdivided in components, interfaces and pages, giving the reader more clarity and organization.

Another FE improvement is the subdivision of big files into logic and components. The *useCompaniesPanelLogic.ts* file basically takes all the logic needed by the *CompaniesPanel.tsx* component and it regroups everything into a single file, ultimately resulting into 2 different files of manageable size, instead of having a single big file that has to display and manage the logic of all of its components (and could easily go over the thousand lines of code).

Chapter 5

Technological Improvements, New Tools and Libraries

This chapter will be dedicated to all new architectural improvements and library or tools addiction that has been done in order to make the application easier to develop and more compliant to the industry standard. The objective of this chapter will be to illustrate and propose all the updates done to the system in order to make it more flexible, scalable and/or maintainable.

5.1 Drizzle ORM

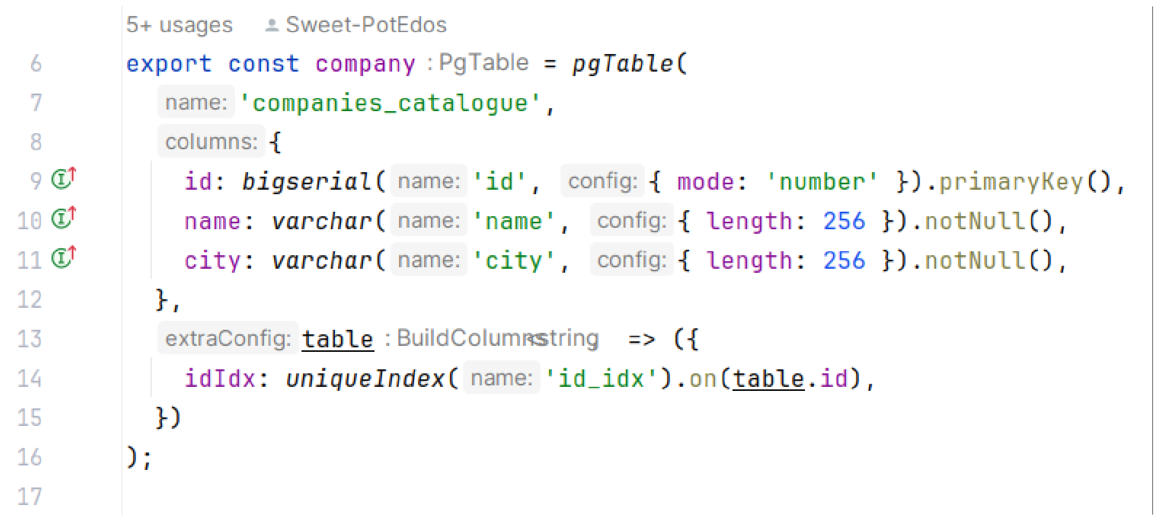
Drizzle is a modern and lightweight ORM (Object Relational Mapper), designed to simplify and make application-database interaction safer.

It is especially indicated for Typescript-based projects, since it leverages Typescript strengths to offer type safety, intuitive API for constructing queries and flexibility. Of course when needed we can specify raw SQL queries, even though the whole purpose of an ORM is to use its syntax to produce automatically optimized queries for your tables. Of course as any other database interface Drizzle needs to be configured, but fortunately it is pretty straight forward. Specifying the type of the database, the host, the port, username, password and database name is a common practice for any database, and this is enough for drizzle to connect to it.

Another upside is Drizzle's compatibility with a large number of the most used databases in the industry. The fact that Drizzle code is independent from the database underneath, allows extreme flexibility, and makes changing the database system a really easy process that requires very little to none code refactoring, granting the programmers the freedom of changing the databases as they see fit, while also leaving the project open to eventually change database system in the future as the services grow larger and larger. Long story short, It makes the project highly scalable.

5.1.1 Model definition and query construction

In Drizzle, Models are classes that represent the database tables. Table definition it's deeply tied to Typescript type safety, since each class property corresponds to a table column and annotations are used to define these mappings, seamlessly translating application objects into database records.



```
5+ usages   Sweet-PotEdos
export const company : PgTable = pgTable(
  name: 'companies_catalogue',
  columns: {
    id: bigserial( name: 'id',  config: { mode: 'number' }).primaryKey(),
    name: varchar( name: 'name',  config: { length: 256 }).notNull(),
    city: varchar( name: 'city',  config: { length: 256 }).notNull(),
  },
  extraConfig: table : BuildColumnsstring  => ({
    idIdx: uniqueIndex( name: 'id_idx').on(table.id),
  })
);
```

Figure 5.1: Example of a Drizzle table

Query construction gives meaning to the whole ORM concept. It makes building query fast and more importantly perfectly readable and maintainable. After a query is built it is executed against the database (and implicitly translated into optimized raw SQL). The query result will be translated back to the defined model classes. This whole process basically allows the developer to exploit Typescript type safety and autocompletion features, making the interrogation of the database a pretty fast and efficient task

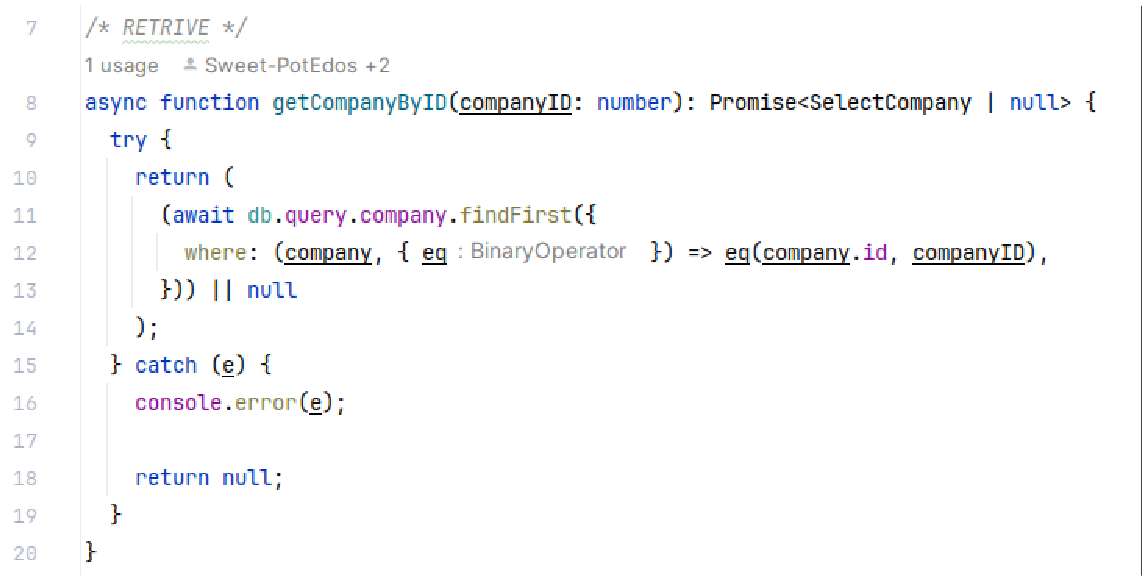

```
/* RETRIVE */
1 usage  Sweet-PotEdos +2
async function getCompanyByID(companyID: number): Promise<SelectCompany | null> {
  try {
    return (
      (await db.query.company.findFirst({
        where: (company, { eq : BinaryOperator  }) => eq(company.id, companyID),
      })) || null
    );
  } catch (e) {
    console.error(e);

    return null;
  }
}
```

Figure 5.2: Query construction example

5.1.2 Database migration and transaction

Database migration is one of the most useful Drizzle features and it allows to incrementally evolve the database schema as the application requirements changes. A migration represents a set of changes, contained in a script, that can be applied or rolled back to switch the database schema from one version to another. Also Drizzle itself keeps track of the applied migrations, ensuring that each one is applied only once.

Database migrations are really useful in deployment scenarios, if we suppose we need to deploy a new version of the application, and this new version needs a different db schema, we can run migrations automatically as part of the deployment process, ensuring that the new version of the db schema is updated correctly and basically granting consistency between development, testing and production stages.

At the same time migrations can be easily rolled back, mitigating the impact of any fatal changes, and significantly reducing the application downtime. It's important to consider that the migration itself is some kind of document that accurately describes any database changes [21].

5.2 ZOD validation

ZOD is a TypeScript-first schema declaration and validation library with static type inference.

Let's start with defining what a validator is. A validator is a tool or in our case a function used in software development to ensure that some data is conformed to a specific format or structure. It is commonly use to check input data or data transmitted through APIs. Slight errors in the data received by the server could cause issues of any sort, and sometime things may take a bad turn.

There are plenty of other libraries for doing the data validation but ZOD was chosen because of its perks and approach. With ZOD, you declare a validator once and Zod will automatically infer the static TypeScript type (ZOD works with plain JavaScript too, but this feature just makes the decision of using Typescript much more valuable). It's easy to compose simpler types into complex data structures.

ZOD is :

1. Designed to be developer friendly : really easy to use. Its goal is to eliminate any duplicate declaration

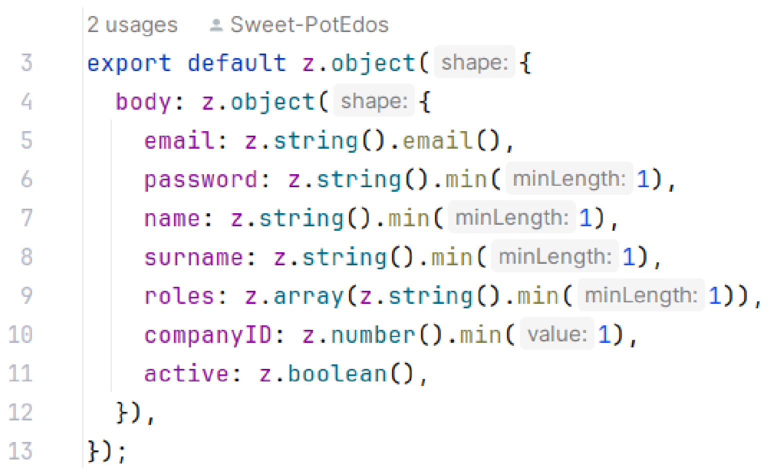
2. Lightweight : 8kb minified + zipped.

3. Has zero dependencies : can be used in any project almost without impacting its size

4. Works in Node.js : and in all modern browsers

5. Immutable : its methods return a new instance

6. Concise and chainable interface : declaring schemas and chaining them is really intuitive



```
2 usages   Sweet-PotEdos
export default z.object( shape: {
  body: z.object( shape: {
    email: z.string().email(),
    password: z.string().min( minLength: 1),
    name: z.string().min( minLength: 1),
    surname: z.string().min( minLength: 1),
    roles: z.array(z.string().min( minLength: 1)),
    companyID: z.number().min( value: 1),
    active: z.boolean(),
  }),
});
```

Figure 5.3: User Schema Example

The code is very simple, it can be reused multiple times and also allows transformations during the validation process, again ensuring the the data is exactly the way it is requested by the schema. All the 403 errors (BAD BODY) are thrown by a single line of code, common in the whole server-side, making the error handling quite precise and informative.

All these features make ZOD a powerful yet versatile validation tool, making it an excellent choice for ensuring data integrity into application and to write a well-organized, simpler and cleaner code [22].

5.3 Test Suite

The larger a project is the harder it is to maintain code quality and ensure reliability. Keeping track of dependencies between components and features, as well as a simple refactoring of some functions, as the project grows linearly becomes exponentially harder, and eventually at some point it becomes close to impossible.

The only way to overcome these difficulties is through the use of a comprehensive test suite, which will provide us with some insightful metrics and data the developers can use to ensure their code functionality and reliability. After a test suite has run, it usually produces some kind of result, usually a document where the test coverage is explicitly stated. Coverage is literally a measure of how much code had been covered by the different types of tests. The higher the percentage of the code coverage is, the better the test suite will be, and higher code quality will be the result. Different types of tests can be written, the two most important ones are Unit tests and Integration tests, which yields better results if combined together since they have different purposes.

5.3.1 Unit Tests

Unit tests verify only a small portion of the application, generally singular functions or methods. They are used to ensure that every function or method behaves exactly as intended, as they have the smallest scope possible. The bigger and larger a function is, the harder it is to correctly test and verify. Knowing this it becomes clear that big functions that do a lot of things are just bad programming practice since they are non trivial to test. The upside is that singular functions have usually little to no dependency since they are isolated, are fast to run and very simple since their narrow scope.

5.3.2 Integration Test

Integration tests are a little bit more complicated than unit tests since they evaluate the interaction between different modules and services in the application, ensuring they work together as expected. Their purpose is to ensure that combined parts of the application function correctly together and to identify any issue that may arise from the integration of different components. They usually involve real instances of a database rather than mocks or stubs, they provide a higher level of coverage of code and they have usually slower performance to run respect than the unit ones. Integration tests are typically used (like in our case) to verify the correct functioning of the APIs.

By calling one by one all the APIs in our back-end system we can drastically reduce the insurgence of problems in our back-end, and we can test even the most unusual combination of data, significantly raising the chances that our system behaves as intended and that the results of our API calls are consistent throughout the whole application.

5.3.3 Benefits

Let’s see some of the upsides of having a well done test suite before looking into the details.

- Improved Development cycle : when tests are automated and properly inserted in a pipeline, developers can quickly verify that new changes do not break existing functionalities. This continuous integration and continuous deployment (CI/CD) approach makes the delivery of new features easier, reducing manual testing and the time spent debugging.

- Better Code Quality : tests can also highlight small bugs and errors, before they propagate further in the development process and cause more significant issues.

- Easier Collaboration : more people working in a project also means more opportunity for everyone to inadvertently affect other's people work. If the test suite runs and finds no errors, the chances that everything still works are pretty high.

- Better Documentation : the documentation explains how the code works, which kind of input data is expected and which kind of output you should get. Through test we verify that the documentation is actually correct.

- Easy Refactoring : code refactoring is a standard (and common) practice during development, and since it is just a refactoring and no functions of the code should be altered, having comprehensive tests is the best way to ensure this process happened flawlessly.

- Less issues post-release: the likelihood of post-release issues and customer dissatisfaction are significantly lower since automated tests provide some assurance that the code will perform as expected in production.

- Cost Efficiency : setupping a test suite and writing test requires a significant amount of time, but it pays off in the long run since usually the later bugs are found the more they cost to be fixed.

5.4 Frameworks and Libraries

The terms "library" and "framework" are often used in software development context, and they may seem similar, but they pack some differences that is better to highlight.

Libraries are a collection of pre-written code that used to add predefined functionalities to applications. They usually solve common programming problems, generally allowing developers to write faster and more efficient code, with fewer errors. Libraries have usually a narrow scope addressing specific functionality, and they are designed to be used across different parts of the application.

A Framework in the other hand is a more comprehensive structure that provides a foundation on which developers can build entire applications. It offers a kind of skeletal/template support that dictates the architecture of the application.

Key Differences	**Library**	**Framework**
Control Flow	Developer's code is in control and calls the library.	Framework is in control and calls the developer's code.
Scope and Purpose	Typically focused on a specific task or a set of related tasks.	Provides a comprehensive platform for building an application, covering many aspects like data handling, UI, and control flow.
Usage	Used as needed, allowing developers to pick and choose which parts to include in their project.	Requires developers to follow its structure and guidelines, which can lead to a more standardized way of building applications.

Table 5.1: Differences between a Library and a Framework

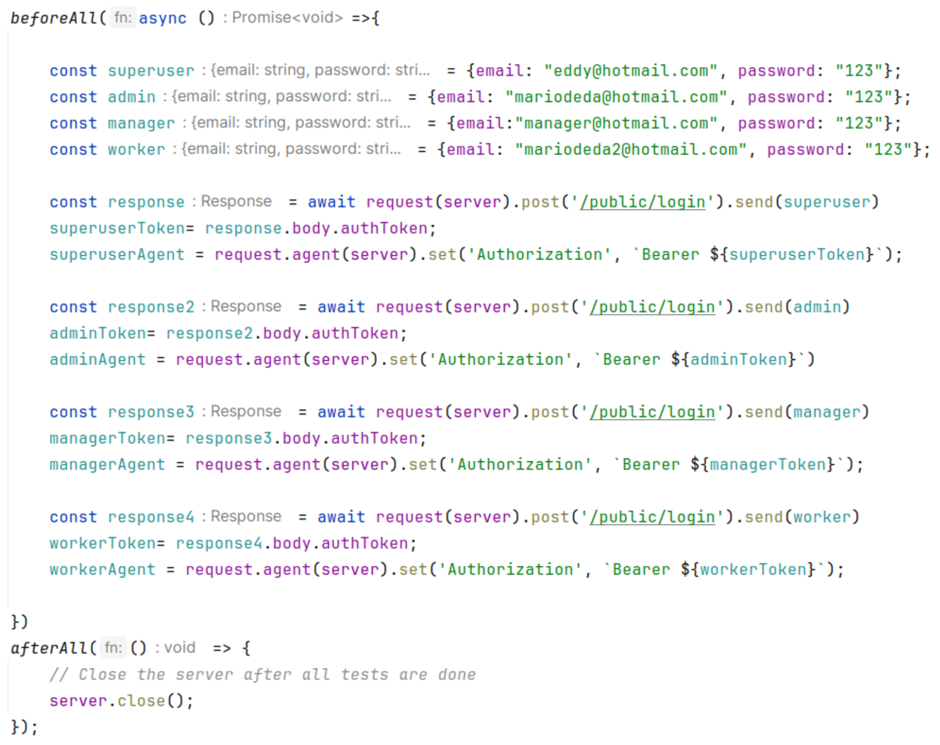
The choice between one library or another can be influenced by a few factors, such as the cadence of updates, the offered functionalities, the support from the developers, the ease of use and/or the flexibility they allow, and the same discourse hold for the frameworks too

The realization of the test suite can be done with a lot of different tools, but the language used for the program influences our choices too. In our case, using Typescript, we have a few widely used libraries and frameworks at our disposition, like Supertest, Mocha, Chai and Jest.

After a few tries, combining different ones together, Jest and Supertest looked like the optimal choice, since their Typescript support was excellent and they are overall more reliable and well documented compared to the others.

5.4.1 Supertest

Supertest is originally born as a JavaScript library used for testing APIs HTTP requests, but nowadays it has some excellent Typescript support. Supertest allows the developers to make HTTP assertions and test API endpoints efficiently [23]. It supports all standard HTTP methods, it is really easy to use and it integrates well with any test framework (Jest in this case).



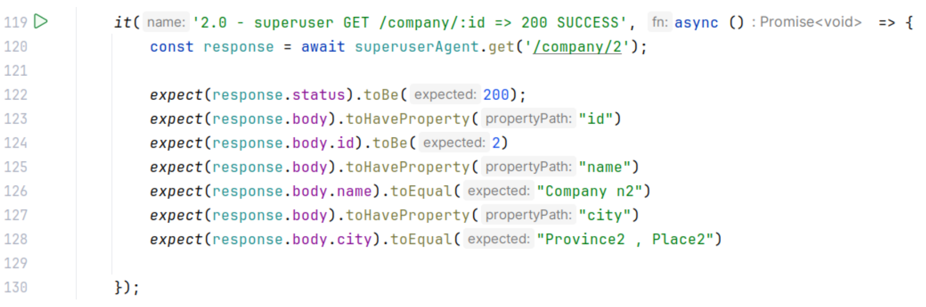
```
beforeAll( fn: async () : Promise<void> =>{

    const superuser : {email: string, password: stri...  = {email: "eddy@hotmail.com", password: "123"};
    const admin : {email: string, password: stri...  = {email: "mariodeda@hotmail.com", password: "123"};
    const manager : {email: string, password: stri...  = {email:"manager@hotmail.com", password: "123"};
    const worker : {email: string, password: stri...  = {email: "mariodeda2@hotmail.com", password: "123"};

    const response : Response  = await request(server).post('/public/login').send(superuser)
    superuserToken= response.body.authToken;
    superuserAgent = request.agent(server).set('Authorization', `Bearer ${superuserToken}`);

    const response2 : Response  = await request(server).post('/public/login').send(admin)
    adminToken= response2.body.authToken;
    adminAgent = request.agent(server).set('Authorization', `Bearer ${adminToken}`)

    const response3 : Response  = await request(server).post('/public/login').send(manager)
    managerToken= response3.body.authToken;
    managerAgent = request.agent(server).set('Authorization', `Bearer ${managerToken}`);

    const response4 : Response  = await request(server).post('/public/login').send(worker)
    workerToken= response4.body.authToken;
    workerAgent = request.agent(server).set('Authorization', `Bearer ${workerToken}`);

})
afterAll( fn: () : void  => {
    // Close the server after all tests are done
    server.close();
});
```

Figure 5.4: Supertest Agent's Setup

5.4.2 Jest

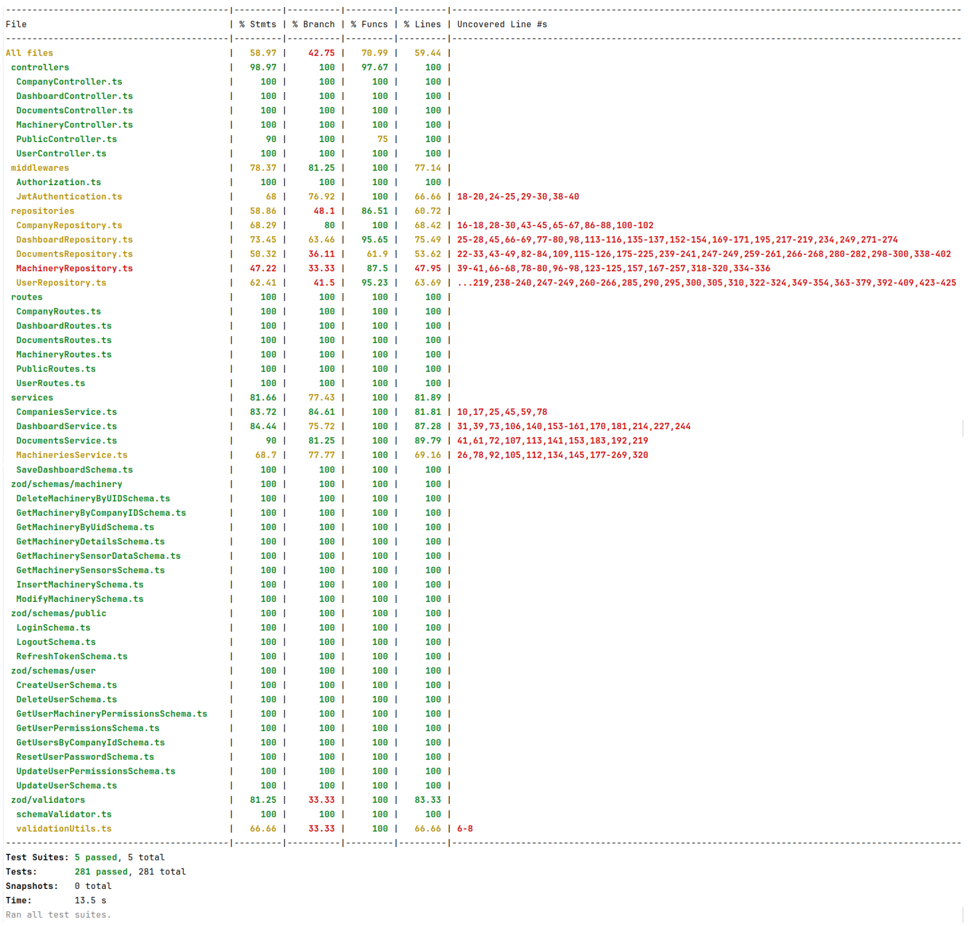
Jest is an open source testing framework for JavaScript. Initially released in 2014 by Facebook. It was designed to test large-scale JavaScript application, particularly those built in React (built by Facebook too). It has excellent support and over the years has evolved significantly, adapting to latest technologies and incorporating community feedback. It is nowadays one of the preferred testing framework in the JavaScript ecosystem as it is used and endorsed by major organizations [24].

Jest requires little to none configurations, and it works just right out of the box, without needing external dependencies. It comes with a built-in test runner and assertion library and it is also really useful for testing UI components. One of the most important features of Jest is that it includes support for code coverage reporting. After all the tests have run it generates a detailed report that highlights the area of the codebase that have been covered by tests as well as other interesting insights.



```
it( name: '2.0 - superuser GET /company/:id => 200 SUCCESS',  fn: async () : Promise<void>  => {
    const response = await superuserAgent.get('/company/2');

    expect(response.status).toBe( expected: 200);
    expect(response.body).toHaveProperty( propertyPath: "id")
    expect(response.body.id).toBe( expected: 2)
    expect(response.body).toHaveProperty( propertyPath: "name")
    expect(response.body.name).toEqual( expected: "Company n2")
    expect(response.body).toHaveProperty( propertyPath: "city")
    expect(response.body.city).toEqual( expected: "Province2 , Place2")

});
```

Figure 5.5: Jest Test Example



```
-----------------------------------------|---------|----------|---------|---------|-------------------------------------------------------------------------------------------------
File                                     | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------------------------|---------|----------|---------|---------|-------------------------------------------------------------------------------------------------
All files                                |   58.97 |    42.75 |   70.99 |   59.44 |
 controllers                             |   98.97 |      100 |   97.67 |     100 |
  CompanyController.ts                   |     100 |      100 |     100 |     100 |
  DashboardController.ts                 |     100 |      100 |     100 |     100 |
  DocumentsController.ts                 |     100 |      100 |     100 |     100 |
  MachineryController.ts                 |     100 |      100 |     100 |     100 |
  PublicController.ts                    |      90 |      100 |      75 |     100 |
  UserController.ts                      |     100 |      100 |     100 |     100 |
 middlewares                             |   78.37 |    81.25 |     100 |   77.14 |
  Authorization.ts                       |     100 |      100 |     100 |     100 |
  JwtAuthentication.ts                   |      68 |    76.92 |     100 |   66.66 | 18-20,24-25,29-30,38-40
 repositories                            |   58.86 |     48.1 |   86.51 |   60.72 |
  CompanyRepository.ts                   |   68.29 |       80 |     100 |   68.42 | 16-18,28-30,43-45,65-67,86-88,100-102
  DashboardRepository.ts                 |   73.45 |    63.46 |   95.65 |   75.49 | 25-28,45,66-69,77-80,98,113-116,135-137,152-154,169-171,195,217-219,234,249,271-274
  DocumentsRepository.ts                 |   50.32 |    36.11 |    61.9 |   53.62 | 22-33,43-49,82-84,109,115-126,175-225,239-241,247-249,259-261,266-268,280-282,298-300,338-402
  MachineryRepository.ts                 |   47.22 |    33.33 |    87.5 |   47.95 | 39-41,66-68,78-80,96-98,123-125,157,167-257,318-320,334-336
  UserRepository.ts                      |   62.41 |     41.5 |   95.23 |   63.69 | ...219,238-240,247-249,260-266,285,290,295,300,305,310,322-324,349-354,363-379,392-409,423-425
 routes                                  |     100 |      100 |     100 |     100 |
  CompanyRoutes.ts                       |     100 |      100 |     100 |     100 |
  DashboardRoutes.ts                     |     100 |      100 |     100 |     100 |
  DocumentsRoutes.ts                     |     100 |      100 |     100 |     100 |
  MachineryRoutes.ts                     |     100 |      100 |     100 |     100 |
  PublicRoutes.ts                        |     100 |      100 |     100 |     100 |
  UserRoutes.ts                          |     100 |      100 |     100 |     100 |
 services                                |   81.66 |    77.43 |     100 |   81.89 |
  CompaniesService.ts                    |   83.72 |    84.61 |     100 |   81.81 | 10,17,25,45,59,78
  DashboardService.ts                    |   84.44 |    75.72 |     100 |   87.28 | 31,39,73,106,140,153-161,170,181,214,227,244
  DocumentsService.ts                    |      90 |    81.25 |     100 |   89.79 | 41,61,72,107,113,141,153,183,192,219
  MachineriesService.ts                  |    68.7 |    77.77 |     100 |   69.16 | 26,78,92,105,112,134,145,177-269,320
  SaveDashboardSchema.ts                 |     100 |      100 |     100 |     100 |
 zod/schemas/machinery                   |     100 |      100 |     100 |     100 |
  DeleteMachineryByUIDSchema.ts          |     100 |      100 |     100 |     100 |
  GetMachineryByCompanyIDSchema.ts       |     100 |      100 |     100 |     100 |
  GetMachineryByUidSchema.ts             |     100 |      100 |     100 |     100 |
  GetMachineryDetailsSchema.ts           |     100 |      100 |     100 |     100 |
  GetMachinerySensorDataSchema.ts        |     100 |      100 |     100 |     100 |
  GetMachinerySensorsSchema.ts           |     100 |      100 |     100 |     100 |
  InsertMachinerySchema.ts               |     100 |      100 |     100 |     100 |
  ModifyMachinerySchema.ts               |     100 |      100 |     100 |     100 |
 zod/schemas/public                      |     100 |      100 |     100 |     100 |
  LoginSchema.ts                         |     100 |      100 |     100 |     100 |
  LogoutSchema.ts                        |     100 |      100 |     100 |     100 |
  RefreshTokenSchema.ts                  |     100 |      100 |     100 |     100 |
 zod/schemas/user                        |     100 |      100 |     100 |     100 |
  CreateUserSchema.ts                    |     100 |      100 |     100 |     100 |
  DeleteUserSchema.ts                    |     100 |      100 |     100 |     100 |
  GetUserMachineryPermissionsSchema.ts   |     100 |      100 |     100 |     100 |
  GetUserPermissionsSchema.ts            |     100 |      100 |     100 |     100 |
  GetUsersByCompanyIdSchema.ts           |     100 |      100 |     100 |     100 |
  ResetUserPasswordSchema.ts             |     100 |      100 |     100 |     100 |
  UpdateUserPermissionsSchema.ts         |     100 |      100 |     100 |     100 |
  UpdateUserSchema.ts                    |     100 |      100 |     100 |     100 |
 zod/validators                          |   81.25 |    33.33 |     100 |   83.33 |
  schemaValidator.ts                     |     100 |      100 |     100 |     100 |
  validationUtils.ts                     |   66.66 |    33.33 |     100 |   66.66 | 6-8
-----------------------------------------|---------|----------|---------|---------|-------------------------------------------------------------------------------------------------
Test Suites: 5 passed, 5 total
Tests:       281 passed, 281 total
Snapshots:   0 total
Time:        13.5 s
Ran all test suites.
```

Figure 5.6: Jest Code Coverage examples

The code coverage document generated by Jest has some interesting data to evaluate the effectiveness of our tests. It evaluates:

- Statements Coverage - percentage of statements executed
- Branches Coverage - percentage of control flow branches executed
- Functions Coverage - percentage of functions that have been called
- Lines Coverage - percentage of lines of code that have been executed

Jest also highlights which lines of code are covered and which are not. This precision allows the developers to efficiently identify where the problems might be, if the tests are thought out and executed correctly or if simply the code performs some useless checks or operation. If some lines of code are never executed in any circumstances, the code should probably be refactored, resulting in a cleaner and easier to read code.

We must also keep in mind that generally reaching a hundred percent coverage is usually just impossible to do, given that some lines of code are meant to be executed only in some extreme edge cases, where recreating the conditions needed would be so difficult that deviates from the testing purposes. In our case, there are some parts of the code that are meant to interact with the Timestream database, which is really hard to mock given its perks and unique functionalities. For this very peculiar characteristics, that part of the project needs to be manually tested and the reached code coverage is just around 80 percent.

As a reference, ninety percent coverage is generally accepted as a real good statistic, and only a really small portion of projects need to go higher than that, like mission critical code run on advanced and expensive machinery, or where faulty code execution could have really huge impact on life of human beings.

Even if the coverage statistic is not that impressive, this is still a first iteration of the test suite, which wasn't present at the start of the project, consequently meaning that the early written prototype code wasn't meant to be tested. From now on every new project feature will be added to the test suite, providing instant feedback on functionality and code quality, that will ultimately lead to better code.

5.5 ESLint

ESLint is an open-source static code analysis tool. "Static" means that it analyzes the code without actually running it, meaning that performances of the code are not in the scope of this tool.

Doing an analogy with writing a book, ESLint focuses on the correct grammar and syntax of the phrases, but it does not care about the contents of the story.

Created in 2013 by Nicholas Zakas it has become an indispensable part of the JavaScript/Typescript ecosystem.

ESLint's main function is to grant code consistency and ensure that code best practices are followed. In big projects, having two different files coded in completely different ways may be an issue, and having some guidelines to follow to make the whole project consistent makes things easier, for the people that are actually working on it and even for the people that will work on it in the future. ESLint job is to enforce those guidelines. Consistent code is easier for team members to read and understand, improving cooperation and reducing the cognitive load on developers.

One of the best features of ESLint is its extensibility and customizability, giving the opportunity to every team to chose the rules they prefer and being able to adapt to every kind of project since different projects may have different programming languages and may require following different rules of formatting. ESLint supports ECMAScript 6 (ES6) and it also supports jsx files (jsx is a syntax extension for React).

The ESLint configuration files can be written in different formats, from JSON to YAML files, but they all serve the same purpose: they specify what rules needs to be enforced.

Most of moderns IDEs, such as Visual Studio Code or Jetbrains products have a plugin or some kind of built in function that supports ESLint, providing developers with real time feedback while working on the code.

ESLint is also used in most of CI/CD systems, to enforce on the repository code consistency and catch any eventual errors before they are merged into the main codebase.

To sum it up, ESLint encourages all the best practices and modern coding techniques and enforces those coding standards in projects, enhancing code safety, readability and maintainability, ultimately leading to better software products [25].

5.6 Test Containers

Traditionally testing software means creating mock services, which can be error-prone, resource-intensive, and they also require a huge effort to setup and populate.

Originally designed for Java, Testcontainers is an open source framework that provides throwaway, lightweight instances of Docker containers, that not only solves this issue but also provides a consistent and reliable testing environment.

It enables to create containers with testing purposes directly within the code, giving developers the freedom to effortlessly generate an isolated and reproducible environment. The power of containerizing stuff means that we can use containerized instances of databases to test our data access layer code, or we can use containerized web browsers to run automated UI tests.

Testcontainers are also especially useful for integration testing, since their purpose is to verify the correct functioning of different systems, being able to quickly generate and destroy a service speeds up the process by a lot. More importantly, when a test fails, we can be confident that the test failed because of some bug or error within the code and not due to environmental discrepancies, and the fact that containers are generated on the fly through code, they can be easily shared and replicated across different machines and CI/CD pipelines.

saving time and effort of setting up a mock db isn't a negligible perk, since in our case we were able to populate the database with the same SQL file we use for testing the app locally, basically using the same instance of the Postgres DB.

The basic usage of Testcontainers is pretty straightforward, but it is possible to create custom containers, configuring all kinds of settings to suit every testing need that a developer might have, allowing for instance to simulate complex microservices architectures. All kinds of languages and databases are supported, making it a tool usable in every project [26].

Another important advantage is the seamless integration into CI/CD pipelines, since the environment is standardized they are an optimal choice for this kinds of practices that will be discussed here below.

Chapter 6

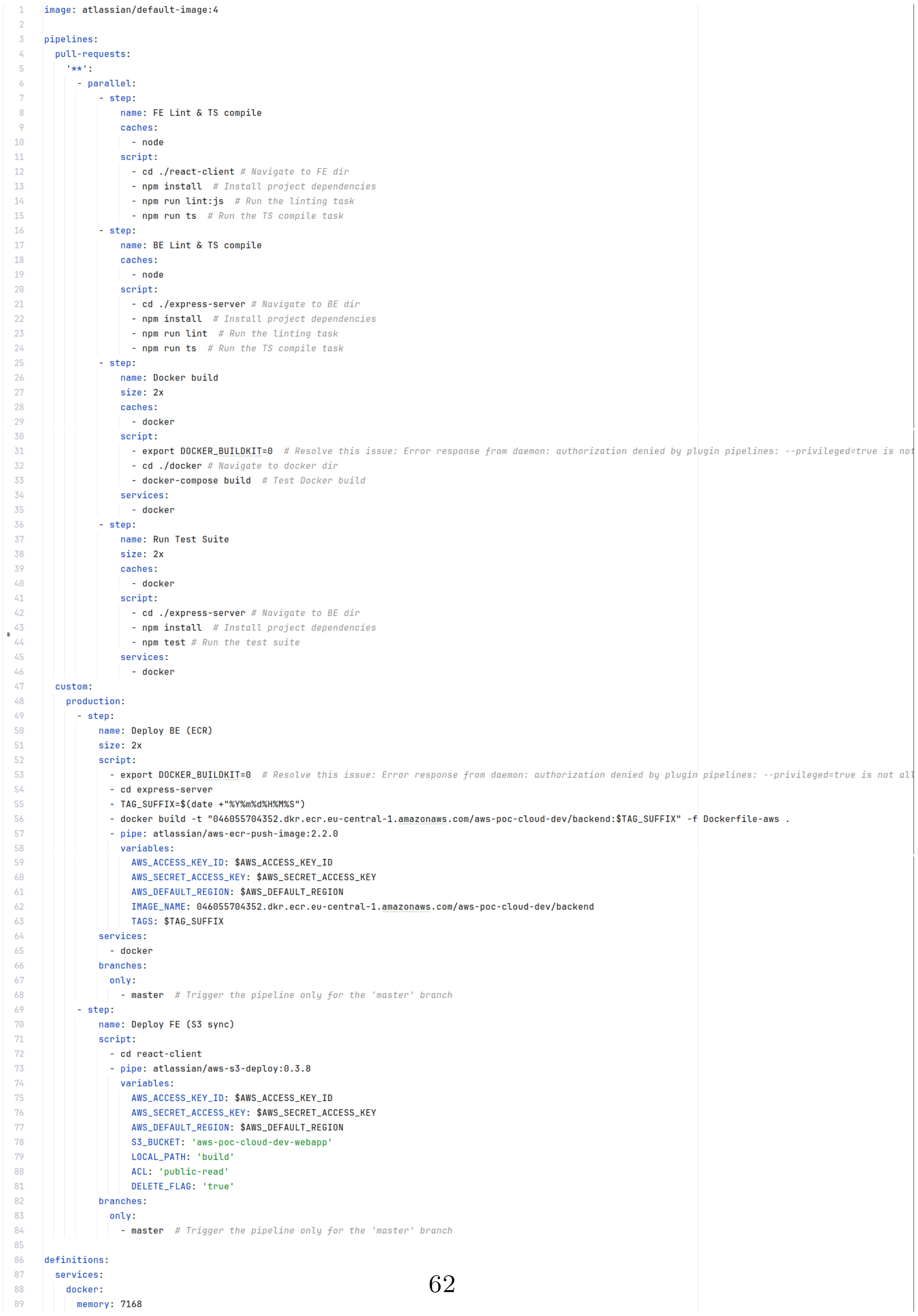
Conclusions and results

This chapter will be focused on the results obtained by all the functionalities and the libraries added to the system, briefly presenting and resuming all the advantages given by this new approach.

6.1 Bitbucket Pipeline

CI/CD (Continuous Integration / Continuous Delivery) is a common and standard practice nowadays, since building, testing and deploying code are the very basics of every modern software product, being able to control and supervise these processes directly through the repository hub of your choice is really useful and speeds up the developing process.

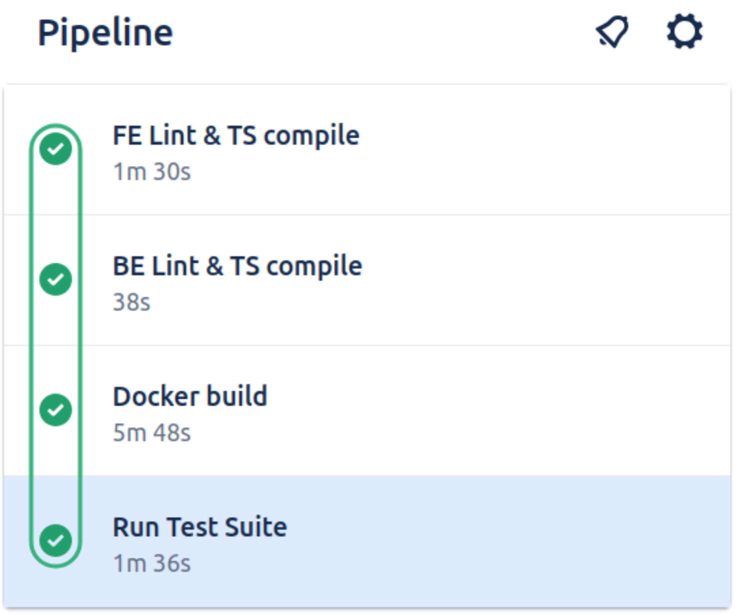
Bitbucket pipelines are a CI/CD service directly built into Bitbucket Cloud, that allows developers that store code on Bitbucket server to automatically build test and deploy their projects through a simple YAML file. Customizing the YAML file the developers can chose to run certain routines that will define the series of tests and checks that will be performed whenever a certain trigger happens (like when a pull request is created). Once the code is stored in the Bitbucket repository, the system will look for the YAML file and will execute the chosen tests in parallel, speeding up the verification time.



```yaml
image: atlassian/default-image:4

pipelines:
  pull-requests:
    '**':
      - parallel:
          - step:
              name: FE Lint & TS compile
              caches:
                - node
              script:
                - cd ./react-client # Navigate to FE dir
                - npm install  # Install project dependencies
                - npm run lint:js  # Run the linting task
                - npm run ts  # Run the TS compile task
          - step:
              name: BE Lint & TS compile
              caches:
                - node
              script:
                - cd ./express-server # Navigate to BE dir
                - npm install  # Install project dependencies
                - npm run lint  # Run the linting task
                - npm run ts  # Run the TS compile task
          - step:
              name: Docker build
              size: 2x
              caches:
                - docker
              script:
                - export DOCKER_BUILDKIT=0  # Resolve this issue: Error response from daemon: authorization denied by plugin pipelines: --privileged=true is not
                - cd ./docker # Navigate to docker dir
                - docker-compose build  # Test Docker build
              services:
                - docker
          - step:
              name: Run Test Suite
              size: 2x
              caches:
                - docker
              script:
                - cd ./express-server # Navigate to BE dir
                - npm install  # Install project dependencies
                - npm test # Run the test suite
              services:
                - docker
  custom:
    production:
      - step:
          name: Deploy BE (ECR)
          size: 2x
          script:
            - export DOCKER_BUILDKIT=0  # Resolve this issue: Error response from daemon: authorization denied by plugin pipelines: --privileged=true is not all
            - cd express-server
            - TAG_SUFFIX=$(date +"%Y%m%d%H%M%S")
            - docker build -t "046055704352.dkr.ecr.eu-central-1.amazonaws.com/aws-poc-cloud-dev/backend:$TAG_SUFFIX" -f Dockerfile-aws .
            - pipe: atlassian/aws-ecr-push-image:2.2.0
              variables:
                AWS_ACCESS_KEY_ID: $AWS_ACCESS_KEY_ID
                AWS_SECRET_ACCESS_KEY: $AWS_SECRET_ACCESS_KEY
                AWS_DEFAULT_REGION: $AWS_DEFAULT_REGION
                IMAGE_NAME: 046055704352.dkr.ecr.eu-central-1.amazonaws.com/aws-poc-cloud-dev/backend
                TAGS: $TAG_SUFFIX
          services:
            - docker
          branches:
            only:
              - master  # Trigger the pipeline only for the 'master' branch
      - step:
          name: Deploy FE (S3 sync)
          script:
            - cd react-client
            - pipe: atlassian/aws-s3-deploy:0.3.8
              variables:
                AWS_ACCESS_KEY_ID: $AWS_ACCESS_KEY_ID
                AWS_SECRET_ACCESS_KEY: $AWS_SECRET_ACCESS_KEY
                AWS_DEFAULT_REGION: $AWS_DEFAULT_REGION
                S3_BUCKET: 'aws-poc-cloud-dev-webapp'
                LOCAL_PATH: 'build'
                ACL: 'public-read'
                DELETE_FLAG: 'true'
          branches:
            only:
              - master  # Trigger the pipeline only for the 'master' branch

definitions:
  services:
    docker:
      memory: 7168
```

Figure 6.1: example of .YAML file

This approach enhances not only the potential scalability of the code, but also allows multiple teams of developers to work together without worrying too much to break other's people code, providing visibility into the build and deploy process, and significantly improving collaboration. In the code snippet above, we run four main test: the first two check that both the FE and the BE correctly compile, while also checking that the ESLint is satisfied, the third performs some Docker checks, making sure that the containers are created properly and that the code builds as intended, while the fourth one runs the whole test suite we discussed in the previous chapter, automatically creating and successively deleting the Testcontainers needed for it to run, further enforcing that everything works properly [27].


Pipeline

FE Lint & TS compile
1m 30s

BE Lint & TS compile
38s

Docker build
5m 48s

Run Test Suite
1m 36s

Figure 6.2: GUI of Bitbucket Pipeline

The complete customizability of the pipelines, allows developers to perform any kind of checks and tests that they prefer, and overtime while the size and complexity of a project grows, new tests and checks can be added, suiting every development's team needs.

6.2 BE refactoring

As we mentioned multiple times previously, the addition of a test suite makes some work such as the refactoring, possible. Given our BE design, and we are now in a condition of being able to test all of our code's main functionalities with a few clicks, we can now polish and clean our Service layer, given that after a few iterations of our app, it got pretty messy and unorganized.

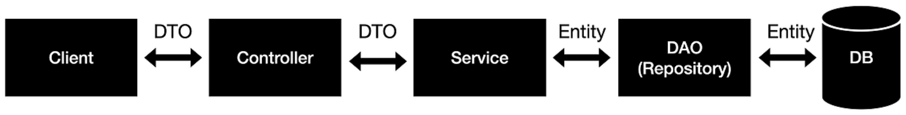


Figure 6.3: BE schema design

The controller layer should be the one in charge of deciding if a certain user or request can access the service layer or not. The repository layer must be the only one able to access the DB and the service layer is the one that has to manage the communication between the two, eventually filtering results or manipulating data obtained from the repository.

The accessibility to data should be one of our primary concerns, and for this reason, all the security and authority checks performed in the service layer will be moved a layer up, to the Controller layer, to keep further away users without the required permissions and authorizations. Here is a table that will summarize this concept:

This approach leads to the creation of new functions in the service layer, as example : verifyPermissionAndOwnership().

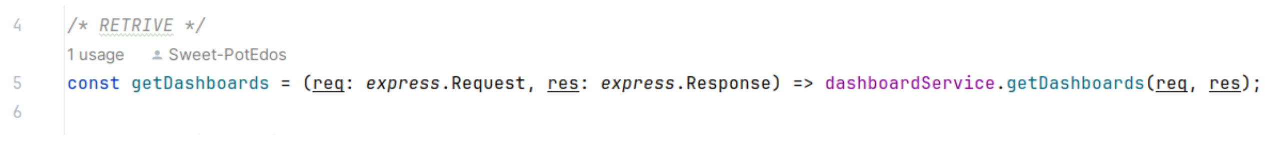
The creation of this function makes the code cleaner and avoids code repetition, also specifically checking both permission to access some machineries and the ownership of those in the service layer, along with user roles and authorizations, brought confusion and unnecessary complexity.

This means that every route that needs to perform these kinds of checks, will call the controller, that will call the "verifyPermissionAndOwnership" services, making the service layer accessible only to the ones that truly have the rights to access them and improving further our security and system design.

	Function	Errors
Routes	API access point, validates input with ZOD, authenticates user with JWT token strategy	403 UNAUTHORIZED, 400 BAD BODY
Controller	Checks that the logged user has all needed permissions and authorization for accessing the service	403 UNAUTHORIZED
Service	Calls repository functions, filters and manages the results	404 NOT FOUND, 500 ERROR, 200 OK
Repository	Calls the DB to access data	x

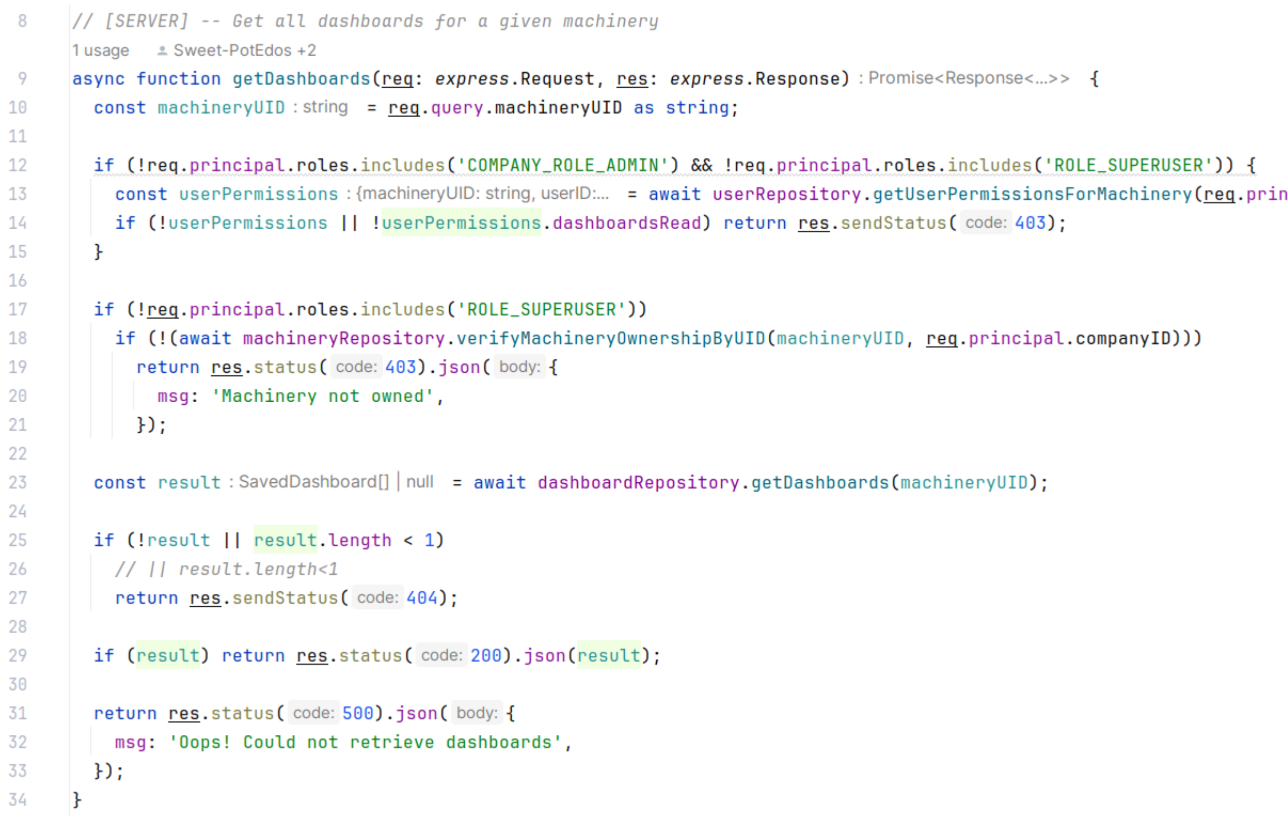
Table 6.1: Summary of BE's architecture

Another benefit of this approach is the fact that errors "403 UNAUTHORIZED" will all be thrown by the controller layer, giving consistency to our project, since if we ever get that kind of error we are sure that the controller layer is where it was generated, also at the same time we remove them from the service layer, leaving it only with 200, 404 or 500 errors to manage.



```
/* RETRIVE */
1 usage   Sweet-PotEdos
const getDashboards = (req: express.Request, res: express.Response) => dashboardService.getDashboards(req, res);
```

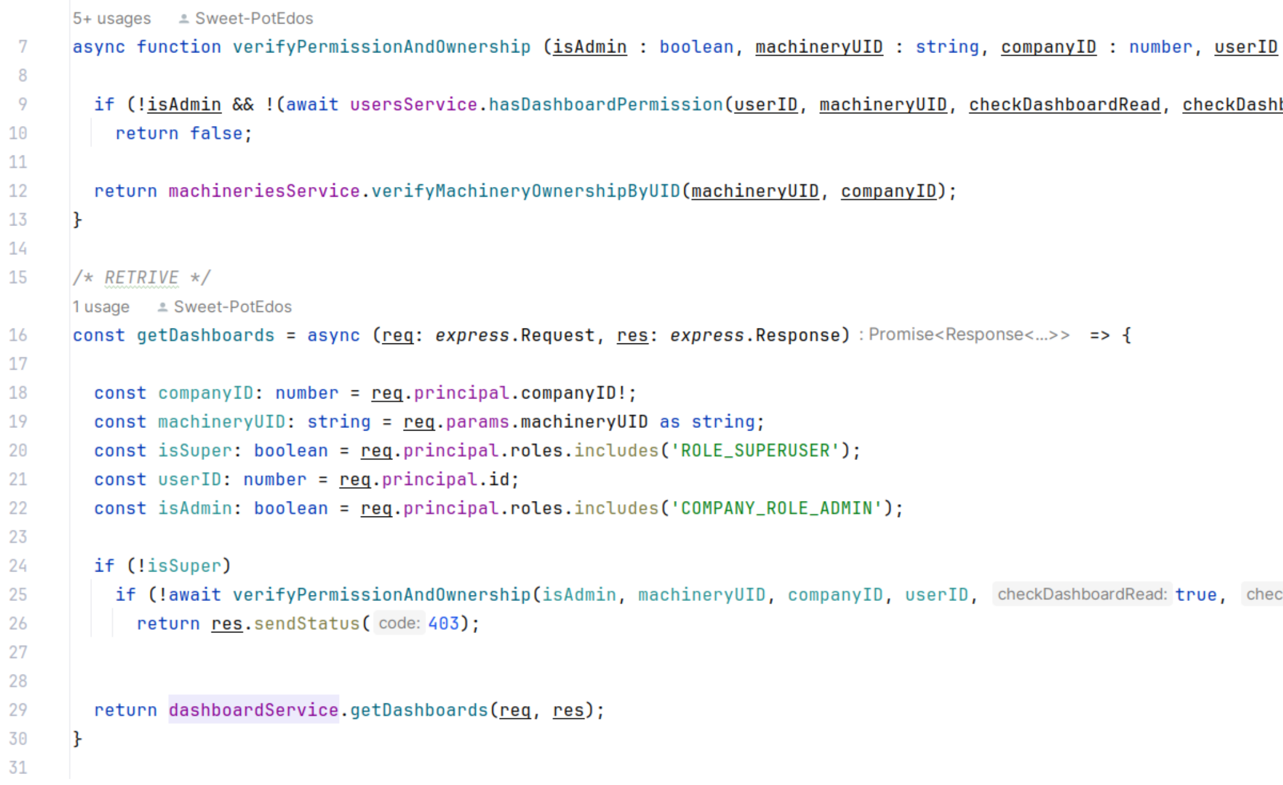
Figure 6.4: BE controller before refactoring



```
// [SERVER] -- Get all dashboards for a given machinery
1 usage   Sweet-PotEdos +2
async function getDashboards(req: express.Request, res: express.Response) : Promise<Response<...>>  {
  const machineryUID : string  = req.query.machineryUID as string;

  if (!req.principal.roles.includes('COMPANY_ROLE_ADMIN') && !req.principal.roles.includes('ROLE_SUPERUSER')) {
    const userPermissions : {machineryUID: string, userID:...  = await userRepository.getUserPermissionsForMachinery(req.prin
    if (!userPermissions || !userPermissions.dashboardsRead) return res.sendStatus( code: 403);
  }

  if (!req.principal.roles.includes('ROLE_SUPERUSER'))
    if (!(await machineryRepository.verifyMachineryOwnershipByUID(machineryUID, req.principal.companyID)))
      return res.status( code: 403).json( body: {
        msg: 'Machinery not owned',
      });

  const result : SavedDashboard[] | null  = await dashboardRepository.getDashboards(machineryUID);

  if (!result || result.length < 1)
    // || result.length<1
    return res.sendStatus( code: 404);

  if (result) return res.status( code: 200).json(result);

  return res.status( code: 500).json( body: {
    msg: 'Oops! Could not retrieve dashboards',
  });
}
```

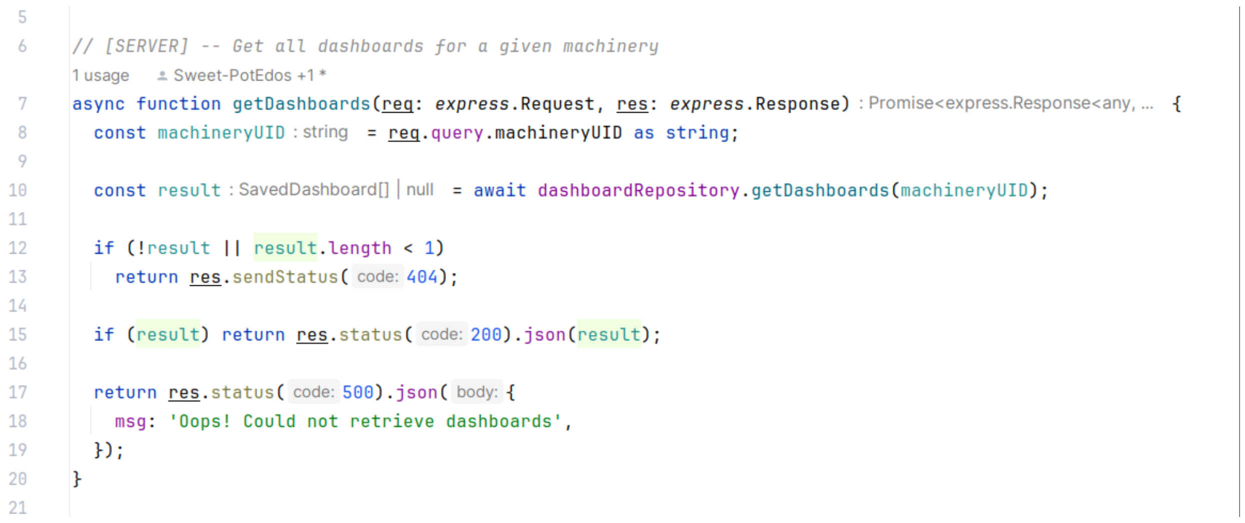
Figure 6.5: BE service before refactoring

As proven by the images, the controller layer wasn't really used before, becoming an unnecessary layer in our code, while the service layer had way too much checks and controls performed, making single functions long, hard to understand and debug properly.



```
5+ usages   Sweet-PotEdos
async function verifyPermissionAndOwnership (isAdmin : boolean, machineryUID : string, companyID : number, userID

  if (!isAdmin && !(await usersService.hasDashboardPermission(userID, machineryUID, checkDashboardRead, checkDashb
    return false;

  return machineriesService.verifyMachineryOwnershipByUID(machineryUID, companyID);
}

/* RETRIVE */
1 usage   Sweet-PotEdos
const getDashboards = async (req: express.Request, res: express.Response) : Promise<Response<...>>  => {

  const companyID: number = req.principal.companyID!;
  const machineryUID: string = req.params.machineryUID as string;
  const isSuper: boolean = req.principal.roles.includes('ROLE_SUPERUSER');
  const userID: number = req.principal.id;
  const isAdmin: boolean = req.principal.roles.includes('COMPANY_ROLE_ADMIN');

  if (!isSuper)
    if (!await verifyPermissionAndOwnership(isAdmin, machineryUID, companyID, userID,  checkDashboardRead: true,  chec
      return res.sendStatus( code: 403);


  return dashboardService.getDashboards(req, res);
}
```

Figure 6.6: BE controller after refactoring



```
// [SERVER] -- Get all dashboards for a given machinery
1 usage   Sweet-PotEdos +1 *
async function getDashboards(req: express.Request, res: express.Response) : Promise<express.Response<any, ...  {
  const machineryUID : string  = req.query.machineryUID as string;

  const result : SavedDashboard[] | null  = await dashboardRepository.getDashboards(machineryUID);

  if (!result || result.length < 1)
    return res.sendStatus( code: 404);

  if (result) return res.status( code: 200).json(result);

  return res.status( code: 500).json( body: {
    msg: 'Oops! Could not retrieve dashboards',
  });
}
```

Figure 6.7: BE service after refactoring

This refactoring makes the controller layer useful again, while also making all the checks in a single place and improving all the code quality factors like coverage of the tests and readability of the overall code.

This process, is not only long and time consuming, but also it does not change the functionalities of our app in any way. The upsides are worth it though, the project will greatly benefit of this refactoring, since ultimately the whole BE will be:

1. Safer - the inappropriate or unauthorized calls will be blocked earlier at the Controller layer and not later on
2. Cleaner - the Controller layer was massively underutilized and the Service layer was filled with security checks that made it all difficult to understand and messy
3. Readable - people that will work on this project will find structured and organized checks, greatly improving readability

All of this work was possible thanks to the test suite, because there wouldn't be any other way of verifying that the whole system is still working as intended after massively changing most of the BE files, and even if we were an exceptional programmer and we were sure of our code, we would have no way of granting that every function in the system still worked as intended. Also at the same time, it slightly improved the project's code coverage, removing many lines of code that had no way of being executed. Repeating this process on more components of the BE will ultimately lead to a well designed and maintainable codebase.

6.3 Cloud deployment vs on premises deployment

Deploying applications on premises means utilizing the company's own infrastructure rather than the cloud.

The trend of these last years is the opposite of the on premises deployment, since nowadays the cloud infrastructure is growing constantly year after year, nonetheless it still is an option that many organizations should consider.

The deployment on premises offers some unique features, having the complete control over the hardware software and data allows for extreme customization and flexibility, tailoring systems to the exact system requirements. It has some security benefits too, owning the hardware and having physical access to services greatly reduces the risk of data breaches from third party providers, also can provide reduced latency with faster data access since the data is stored locally and does not have to travel over the internet, and the performance are more consistent too because they are not subject anymore to the internet speed or the cloud service provider limitations.

Of course this approach has some cons too, otherwise the could hype over this years wouldn't be justified. The initial investment for hardware and software isn't negligible and the needed system maintenance has a cost too, not to mention the eventual scalability problem that may arise and lead to buy and set up additional hardware, compared to cloud services that can just adjust the resources needed dynamically. Everyone always hope that nothing bad happens, but accidents may happen and disaster recovery functions, that come out of the box in cloud services, may be costly to be properly setup and manage.

All of this discussion holds, but the company and the system that we need to host may influence our deployment decision, the point is that nowadays the decision looks obvious, and everybody wants to jump into the cloud trend, but many systems and companies don't really need the cloud for their services.

Cloud services can become pretty pricey pretty fast, and depending on system requirements, a company maybe doesn't even need all the benefits offered by the provider. Organizations need to carefully assess their requirements, resources, and long-term strategy when deciding between on-premises and cloud deployments.

6.4 Conclusions and future works

The application now offers more features and functionalities: users will be able to select the template they prefer, simplifying the dashboard creation process, and AROL itself has way more management features, making it able to control and manage every aspect of the system with ease and efficiency.

The addition of libraries like Drizzle ORM and ZOD validation grants respectively flexibility and security, decoupling the code from the database underneath, improving the scalability and adaptability of the project.

The whole code infrastructure is cleaner and safer, the code is now well documented and the introduction of the test suite and the integration with the Bitbucket pipelines not only grants a reliable development process but also made a huge refactoring possible, making the architecture simpler and more defined.

This all together made the application resilient and sustainable in the long run which was exactly our objective right from the start. The number of functionalities added isn't very high but most of the work was dedicated on enhancing and improving not functionalities but readability and maintainability of the code. Refactoring by definition does not change anything functionally, but it has insane value over time especially for the ones that will work on this project after me, and the same holds for the produced documentation.

In the future of this project, developers will need to dedicate more time to test and document new features, but if the test are properly designed and they pass, it ultimately means that the error in the code can be only logic ones, as the functionality is granted by all the numerous CI/CD instruments implemented.

Another important addiction can be the actual implementation of the notification system, only proposed and designed in this thesis but not actually implemented, which will further improve customer support and satisfaction.

Acknowledgements

I would like to express my sincere gratitude to all those who supported and guided me throughout my university career and the development of this thesis.

My heartfelt thanks go to my family and friends for their unwavering support and encouragement throughout my academic journey. Their belief in me provided the motivation I needed to complete this thesis.

Lastly, I am grateful to my colleagues and fellow researchers for their collaborative spirit and stimulating discussions, which greatly enriched this research.

Thank you all for your contributions and support.

Bibliography

[1] *AROL group website.* Accessed: 2024-07-09. URL: https://www.arol.com/it/ (cit. on p. 6).

[2] *IoT architecture.* Accessed: 2024-07-09. URL: https://dgtlinfra.com/internet-of-things-iot-architecture/ (cit. on p. 7).

[3] *Equatorque image.* Accessed: 2024-07-09. URL: https://www.arol.com/it/macchine-tappatrici-per-bevande/tappatrici-per-tappo-in-plastica-pre-filettato-flat/288-equatorque-pk-it (cit. on p. 8).

[4] *MQTT protocol.* Accessed: 2024-07-09. URL: https://www.twilio.com/en-us/blog/what-is-mqtt (cit. on p. 10).

[5] *VS Code website.* Accessed: 2024-07-09. URL: https://code.visualstudio.com/docs (cit. on p. 14).

[6] *IntelliJ website.* Accessed: 2024-07-09. URL: https://www.jetbrains.com/idea/ (cit. on p. 15).

[7] *Javascript wikipedia page.* Accessed: 2024-07-09. URL: https://it.wikipedia.org/wiki/JavaScript (cit. on p. 16).

[8] *Typescript website.* Accessed: 2024-07-09. URL: https://www.typescriptlang.org/docs/ (cit. on p. 17).

[9] *DOM and VDOM.* Accessed: 2024-07-09. URL: https://www.babbel.com/en/magazine/build-your-own-react-episode-2 (cit. on p. 21).

[10] *React website.* Accessed: 2024-07-09. URL: https://react.dev/ (cit. on p. 22).

[11] *Node.js docs.* Accessed: 2024-07-09. URL: https://nodejs.org/docs/latest/api/ (cit. on p. 23).

[12] *ExpressJS website.* Accessed: 2024-07-09. URL: https://expressjs.com/ (cit. on p. 24).

[13] *PostgreSQL website*. Accessed: 2024-07-09. URL: `https://www.postgresql.org/docs/` (cit. on p. 25).

[14] *AWS Timestream website*. Accessed: 2024-07-09. URL: `https://aws.amazon.com/it/timestream/` (cit. on p. 26).

[15] *AWSS3 website*. Accessed: 2024-07-09. URL: `https://aws.amazon.com/it/s3/` (cit. on p. 26).

[16] *Docker website*. Accessed: 2024-07-09. URL: `https://docs.docker.com/` (cit. on p. 27).

[17] *Localstack website*. Accessed: 2024-07-09. URL: `https://www.localstack.cloud/` (cit. on p. 31).

[18] *JWT Token Structure*. Accessed: 2024-07-09. URL: `https://www.miniorange.com/blog/what-is-jwt-json-web-token-how-does-jwt-authentication-work/` (cit. on p. 33).

[19] *JWT token schema*. Accessed: 2024-07-09. URL: `https://auth0.com/learn/json-web-tokens` (cit. on p. 35).

[20] *JWT refresh schema*. Accessed: 2024-07-09. URL: `https://www.bezkoder.com/spring-security-refresh-token/` (cit. on p. 35).

[21] *Drizzle website*. Accessed: 2024-07-09. URL: `https://orm.drizzle.team/docs/overview` (cit. on p. 50).

[22] *ZOD website*. Accessed: 2024-07-09. URL: `https://zod.dev/` (cit. on p. 52).

[23] *Supertest website*. Accessed: 2024-07-09. URL: `https://www.npmjs.com/package/supertest` (cit. on p. 55).

[24] *Jest website*. Accessed: 2024-07-09. URL: `https://jestjs.io/` (cit. on p. 56).

[25] *ESLint website*. Accessed: 2024-07-09. URL: `https://eslint.org/` (cit. on p. 59).

[26] *Testcontainers website*. Accessed: 2024-07-09. URL: `https://testcontainers.com/` (cit. on p. 60).

[27] *Altissan website*. Accessed: 2024-07-09. URL: `https://www.atlassian.com/it/software/bitbucket/features/pipelines` (cit. on p. 63).