
UNIVERSITÀ
DI SIENA
1240

Department of
Information Engineering and Mathematics

Corso di laurea in
INGEGNERIA GESTIONALE

# Implementation of Go kit and jsreport Microservices for an Industrial IoT Start-up

**Supervisor:**
**Prof. Marco Maggini**

**Candidate:**
**Francesco Papini**
**Mat. 092351**

Academic Year 2023-2024

A mia nonna Giuliana,
che avrebbe voluto festeggiare
questo importante traguardo

# Contents

# List of Figures

# Chapter 1

# Introduction

Animals evolve gradually over time, striving to adapt and survive in the face of the impact of humans. Meanwhile, humans are evolving technology exponentially over time, in order to enhance their quality of life. To do this, we require more and more energy. The greatest challenge of our time lies in effecting a sustainable energy transition, trying to avoid a sixth mass extinction. Beyond improving energy sources, we must also optimize energy utilization, a task facilitated by the Industrial Internet of Things (IIoT).

## 1.1 Industrial Internet of Things

Industrial Internet of Things refers to connecting devices within industrial settings integrated with advanced analytics. In simpler terms, sensors, smart actuators, and other devices are deployed for the collection, monitoring, and analysis of data emanating from industrial processes and equipment in real-time. This will improve operational efficiency, saving costs and allowing predictive maintenance through data usage.

Zerynth is an IIoT start-up based in Pisa, where I had the pleasure of doing

my internship during 2023. Zerynth enables companies to streamline production processes and increase the value of connected industrial products. Through a plug-and-play IoT and AI platform, Zerynth connects any industrial machine, allowing for a complete industrial 4.0 transformation quickly, flexibly and securely.

## 1.2 Goals of the project

I worked in the cloud team, where I was in charge of creating the report generation system, consisting of:

- **reports generator microservice**, which has the task of generating the custom report based on the custom template and the data received as input; the report is saved in a database connected to this service
- **reports manager microservice**, which is the service able to manage client requests, being able to perform the classic CRUD (Create, Read, Update, and Delete) operations. The information about reports, including the ids, are stored in a specific database.
- **standalone web app**, able to create a new report, fill in a form, delete or modify it. It must also be able to list all reports, search them by name or filter them by other criteria and finally display them.

This web app will then be integrated into the Zerynth cloud.

# Chapter 2

# Architecture

## 2.1 Monolithic Architecture

The first choice one faces when creating a new software architecture is between monolithic or microservices. The choice is mainly based on the size of the project envisioned and the development team. But it's difficult to predict the size of the final project since often some features are conceived and added along the way. A common choice, made when you have a small team and the need to quickly deploy a functioning product, is to choose the monolithic architecture. In fact, it is very good for the development of an MVP (Minimum Viable Product).

The monolithic architecture is basically a unique complex codebase that generally uses a single framework. A good way to try to keep it maintainable and readable is to divide the monolith into some modules, each responsible for a specific feature and designed to be as much independent as possible. This could also simplify a gradual migration to the microservice architecture in the future.

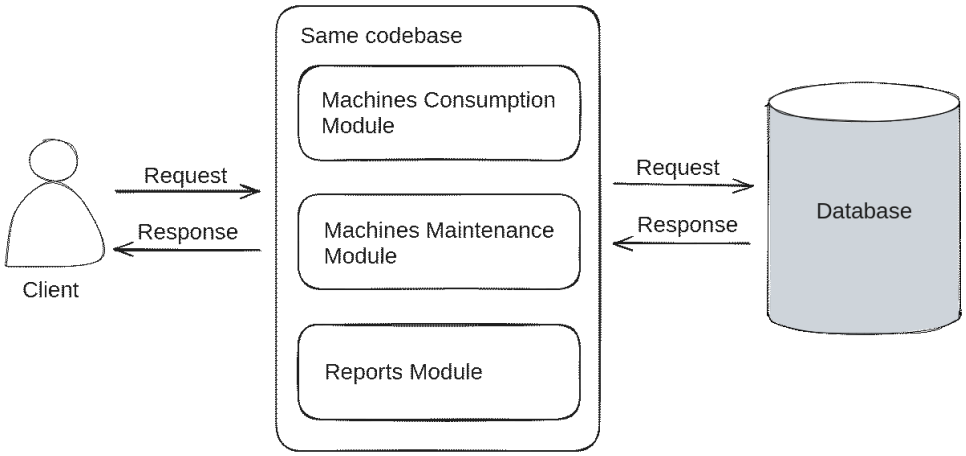


Figure 2.1: Basic monolithic architecture for an IIoT project

The great advantages in the ease of development and testing, as well as the ease of deployment and scaling, decrease as the size of the project and team grows, because it becomes difficult to manage a single very large codebase. More and more developers working on the same code results in a higher likelihood of merge conflicts. Additionally, the need to scale and test code more specifically and precisely increases. This is where the microservice architecture comes in.

## 2.2 Microservice Architecture

The microservice architecture is used to overcome the problems that arise as software projects grow in size. This structure is so called because it consists of several small services communicating with each other via an API (Application Programming Interface). Each service is designed to be relatively small, in order to be easy to develop for a small team, which can choose the framework best suited to the microservice task. For example, in the case of this project with Zerynth, we chose:

- **go kit** for the implementation of the **reports manager service**, because we needed a service that was lean and easy to develop, and being widely used

within Zerynth made it easier to be maintained in the future.

- **jsreport** for the **reports generator service** that allows you to create custom templates and then generate reports via a REST API.

These services that create an interconnected network must be able to interact with a client. Usually, for simplicity an **API gateway** is used as an intermediary, that is a server that routes requests from the client to the required microservice. It can also have many other features, among which:

- **caching**, the API gateway can cache responses to improve performance and reduce the load on the other services;
- **rate limiting**, it can be enforced to prevent the abuse of resource usage;
- **security**, it can handle authentication and authorisation, ensuring that only legitimate requests reach the microservices.

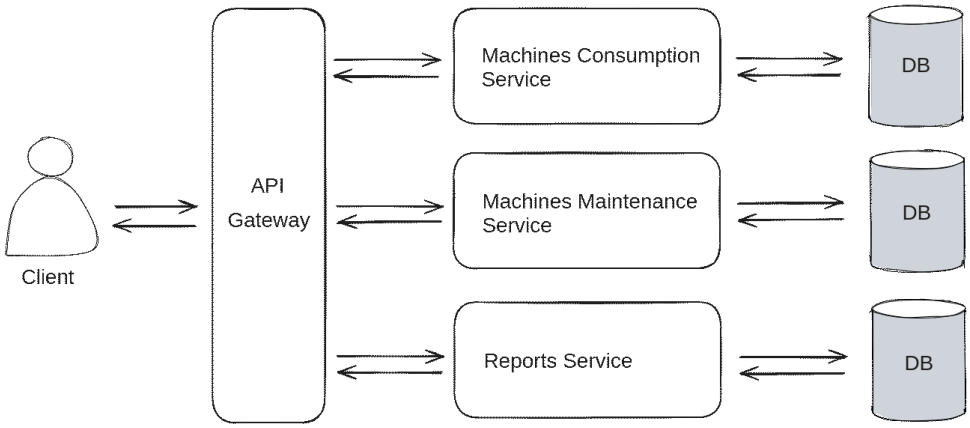


Figure 2.2: Basic microservices architecture for an IIoT project

### 2.2.1 Synchronous microservices

The term *Synchronous microservices* refers to an architecture where communication between services happens in a synchronous manner. This means that when the first service makes a request to the second one, service 1 waits for a response from service 2 before it can continue its execution.

Synchronous communication is widespread, using protocols such as gRPC and REST for scenarios that require immediate feedback. Services tend to be coupled; service 1 expects the immediate response of service 2. This results in dependencies, where changes in service 2 can directly affect how service 1 will work. Since it is simple to interact with execution flow that is conducted in real-time, the modules can be debugged easily compared to asynchronous communication, where tracing the message flow is tricky.

The pros of synchronous microservices is their simplicity, ability to receive immediate feedback, and more straightforward error handling. Errors and exceptions are easier to deal with in real-time. The downside is the increased latency, which can create scalability issues and reduce fault tolerance. For example, if service 2 goes down or experiences high latency, it will directly affect both the performance and availability of service 1. Synchronous microservices are a suitable design option in some scenarios where there is an immediate answer and the communication pattern is straight. Still, these have trade-offs with performance and scalability.

### 2.2.2 Asynchronous microservices

Unlike the architecture analyzed above, non-blocking operations are performed in this case. This simply implies that, at the time when a service requests another one, it doesn't wait for an immediate response but stays running so that it can continue working and can take on another task.

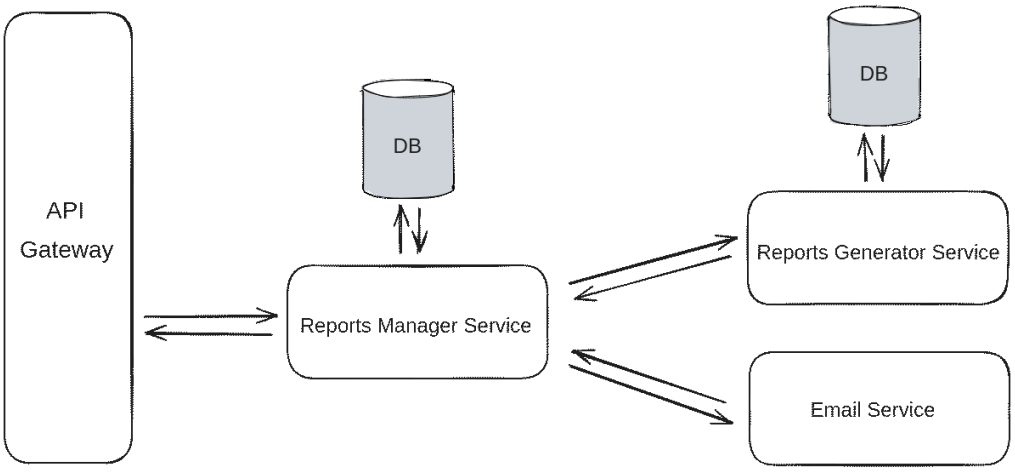


Figure 2.3: Basic sync reports microservices architecture

The pros of asynchronous microservices include optimized performance and scalability, because services can go on doing their works without pausing to wait for responses — a reason why it enables the better use of resources and scalability. The system can tolerate failures more gracefully. If a service is down, messages can be queued for later processing. The most common form of asynchronous communication leverages message queues or event streams, such as Apache Kafka or RabbitMQ, to decouple services. Messages are sent to a queue or topic; they remain there until picked up by the receiving service.

In the reports system design, the idea was to send an email to the customer once the report has been generated correctly. To do this, I used the existing Zerynth email service, which works asynchronously, such that to send a new message it is sufficient to generate a new *send-email* event for the message broker with the necessary information. In RabbitMQ, the message broker that we used, a producer is an application that publishes messages to the broker. An exchange agent receives a message from a producer and forwards it to the queues according to routing rules. There are direct, topic, and headers exchanges, all with different routing logic. A queue is a buffer in which messages are stored until a consumer consumes them.

The queues are associated with the exchanges according to binding rules. Several producers can send messages to a queue, and several consumers can retrieve messages from a queue.

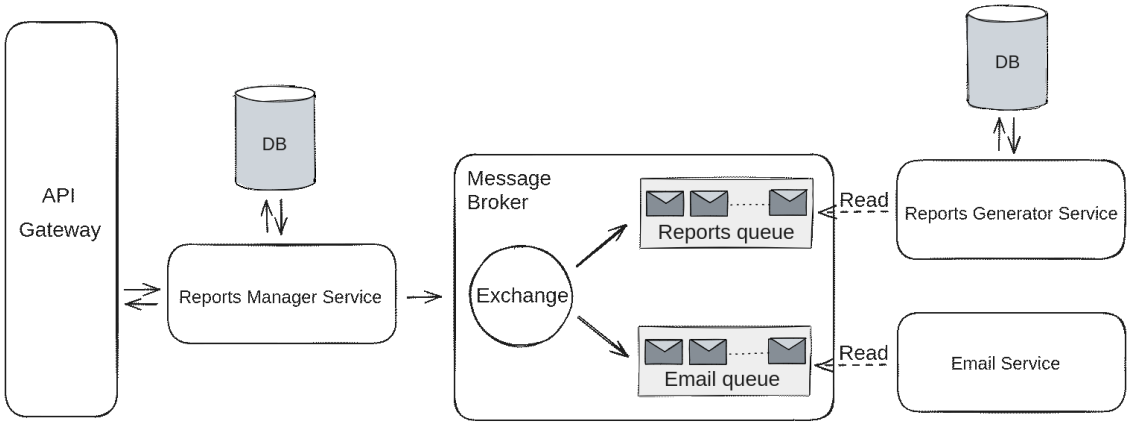


Figure 2.4: Basic async reports Microservices architecture

## 2.3 Go backend

Go is a statically typed, compiled programming language designed for simplicity and concurrency. Developed and then announced in 2009 by Google, its syntax is inspired by C. Go features fast compilation, garbage collection, and a rich standard library. Its concurrency model is based on goroutines and channels, enabling efficient concurrent programming without the complexity of traditional threading. With its focus on simplicity and performance, Go has gained popularity in building cloud-native systems, command-line tools and scalable backend services.

### 2.3.1 Goroutines and Channels

A goroutine is a lightweight thread managed by the Go runtime. Goroutines enable concurrent execution without the overhead traditionally associated with operating system threads. This means that even if you have thousands of goroutines, they

can be efficiently scheduled onto a smaller number of OS threads, reducing context switching overhead and minimizing memory usage. A single goroutine has a starting stack size of just 2KB in the current 1.22 version and dynamically adjusts its size based on the needs.

In many async/await programming languages like JavaScript, asynchronous operations require functions to be explicitly marked as async, leading to **function coloring**. This means that once a function is marked as async, any function that calls it and needs to await its result must also be marked as async, creating a cascade that complicates the code structure. Additionally, managing deeply nested callbacks can lead to **callback hell**, making the code hard to read and maintain. In Go, any function can run concurrently with the *go* keyword. This design eliminates the complexities of function coloring and callback hell, making concurrent programming more intuitive.

Channels provide an efficient and elegant way to communicate between goroutines, enabling safe concurrent programming. They provide a way for goroutines to communicate and share data without the use of traditional mechanisms like mutexes.

*Don't communicate by sharing memory; share memory by communicating.*

There are two types of channels:

**Unbuffered Channel** When sending a value on an unbuffered channel, the sending goroutine will wait for another one to receive the value. Similarly, the receiving goroutine waits for a value to be sent to the channel. This synchronization property of unbuffered channels is what makes them so useful in the context of making sure that two goroutines are synchronized at some exact point during their execution.

```go
package main

import "fmt"

func main() {
    ch := make(chan string) // Unbuffered channel

    // Producer goroutine
    go func() {
        ch <- "Hello" // Block until another goroutine receives
        fmt.Println("Message␣sent")
        // Printed after the message is received
    }()

    // Consumer
    msg := <-ch // Unblock the sending goroutine
    fmt.Println(msg) // Output: Hello
}
```

**Buffered Channel** A buffered channel can store a number of elements in it without having to block the goroutine that is sending. Unlike an unbuffered channel, if a value is sent and the Channel is full, the sending goroutine is blocked until the channel has enough space to accommodate the value. On the other hand, a receiving goroutine blocks if the channel is empty. So, a buffered channel is more flexible. In many cases, buffered channels help speed up the process because sending and receiving are decoupled.

Under the hood, a buffered channel is implemented as a **ring buffer FIFO** (First-In-First-Out). This means that the order of the elements is maintained: the first one to enter is the first to exit. In this data structure, when the end is reached, it wraps around to the beginning, making efficient use of the fixed-size allocated memory block.

```
package main

import "fmt"

func main() {
    ch := make(chan int, 2) // Buffered channel with capacity 2

    // Producer goroutine
    go func() {
        ch <- 1
        ch <- 2
        fmt.Println("First two values sent")
        fmt.Println("Buffered channel is now full")

        ch <- 3 // Block until there is space in the buffer
        fmt.Println("Third value sent after buffer had space")
    }()

    // Consumer
    msg1 := <-ch // Receive the first value
    // The above receive will unblock the producer
    // allowing it to send the third value
    fmt.Println(msg1) // Output: 1

    msg2 := <-ch // Receive the second value
    fmt.Println(msg2) // Output: 2

    msg3 := <-ch // Receive the third value
    fmt.Println(msg3) // Output: 3
}
```

### 2.3.2 Go kit

Go Kit is a toolkit for building microservices with Clean Architecture in mind. This approach involves the separation of problems within an application and the independence of business logic from distribution mechanisms and infrastructure. That can be achieved by organizing the code into distinct layers. The architecture of the Go Kit application follows these three layers:

- **Transport layer** allows flexibility to multiple protocols and interfaces through communication mechanisms such as HTTP, gRPC, and CLI. This layer man-

ages input and output by converting external requests into a format that the application can process.

- **Endpoint layer** sits between the Transport layer and the core business logic. It defines the use cases of the application by showing the input and output of every operation. In Clean Architecture, this is the Request Model and Response Model. This layer does the translation of the user's actions into something the business logic can understand and then act upon.

- **Service layer** is where the actual business logic resides. It holds the core functionalities of the application and rules, which do not depend on how requests are accepted or responses are sent back. The invariability of the core logic maintains the code modularity, keeping it highly maintainable across any changes to the user interface or infrastructure.

## 2.4 Final Architecture

The final architecture is a hybrid structure of synchronous and asynchronous modules. In fact the project is implemented in an existing structure one must make the best use of the internal services already available, such as the API gateway, the message broker, the email service and the devices data service. The reports manager service communicates with the reports generator service synchronously, because jsreport only exposes a REST API and since our goal is to create a simple system and using relatively basic templates, report generation takes only a few tenths of a second, we can still use a blocking architecture at least for the MVP. On the other hand, for convenience we use the asynchronous architecture for sending emails, since there is already an existing service for this within the Zerynth cloud.

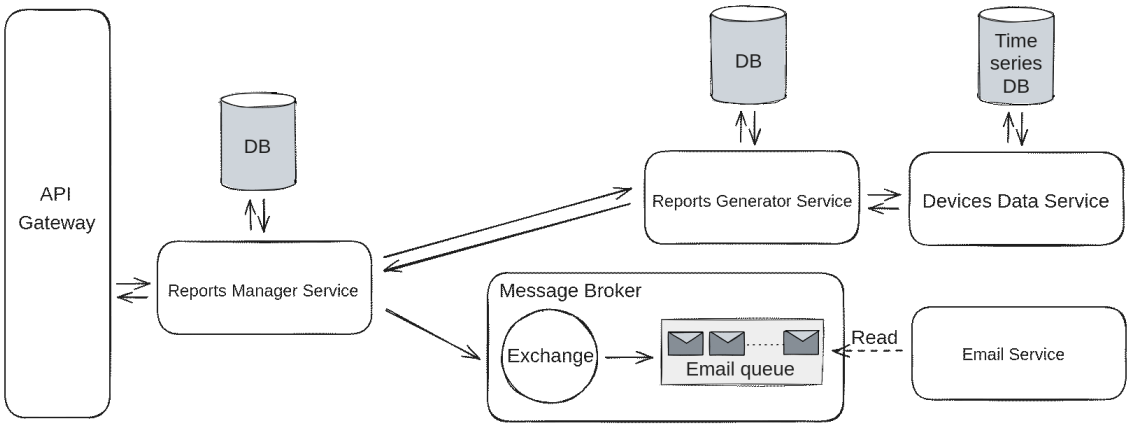


Figure 2.5: Final reports Microservices architecture of the project

# Chapter 3

# Development

## 3.1 Containerization with Docker

Docker is the most common tool in DevOps that allows developers to automate the deployment of apps inside portable, lightweight containers. Containers build an application and all its dependencies to run reliably from one environment to another. Docker helps to create, deploy, and run applications using a container that is isolated from each other and the operating system. Docker Compose, on the other hand, is a tool that is specifically designed to operate multiple container Docker applications. It does so by defining and running multi container applications through the service configuration in a YAML file. With a single command, is possible to start, stop, or rebuild services. This makes Docker Compose efficient for application development, testing, and deployment in situations where several services need to be running side by side. A Compose file describes the services, networks, and volumes an application needs in a single way that can be used consistently across different environments. Docker Compose allows the definition of a web application, including its front-end, back-end, and database as separate services, to be described in one file. It provides a way of managing the entire lifecycle of these services with simple commands, hence

efficient orchestration and scalability of containerized applications.

In this project I used docker to create the container for the go microservice, downloading within the container *golang:1.19-alpine*, the go 1.19 version on the alpine linux distro famous for its lightness. The same applies for the Dockerfile related to the graphic user interface (GUI) done in React.js where I downloaded *node:16-alpine*. Then thanks to docker compose, I uploaded the various Dockerfiles of each service specifying the port and other parameters, so also for the two databases for which I downloaded the official postgres image.

Special attention should be paid to a container with a particular task: the database migration system. As its name suggests, its task is to manage the data migration of databases automatically. I added this feature, mainly to initialise the database schema, automatically whenever I needed to initialise the architecture with *docker-compose up* command. This way, by specifying the table schema only once, a consistent and reliable application is achieved in the different development and production steps.

## 3.2 Reports Generator Microservice

The task of this service is to generate reports, via a REST request sent by the reports manager service with all the parameters necessary for the correct generation of the new report. This microservice is based on jsreport of which there is an official docker image, so it is easy to install and connect to a database that will then be used to store both the templates and the generated reports. Automatically, jsreport exposes the path */api/report* to receive the POST request needed to create the report. An example of request body cloud be:

```
{
    "template": { "name" : "machine_consumption"  },
    "data" : {
      "title": "Machine Consumption Report",
      "workspace_id": "323e7402-b12e-13e3-c420-662218289500"
      "machine_id": "550e8400-e29b-41d4-a716-446655440000"
      "start_timestamp": "2022-09-04T15:15:00+02:00",
      "end_timestamp": "2022-10-02T14:00:00+02:00",
    },
    "options": { "reports": { "save": true } }
 }
```

These few pieces of information are enough to create a comprehensive report, because thanks to jsreport we can run a javascript script before the actual generation, and with this script a request is sent to an external service to obtain the data from the industrial machine we want to represent, for which we need the start timestamp, the end timestamp and the machine id, to get the time frame in which we want to see the consumption of the given machine.

If we go to the port assigned to jsreport, we can see the classic jsreport GUI where we can create or customize our templates using html, css and an engine like handlebars that is a software component used to merge templates with data to produce the final document.

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Machine Consumption Report</title>
</head>
<body>
    <header>
        <h1>Machine Consumption Report</h1>
        <h2>(ID: {{machine_id}})</h2>
        <p>Start: {{start_timestamp}}</p>
        <p>End: {{end_timestamp}}</p>
    </header>

    <div class="content">
        <div class="statistics">
            <h3>Consumption Statistics</h3>
            <table>
                <thead>
                    <tr>
                        <th>Time Period</th>
                        <th>Power Consumption (kWh)</th>
                        <th>Operational Hours</th>
                    </tr>
                </thead>
                <tbody>
                    {{#each statistics}}
                    <tr>
                        <td>{{this.power_consumption}}</td>
                        <td>{{this.operational_hours}}</td>
                    </tr>
                    {{/each}}
                </tbody>
            </table>
        </div>
    </div>
</body>
</html>
```

This is just a very basic example to show how a template can be structured using jsreport. We can then create precise and eye-catching templates thanks to customized graphics and charts, which will be used by clients to better analyze their industrial data.

## 3.3 Reports Manager Microservice

As anticipated, this service is tasked with managing reports from their creation through to their listing and potential deletion. The microservice is built using Go kit to leverage its simple and intuitive three-layer structure.

Upon receiving a trigger, for example a scheduled task or an API call, the service sends a REST request to the Reports Generator Service. This external service then generates a report and returns a URL pointing to the location of the generated report. After obtaining the URL of the generated report, the Reports Manager Service sends a message containing the report URL to the Email Service via RabbitMQ. This message broker facilitates asynchronous communication between services, ensuring that email notification is decoupled from report generation.

```
+-- src/
    +-- models/
    |   +-- emalier.go
    |   +-- reports.go
    +-- service/
    |   +-- endpoint.go
    |   +-- logging.go
    |   +-- repository.go
    |   +-- repository_test.go
    |   +-- service.go
    |   +-- transport.go
    |   +-- transport_test.go
    +-- Dockerfile
    +-- Dockerfile.local
    +-- go.mod
    +-- go.sum
    +-- main.go
```

This is the folder structure of the service.

- **repository.go** manages data persistence and interactions, encapsulating all db related operations, like entering information about a report in the database, such as id, title, URL, start timestamp, end timestamp. Then it also needs a query for the soft deletion of the report given a certain id, for soft deletion we mean marking the report as deleted for example by adding a deletedAt field, so customers could potentially review the deleted reports in the future. Then I added the feature of listing all reports that do not have the deletedAt field. Since the repository file is pivotal for the entire project, it requires an unit test for each function.

- **service.go** contains the business logic for the correct reports management, calling the functions implemented in the repository to interface with the db and perform operations in the correct order and possibly logging errors in interfacing with the db. This is also the layer where I added the AMQP client request, the protocol used by Rabbitmq over TCP to send the mail.

- **transport.go** is where all the endpoint paths are declared using the Go-chi routing library. It is also where the functions for encoding and decoding are implemented. These functions play a crucial role in data transmission between the client and server. Encoding is the process of transforming data structures or objects into a format that is suitable for storage or transmission. It converts complex information into a string or byte format that can be easily sent over the network. In a REST API using JSON, encoding would involve converting a Go data structure into a JSON string. Decoding is the reverse process. It involves taking the encoded data (such as a JSON string received in an HTTP request) and converting them back into a native data structure that the application can work with.

- **main.go** serves as the orchestrator for setting up and running the microservice. It uses dependency injection to integrate various features, such as logging, database connectivity, and message publishing into the service architecture. This approach streamlines the initialization of each component and ensures that they are loosely coupled, enhancing the maintainability and scalability of the application.

## 3.4 React.js Web App

The backend architecture I've developed also requires a client interface to interact intuitively with the report generation process. Therefore, after creating a quick mock-up with Figma to get an idea of how to arrange the components on the page, I proceeded to develop the client-side application. This mock-up helped to visualize the layout and functionality, ensuring a user-friendly experience.

For the framework, we chose React.js with TypeScript because it is the one I am most proficient in. Additionally, since the main Zerynth web app is also built using React, this choice will facilitate the implementation of new features in the future. The consistent use of React across projects ensures that the codebase remains clean and maintainable, aiding in the seamless integration of updates and enhancements.

The web app consists of two pages:

- **home page**, which is used to display a table listing the generated reports. I have used Material UI (MUI) as the component library for this purpose. Thanks to this library, which offers a wide range of pre-built components such as for example buttons, text fields, tabs, and tables, so it was straightforward to implement a well-tested and fully functional table. Moreover, MUI tables comes with built-in features that allow for the filtering and searching of reports, making it extremely efficient to find reports of interest.

- **form page**, accessed by clicking a button to create a new report on the homepage. The goal of this page is to allow users to select parameters for creating a report in an intuitive manner. It includes a text field for the report's title, the notification email and a select component for choosing the desired template. If the selected template pertains to an individual machine rather than the entire industry, users can select it using another dedicated select component. Additionally, a DateTime Range Picker enables users to quickly select the period they want to analyze. Finally, a button allows users to submit the chosen parameters to the reports manager service, which then processes the creation of the report.

To develop this web app, I used Redux, specifically incorporating Redux Toolkit (RTK) Query, to optimize state management and streamline data interactions. RTK Query is a powerful tool within the Redux ecosystem that automates and simplifies the data fetching process directly integrated with the Redux store. It allows me to predefine API endpoints, automating how query parameters are generated from arguments and how responses are transformed for efficient caching. It also provides React hooks that encapsulate the entire data fetching process. These hooks deliver crucial state management features such as tracking loading states and managing the lifetime of cached data.

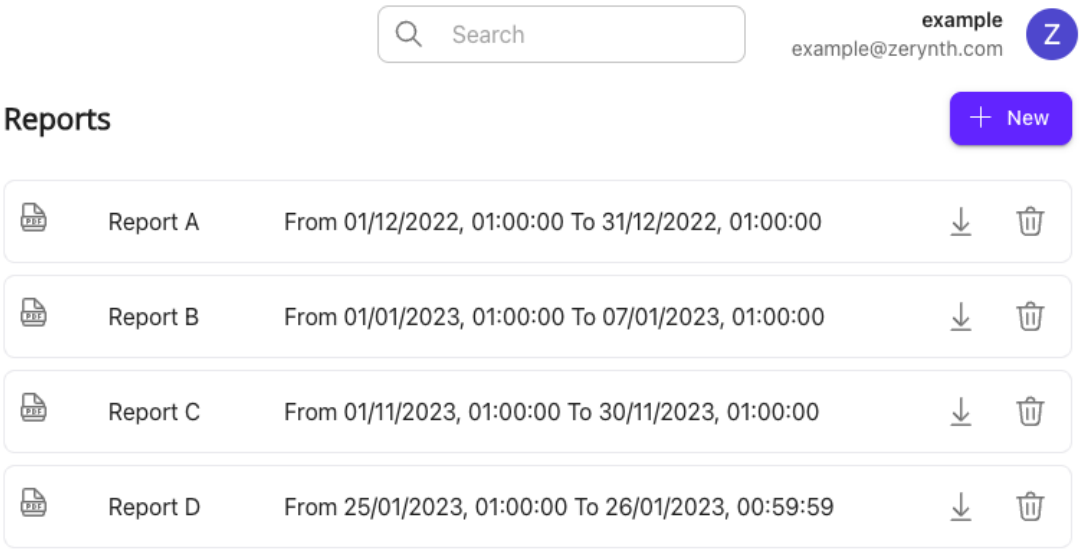


Figure 3.1: Reports home page

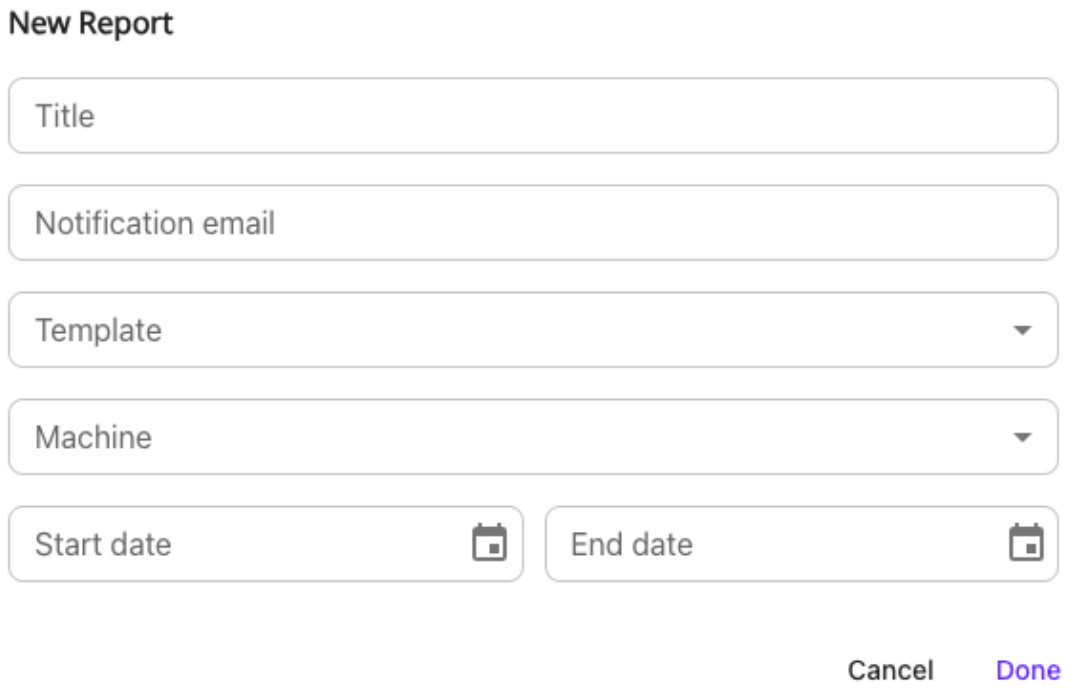


Figure 3.2: Form for creating a new report

# Chapter 4

# Conclusion

From my valuable experience working at Zerynth, I learned to manage the entire lifecycle of a project from architecting prototypes to development and testing. I was pleasantly surprised by the intuitiveness of new technologies learned such as Go, which I continue to enjoy using. My skills in handling asynchronous microservices and message brokers like RabbitMQ have improved significantly. I also learned to manage and coordinate project milestones with the team to meet deadlines and navigate trade-offs in technical engineering decisions.

After completing my internship, I had the pleasure of continuing my work at Zerynth. We focused on evolving this MVP to fully integrate it within the Zerynth cloud platform. To achieve this, we enhanced its scalability to handle much more complex templates and achieved a completely asynchronous architecture. In addition, we added the feature to schedule automatic report generation at specific times, either daily, weekly, or monthly.

Finally, I am pleased with the idea of having made a positive impact on Zerynth's clients, who can now receive their reports in a more customized and intuitive manner. This enhancement facilitates their efforts to optimize production and consumption efficiency.

# Bibliography

[1] *Zerynth docs*, `https://docs.zerynth.com/`

[2] *Effective Go*, `https://go.dev/doc/effective_go`