



FACULTY OF SCIENCE AND TECHNOLOGY

## Bachelor's Thesis

study program/specialisation:  Bachelor in engineering / Bachelor of Science in Computer Science	Spring semester 2024  Open or <del>Confidential</del>
Author(s): Alexander Bjørnås	
Faculty supervisor: Nejm Saadallah  Netpower supervisor: Anders Endresen	
Tittel på bacheloroppgaven: Timeføringssystem  English title: Time tracking system	
Credits: 20	
Keywords:  Time tracking, C#, ASP.NET Core,  Domain-Driven Design, MUI, React, Vite	Pages: 74  + Attatchments:  Stavanger 15. may 2024

# Contents

<b>Contents</b>	<b>i</b>
<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and motivation . . . . .	1
1.2 Problem statement . . . . .	1
1.3 About the company . . . . .	2
1.4 Thesis Outline . . . . .	2
<b>2 Technology and library</b>	<b>4</b>
2.1 React . . . . .	4
2.2 ViteJS . . . . .	5
2.3 Material UI . . . . .	5
2.4 TypeScript . . . . .	5

## CONTENTS

---

2.5	ASP.NET Core 8 . . . . .	6
2.6	Entity FrameWork . . . . .	6
2.7	ASP.NET Core Identity . . . . .	6
<b>3</b>	<b>Design and construction of software</b>	<b>7</b>
3.1	Miro . . . . .	7
3.2	Figma . . . . .	10
<b>4</b>	<b>Architecture and Models</b>	<b>19</b>
4.1	Architecture . . . . .	20
4.2	Domain-Driven Design . . . . .	21
4.3	Model . . . . .	21
<b>5</b>	<b>Back-end Implementation</b>	<b>24</b>
5.1	Infrastructure implementation . . . . .	24
5.2	ASP.NET Core Identity . . . . .	26
5.3	Controller . . . . .	27
5.3.1	The GET method . . . . .	28
5.3.2	The POST method . . . . .	28
5.3.3	The PUT method . . . . .	30
5.3.4	The DELETE method . . . . .	32

## CONTENTS

---

5.4	MediatR . . . . .	34
5.5	Pipelines . . . . .	34
<b>6</b>	<b>Front-End Implementation</b>	<b>37</b>
6.1	Components . . . . .	37
6.1.1	Register Hours . . . . .	39
6.1.2	Admin Menu . . . . .	41
6.2	Pages . . . . .	41
6.3	HTTP-requests . . . . .	42
6.4	Routing . . . . .	43
6.4.1	React Hooks . . . . .	43
6.4.2	React Router . . . . .	45
<b>7</b>	<b>Results</b>	<b>51</b>
7.1	Final design . . . . .	51
7.2	Admin Menu . . . . .	62
<b>8</b>	<b>Discussion</b>	<b>71</b>
8.1	Next Steps . . . . .	71
8.2	Looking Back . . . . .	72
<b>9</b>	<b>Conclusion</b>	<b>74</b>

## **CONTENTS**

---

<b>Bibliografi</b>	<b>76</b>
<b>Attachments</b>	<b>76</b>
<b>A Programlisting</b>	<b>77</b>
<b>B Data sheet</b>	<b>78</b>

# Abstract

The need for efficient and user-friendly time tracking solutions is crucial for any company aiming to improve productivity. Many existing applications, such as SharePoint Webpart and Power Apps, require a Microsoft account and can be difficult for users with limited technological experience. This thesis presents the development of a standalone time logging/tracking web application that simplifies the hour logging process.

The main objective of this project was to create an easy to use application using modern technologies. The application features a minimalistic design to ensure ease of use for regular users while providing administrators with necessary tools for management and customization. Key functionalities include user-friendly interfaces for logging and viewing hours, along with administrator privileges for managing employees and projects.

This thesis shows how I have developed a standalone time tracking application that meets the needs of users seeking a standalone, simple and efficient solution, making it a valuable asset for companies that want a simple application to view and log their hours.

# Acknowledgements

I would like to thank Nejm Sadallah, Associate Professor in Computer Science at the University of Stavanger, for his help with writing this thesis and answering any questions I had. Additionally, I want to thank several people from Netpower for their and help with developing the application. People worth mentioning are Joakim Hannestad Tveter and Andre Mæland for their help on the front-end, Siren Melkevig for her help on the back-end.

Cecilie Dalva for helping me design the application, and Anders Endresen for helping me with any issues I had and for giving me this opportunity to work on this project.

I would also like to thank Jim Seo Markussen and Marius Kaada Heske for proof reading.

# Chapter 1

## Introduction

### 1.1 Background and motivation

The background for my bachelor thesis is when I met Netpower at IKT-dagen at the University of Stavanger, and they told me about their idea of a new application for their customers to register and keep track of their hours where it is not possible to use Sharepoint Webpart, Power Apps or other similar technologies which requires that the users have Microsoft or guest accounts. Hence comes the need for a standalone application where external users, often from subcontractors or similar, can log in and register their hours.

I accepted this project because I found it exciting and challenging. It covers many different parts of software development, from the planning stages to developing the application. This experience will be valuable for my future career.

### 1.2 Problem statement

My goal is to develop a standalone application designed to simplify the hour logging process, providing a user-friendly interface that eliminates us-



### 1.3 About the company

---

ing applications such as Sharepoint Webpart, Power Apps or other similar technologies that requires a Microsoft account, which can be a difficulty barrier for someone with limited experience with technology. This solution will specifically benefit users who prefer a simple, efficient method to track their working hours without the need for extensive setups or platform dependencies.

### 1.3 About the company

Netpower is an IT company delivering self-developed software products such as Software as a service(SaaS), messaging systems, quality management systems, websites, online stores, digital marketing services, data center services and IT operations. Netpower is an established competence center within web solutions and internet-based technology. Netpower has its head office on Forus, and also has branch offices in Bergen, Harstad and Oslo, and their own development department in HO Chi Minh City in Vietnam. [Netpower, nd]

### 1.4 Thesis Outline

2. **Chapter 2 - Technology and library**  
Presents the technology choices, as well as frameworks and libraries I used.
3. **Chapter 3 - Design and construction of software**  
Describes the design process of the front-end of the application.
4. **Chapter 4 - Architecture and Models**  
I discuss the different architecture and models used in the application, and how it is built up.
5. **Chapter 5 - Back-end Implementation**  
Demonstrates the implementation of the back-end, detailing the infrastructure and controllers and explaining their connection with the front-end.

## 1.4 Thesis Outline

---

6. **Chapter 6 - Front-end Implementation**  
Demonstrates the implementation of the front-end, highlighting key components and how it connects with the back-end.
7. **Chapter 7 - Results**  
Presents the final front-end design through a YouTube video and images, and provides an analysis of the initial design concepts to the finished application.
8. **Chapter 8 - Discussion**  
Discusses next steps for the application, and decisions that should have been made differently.
9. **Chapter 9 - Conclusion**  
Concludes my bachelor thesis.

## Chapter 2

# Technology and library

This chapter presents the technology and libraries used for developing the application. The first meetings with Netpower was used to discuss which technologies that would suit the development of this application the best. Both Netpowers recommendations and my own experiences were taken into consideration while figuring out what technologies and libraries to use.

This chapter introduces technologies such as: React, ViteJS and Material UI and ASP.NET Core 8.

### 2.1 React

React is a front-end Javascript library maintained by Meta and currently is one of the most popular frameworks. Netpower recommended that I use React as my front-end framework based on their experience with React, and makes it easier for Netpower to assist me if I encounter any issues and also streamlines the update process for them, if the application should need to be updated in the future.

## 2.2 ViteJS

---

## 2.2 ViteJS

Vite is a build tool that aims to provide a faster and leaner development experience for modern web projects. [Vite, nda] I choose to pair React with Vite because of the advantages it has over just a normal React application. One of the main reasons is the speed of loading the project, Vite utilises esbuild instead of webpack, Vite explains that Vite pre-bundles dependencies using esbuild. esbuild is written in Go and pre-bundles dependencies 10-100x faster than JavaScript-based bundlers. [Vite, ndb]

## 2.3 Material UI

Material UI is an open-source React component library that implements Google's Material Design. It includes a comprehensive collection of pre-build components that are ready for use in production right out of the box. [MUI, nd]

Netpower advised me to use Material UI as the design component for the front-end, as it's the library that they use for their front-end design, which makes it more in style with the rest of their applications.

## 2.4 TypeScript

W3Schools explains the following on Typescript: TypeScript is a syntactic superset of JavaScript which adds static typing. This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add types [W3Schools, nd].

Netpower recommended me to use TypeScript due to focus on maintainability, as well as their familiarity and experience with the language.

## 2.5 ASP.NET Core 8

---

## 2.5 ASP.NET Core 8

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps. [Microsoft, 2023]. ASP.NET Core uses the C# programming language, due to earlier experiences with .NET and the programming language.

## 2.6 Entity Framework

Entity Framework is a modern object-relation mapper that lets you build a clean, portable, and high-level data access layer with .NET (C#) across a variety of databases, including SQL Database (on-premises and Azure), SQLite, MySQL, PostgreSQL, and Azure Cosmos DB. It supports LINQ queries, change tracking, updates, and schema migrations [Microsoft, nd].

I decided to use SQLite as the database due to my previous experience with it. Entity Framework Core made the SQLite database's changes and updates easy to maintain throughout this project.

## 2.7 ASP.NET Core Identity

Rick Anderson writes the following: ASP.NET Core Identity is an API that supports user interface (UI) login functionality, and manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more. [Anderson, 2024]

I opted to use ASP.NET Core Identity for my application because of its feature set and easy integration with ASP.NET Core. This also ensures that my application had everything necessary to implement a secure authentication.

## Chapter 3

# Design and construction of software

In this chapter, I discuss what software tools I used to design the applications front-end, with the help from Netpower. The main tools chosen for this task were Miro, which was used as a wireframe/prototype and Figma, for a more finalized design. the finalized front-end of the application can be seen in Chapter 7.

### 3.1 Miro

Miro is the online workspace for innovation that enables distributed teams of any size to dream, design [Miro, 2023]. Miro is a whiteboarding platform, and offers a range of tools I used to design my wireframe/prototype.

In the following images I show how I designed the wireframe/prototype in Miro before discussing with Cecilie Dalva (Interaction designer at Netpower) to finalize the design of the completed application.

### 3.1 Miro

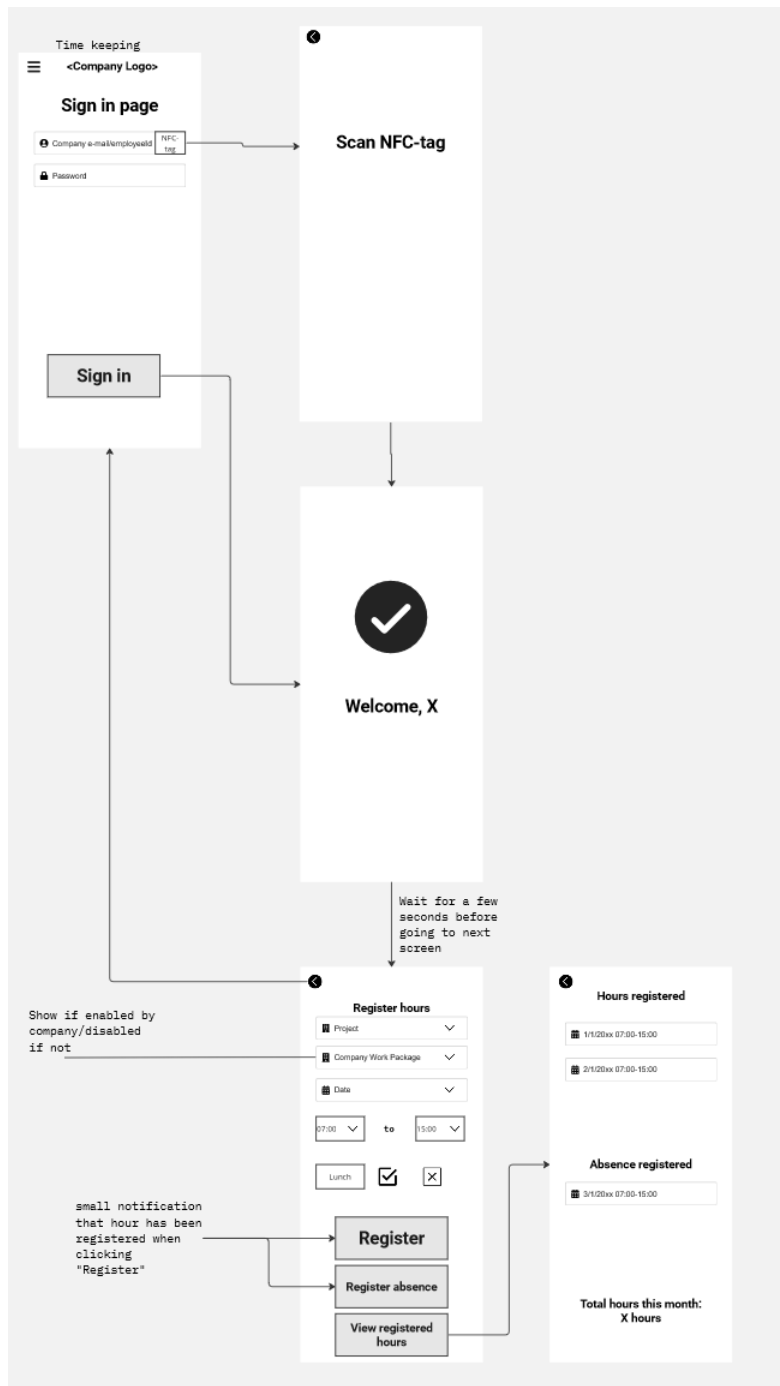


Figure 3.1: Login page and register hours and view page.

### 3.1 Miro

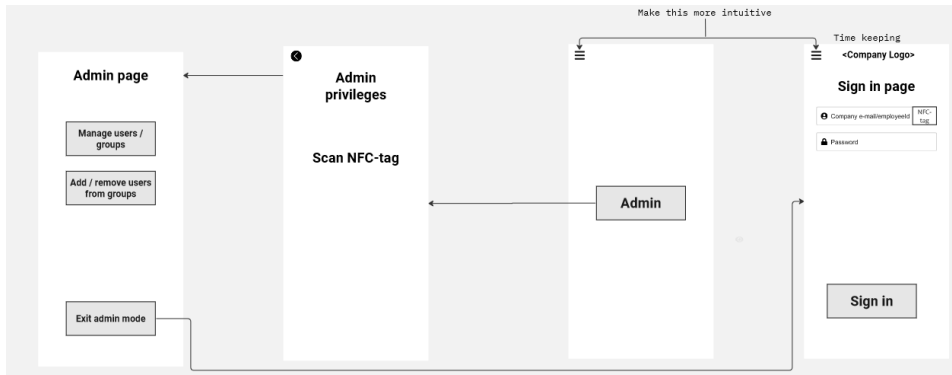


Figure 3.2: Entering admin mode.

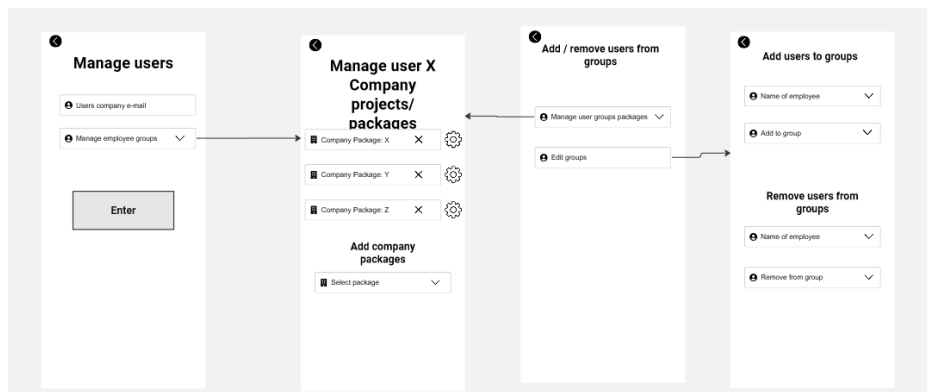


Figure 3.3: Manage employees and their projects and work packages.



## 3.2 Figma

---

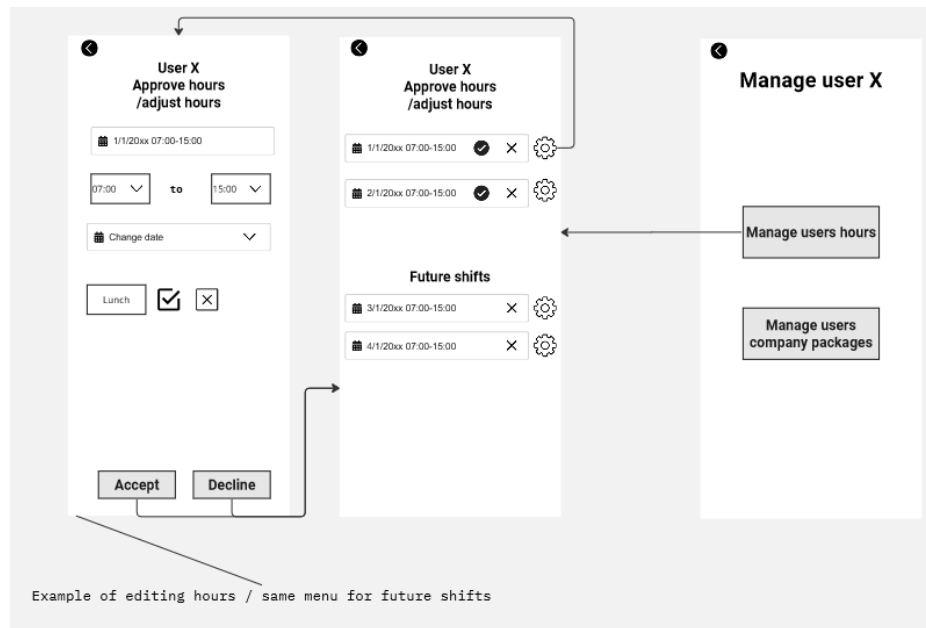


Figure 3.4: Approving hours.

## 3.2 Figma

Figma design is for people to create, share, and test designs for websites, mobile apps, and other digital products and experiences. It is a popular tool for designers, product managers, writers and developers and helps anyone involved in the design process contribute, give feedback, and make better decisions, faster [Figma, nd].

After using Miro to create the wireframe/prototype as explained in 3.1, I sat down with Cecilie Dalva and Anders Endresen and discussed how the design should look, and the following pictures is how the end product should look like.

## 3.2 Figma

---

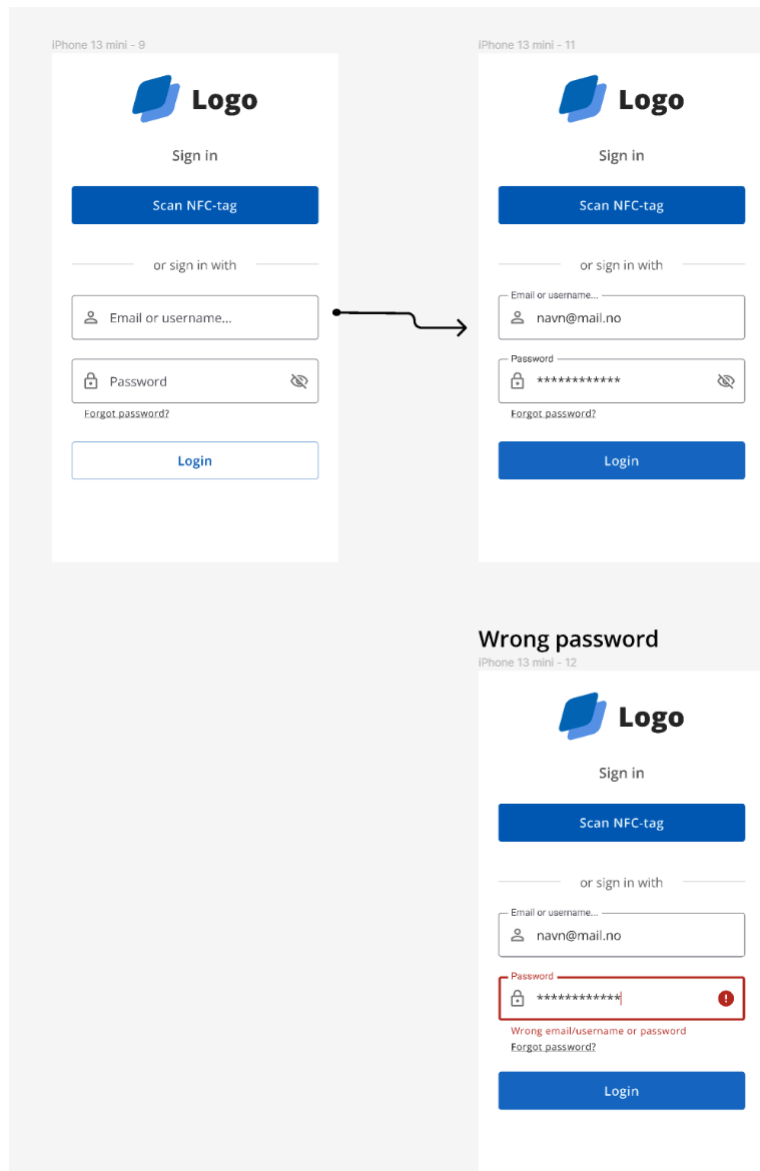


Figure 3.5: Sign in page.

## 3.2 Figma

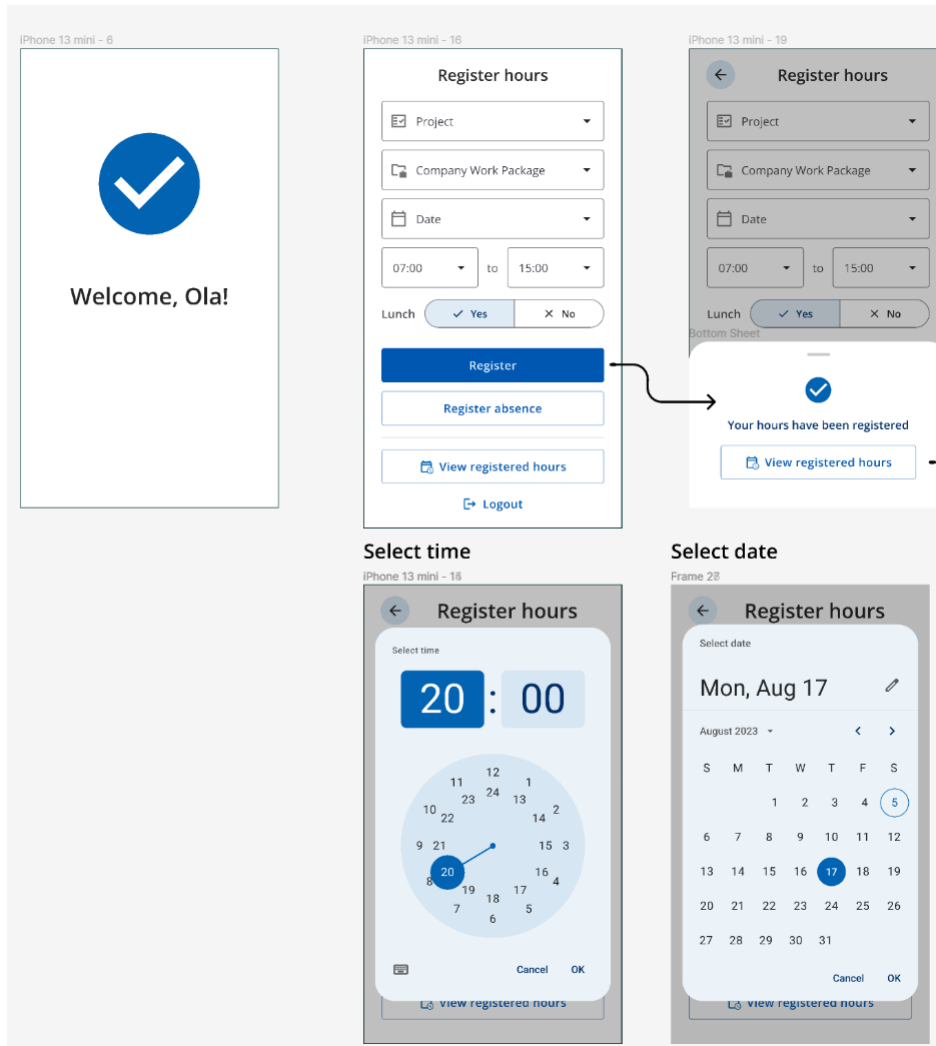


Figure 3.6: Welcome page and register hours page.

## 3.2 Figma

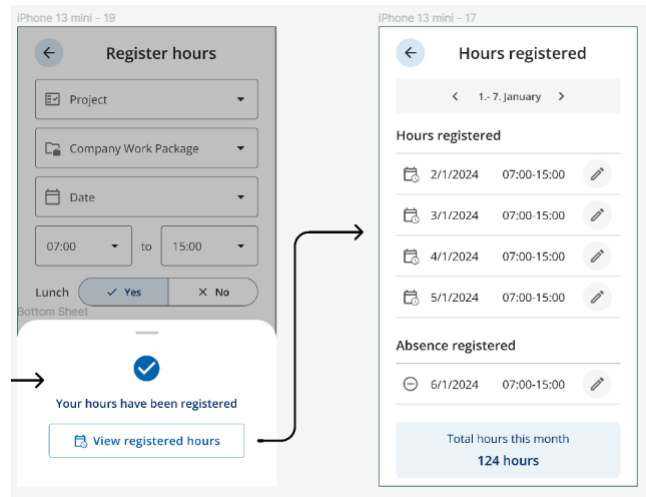


Figure 3.7: Hours registered page.

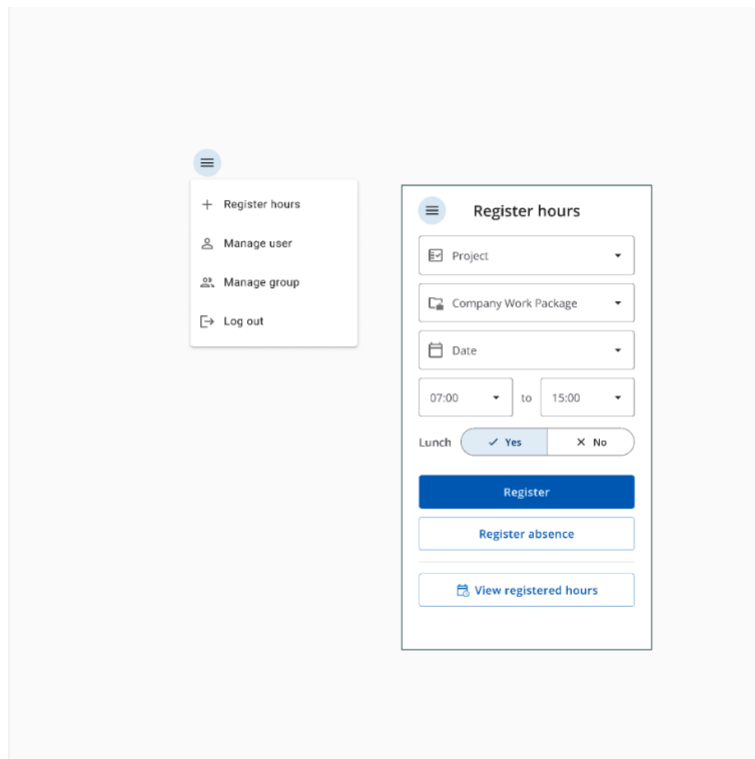


Figure 3.8: Admin menu after logging in.

### 3.2 Figma

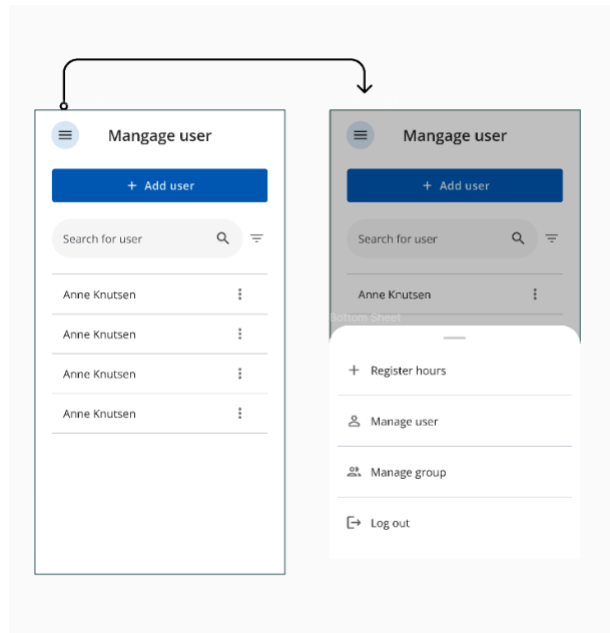


Figure 3.9: Manage employees and the admin menu.

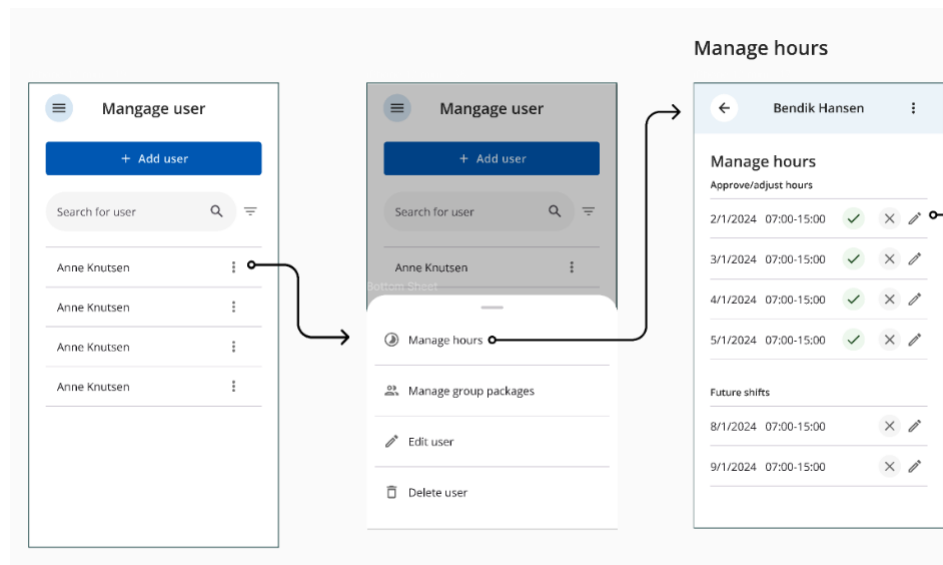


Figure 3.10: Manage employees and manage hours.

## 3.2 Figma

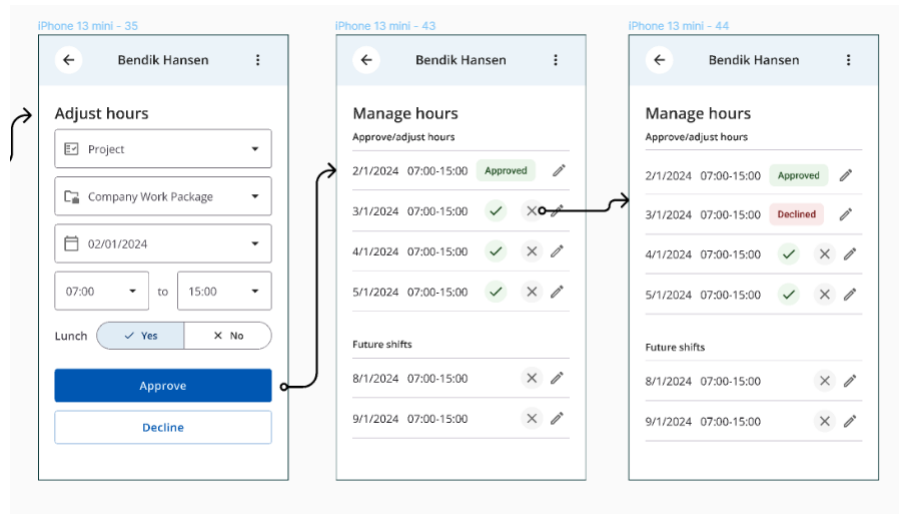


Figure 3.11: Approving or declining hours.

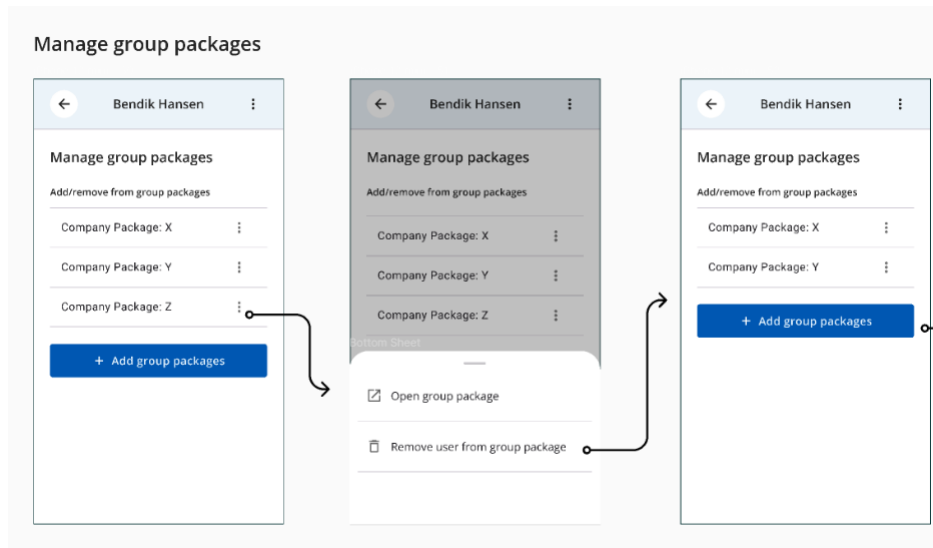


Figure 3.12: Manage work packages

## 3.2 Figma

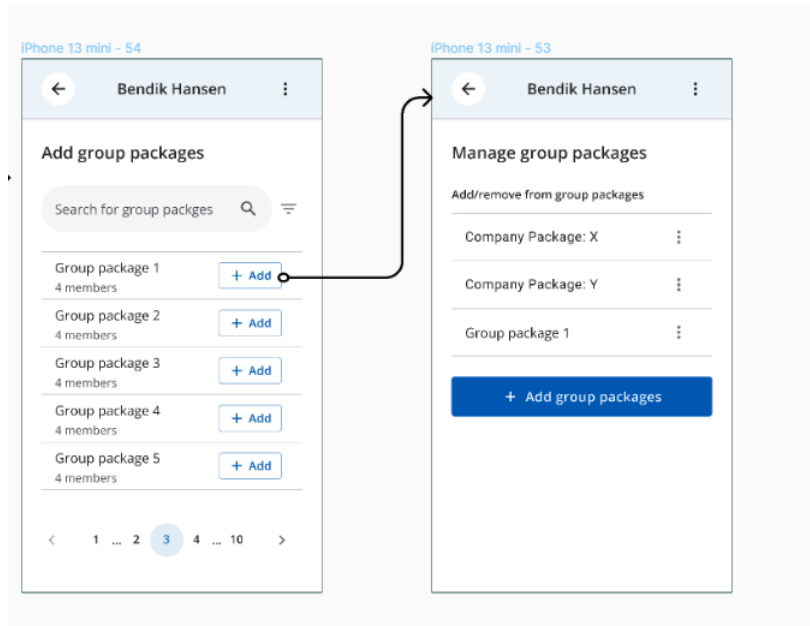


Figure 3.13: Add employee to work package.

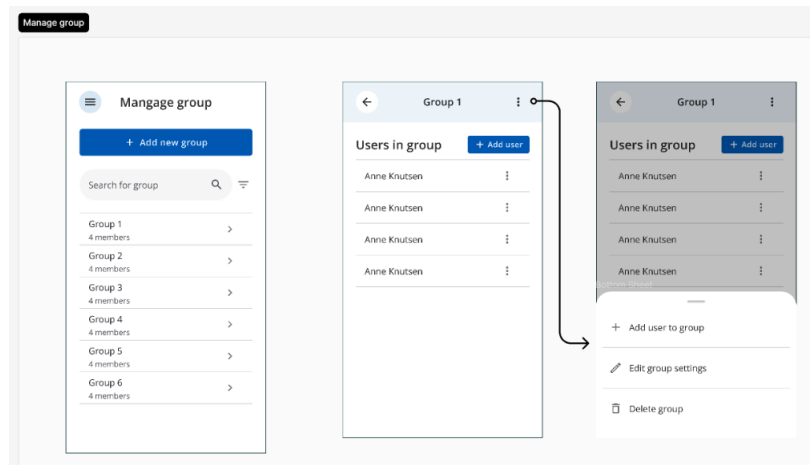


Figure 3.14: Viewing project and adding employees to projects.

## 3.2 Figma

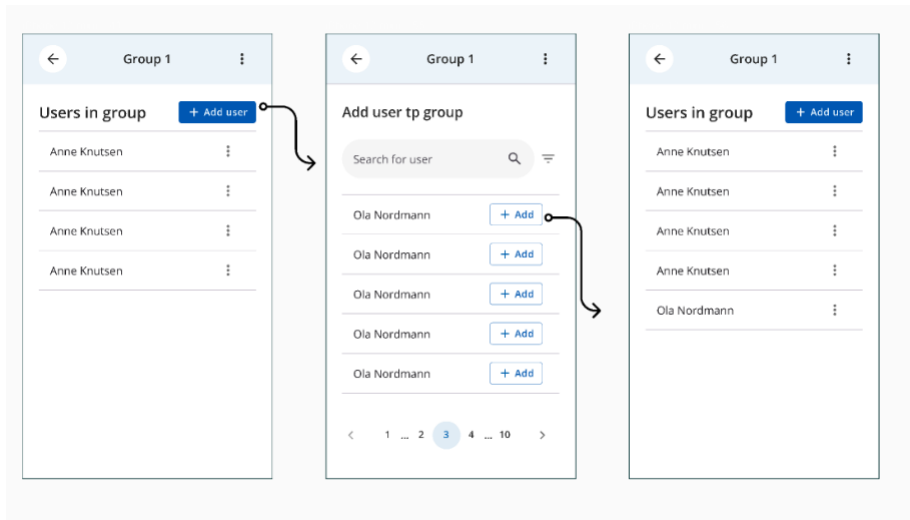


Figure 3.15: Viewing project groups and adding employees.

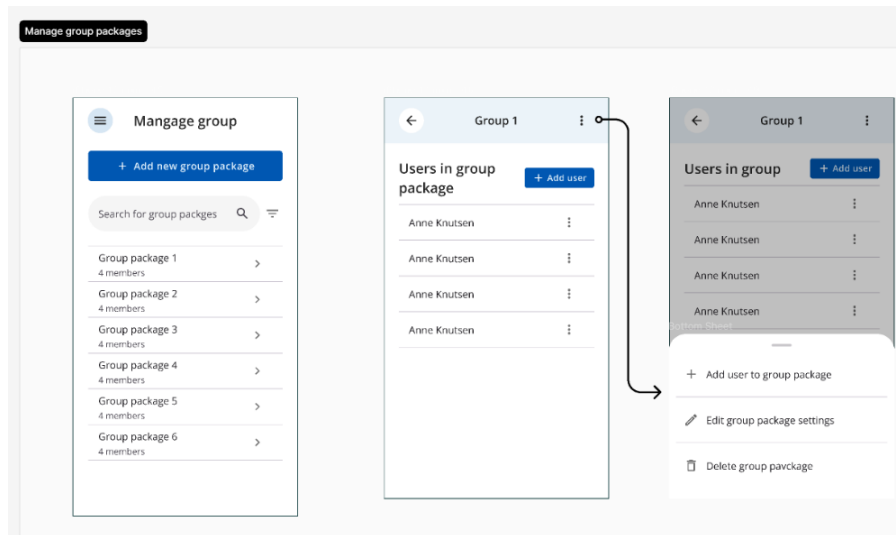


Figure 3.16: Viewing work package groups and options.



### 3.2 Figma

---

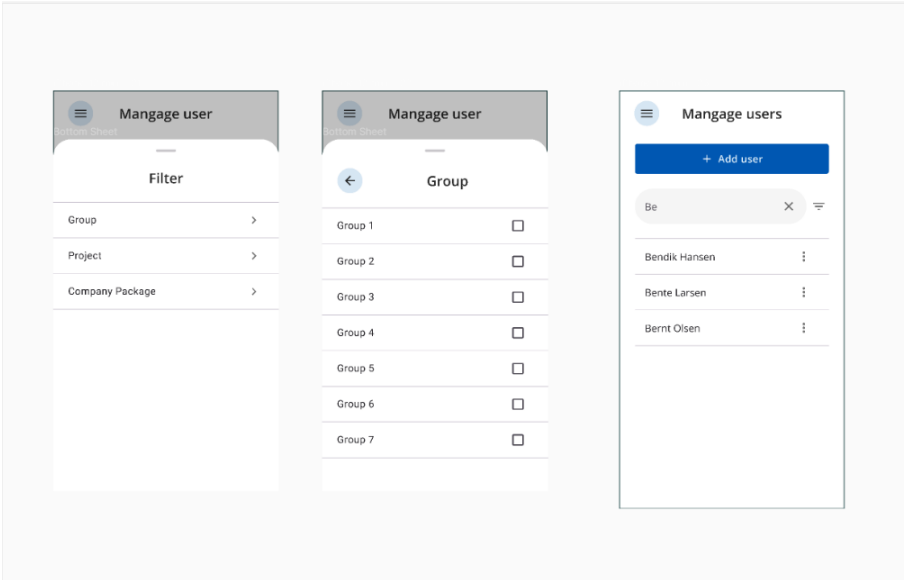


Figure 3.17: How the filter works.

## Chapter 4

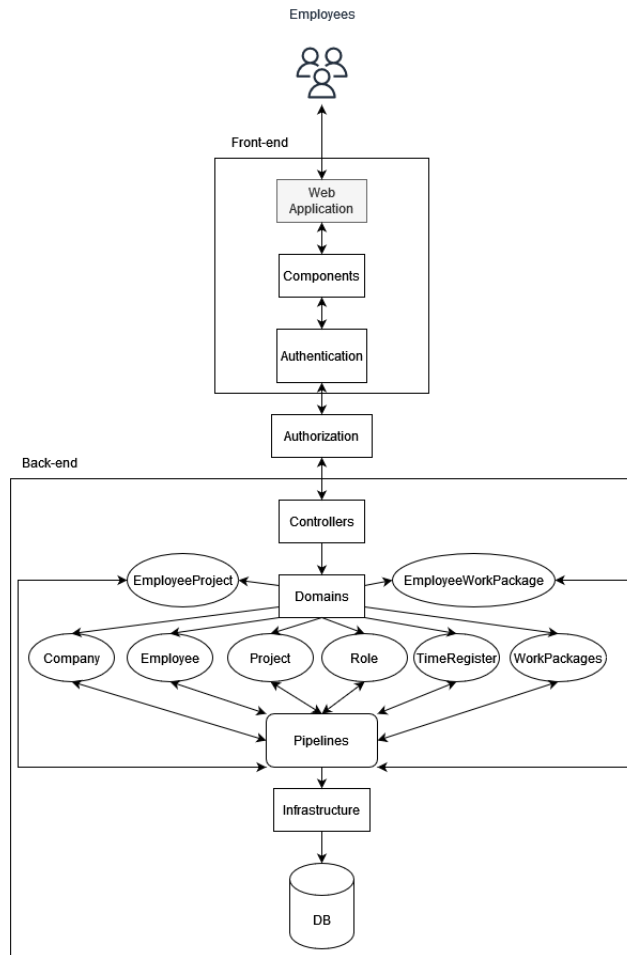
# Architecture and Models

This chapter details my approach on how I made maintainable code using different architecture and design strategies. This chapter also includes information on domain-driven design, a software development methodology which is used in this project.

## 4.1 Architecture

---

### 4.1 Architecture



**Figure 4.1:** Relationship among the various domain objects.

In Figure 4.1, each domain has its own associated pipelines, however, for simplicity sake, all pipelines are represented as a single pipeline in the diagram.

## 4.2 Domain-Driven Design

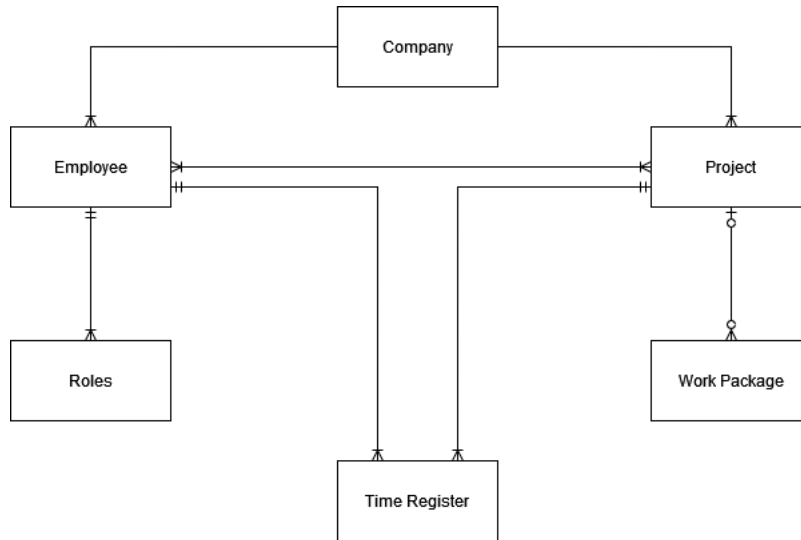
---

## 4.2 Domain-Driven Design

Domain-Driven Design(DDD) is a collection of principles and patterns that help developers craft elegant object systems. Properly applied it can lead to software abstractions called domain models. These models encapsulate complex business logic, closing the gap between business reality and code [Laribee, 2016].

## 4.3 Model

The Figure 4.2 below, visualizes the relationship between the different domain objects.



**Figure 4.2:** Entity relationship diagram.

**Company:** The Company object represents an organization that uses the time tracking system. It serves as the central entity and has relationships with employees and projects.

**Employee:** This entity represents individuals working for the company. Employees are directly linked to the company. Their roles are also defined to

### 4.3 Model

---

accommodate various employment types, such as external or temporary employees.

**Roles:** The Roles entity links to employees to define their responsibilities and access within the system. It helps manage what tasks or projects an employee can log time against, depending on their role.

**Project:** Projects are associated with the company, so multiple projects can be initiated by a single company. Employees are linked to projects as they need to assign their registered hours to specific projects they've worked on.

**Work Package:** These are subdivisions of projects that allow for more detailed task management, such as specifying work on particular aspects of a project (e.g., electrical systems in a building). Work packages are not used by every company, their utilization depends on the specific needs of the project and company.

**Time Register:** This is where the employees log their hours. It is connected to both employees and projects, ensuring that hours are recorded accurately against the correct project. An employee can have multiple time registrations across different projects.

### 4.3 Model

---

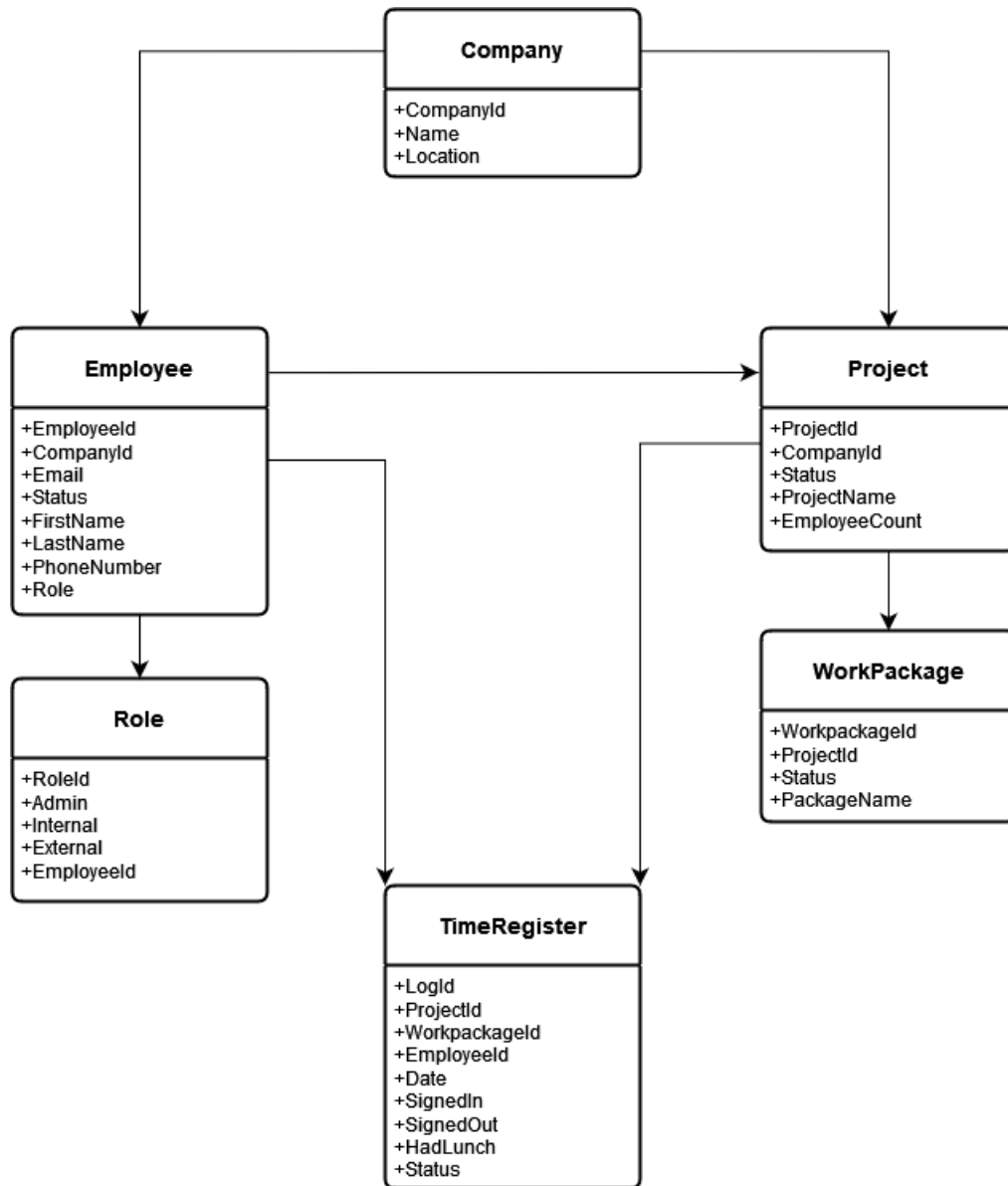


Figure 4.3: Overview of system entities and their connections.

## Chapter 5

# Back-end Implementation

The back-end is structured into three sections: Controllers, Domains and Infrastructure. The controller section consists of various system controllers that communicates with the front-end. The domains section contains the domain models, their associated objects and functionalities. The infrastructure section contains the database and its initialization procedures. I explain the implementation of the three parts that creates the back-end, detailing the reasons behind the choices I made.

### 5.1 Infrastructure implementation

The data access layer of the system is implemented by using Entity Framework Core, which streamlines the interaction between the domain models and the database. The model outlines how the application entities are mapped to the underlying database. A database context class is used to specify which entities are included in the model. Also, this context class establishes a session with the database, which enables migrations to update and create the database. In this system, the model is made up of the context class named `Trcontext` and the entity sets `Company`, `Project`, `Employee`, `Role`, `WorkPackage`, `Timeregister`, `Absenceregister`, `EmployeeProject` and `EmployeeWorkPackage`. To include the entity sets into our model, the `DbSet` for each entity type is included. In Listing 5.1, all the domain

## 5.1 Infrastructure implementation

---

objects necessary for the application are added to the model. This allows Entity Framework Core to convert the entity sets into database tables and manage the entities, including reading and writing instances to and from the database.

```
1  public class Trcontext : IdentityDbContext<IdentityUser>
2  {
3      public Trcontext(DbContextOptions<Trcontext> options) :
4          ↪ base(options) { }
5      public DbSet<Company> Companies { get; set; } = null!;
6      public DbSet<Project> Projects { get; set; } = null!;
7      public DbSet<Employee> Employees { get; set; } = null!;
8      public DbSet<Role> Roles { get; set; } = null!;
9      public DbSet<WorkPackage> WorkPackages { get; set; } = null!;
10     public DbSet<Timeregister> Timeregisters { get; set; } = null!;
11     public DbSet<Absenceregister> Absenceregisters { get; set; } =
12         ↪ null!;
13     public DbSet<EmployeeProject> EmployeeProjects { get; set; } =
14         ↪ null!;
15     public DbSet<EmployeeWorkPackage> EmployeeWorkPackages { get;
16         ↪ set; } = null!;
```

**Listing 5.1:** Code excerpt showing the Trcontext class, used by Entity Framework Core to manage database operations.

Entity Framework Core provides several benefits through its built-in conventions. One such benefit, is that it automatically designates properties containing "id" as the primary key of the corresponding database table for the class. Another one is that Entity Framework Core establishes a relationship whenever it finds a navigation property on a type. A property qualifies as a navigation property if it refers to other types of entities. See Figure 4.3 on how the entities are connected.

To establish the needed relationships between these entities, Fluent API mappings are defined within the OnModelCreating method. These mappings include defining a primary keys and setting up many-to-many relationships using the specified foreign key properties, as shown in Listing 5.2.



## 5.2 ASP.NET Core Identity

---

```
1     protected override void OnModelCreating(ModelBuilder modelBuilder)
2     {
3         base.OnModelCreating(modelBuilder);
4         modelBuilder.Entity<EmployeeProject>().HasKey(i => new {
5             ↪ i.EmployeeId, i.ProjectId });
6         modelBuilder.Entity<EmployeeProject>().HasOne(e =>
7             ↪ e.Employee).WithMany(p =>
8                 ↪ p.EmployeeProjects).HasForeignKey(e => e.EmployeeId);
9         modelBuilder.Entity<EmployeeProject>().HasOne(p =>
10            ↪ p.Project).WithMany(p =>
11                ↪ p.EmployeeProjects).HasForeignKey(p => p.ProjectId);
12
13        modelBuilder.Entity<EmployeeWorkPackage>().HasKey(i => new {
14            ↪ i.EmployeeId, i.PackageId });
15        modelBuilder.Entity<EmployeeWorkPackage>().HasOne(e =>
16            ↪ e.Employee).WithMany(wp =>
17                ↪ wp.EmployeeWorkPackages).HasForeignKey(e =>
18                    ↪ e.EmployeeId);
19        modelBuilder.Entity<EmployeeWorkPackage>().HasOne(wp =>
20            ↪ wp.WorkPackage).WithMany(e =>
21                ↪ e.EmployeeWorkPackages).HasForeignKey(wp =>
22                    ↪ wp.PackageId).IsRequired(false);
```

**Listing 5.2:** Code Excerpt, this configures the relationships between the EmployeeProject and EmployeeWorkPackage.

## 5.2 ASP.NET Core Identity

As previously written in 2.7, I utilized Identity to handle the login process. This implementation serves as a backbone for both authentication and authorization within the application, ensuring robust security and streamlined user management.

I configured ASP.NET Core Identity to use IdentityUser and IdentityRole for managing users and roles, respectively. This setup was integrated with Entity Framework Core, allowing seamless interaction with the SQLite

## 5.3 Controller

---

database, which stores user credentials and role information securely.

```
1 // Identity configuration with roles
2 builder.Services.AddIdentity<IdentityUser, IdentityRole>()
3     .AddEntityFrameworkStores<Trcontext>()
4     .AddDefaultTokenProviders();
```

**Listing 5.3:** Configuring ASP.NET Core Identity with roles in Program.cs

At first run, the application ensures that essential user roles are present and creates an administrative user if not already set up. This is critical for maintaining a controlled access environment from the start.

By invoking `AddDefaultTokenProviders()`, the application is equipped to handle token generation for operations such as password resets and two-factor authentication, increasing the security.

The addition of `app.UseAuthentication()` and `app.UseAuthorization()` middleware ensures that each request is appropriately authenticated and authorized before it is processed by the application. This middleware setup is key to enforcing security policies across all endpoints.

## 5.3 Controller

The controllers are used for communicating with the front-end. The controllers execute tasks that is requested by the front-end, such as sending information, modifying database entries and other relevant activities. The back-end has a variety of controllers, depending on the scenario.

The controllers feature various methods that perform specific tasks. These methods are named `Post`, `Patch`, `Delete`, and `Get`. Each method corresponds to a specific attribute that indicates the type of Web API it belongs to, such as `HttpGet`, `HttpPost`, `HttpPatch`, and `HttpDelete`.

## 5.3 Controller

---

### 5.3.1 The GET method

The GET method retrieves data from the server. For example, the `HttpGet` method in *TimeRegisterController* lists the hours recorded for an employee by using their `employeeId`, which then displays on the front-end. See Listing 5.4.

```
1 [HttpGet("ListHoursRegistered/{employeeId}")]
2 public IActionResult GetTimeRegistrations(string employeeId)
3     {
4         if (!Guid.TryParse(employeeId, out Guid parsedEmployeeId))
5         {
6             return BadRequest("Invalid employee ID.");
7         }
8
9         var listregisterhours = _context.Timeregisters
10            .Where(tr => tr.EmployeeId == parsedEmployeeId)
11            .ToList();
12
13         return Ok(listregisterhours);
14     }
```

**Listing 5.4:** `HttpGet` that shows hours registered on a employee based on their Id.

### 5.3.2 The POST method

The POST method sends data to the server to create a new resource. For example, the `HttpPost` method in *EmployeeProjectController* allows you to add an employee to a project by submitting the employee's details, which are then processed and stored. See Listing 5.5

## 5.3 Controller

---

```
1 [HttpPost("AddEmployee")]
2 public async Task<IActionResult> AddEmployee(
3     [FromBody] AddEmployeeToProjectDto dto)
4 {
5     var newEmployeeProject = new EmployeeProject(
6         dto.ProjectId, dto.EmployeeId);
7
8     bool projectExists = await _context.Projects.AnyAsync(
9         p => p.ProjectId == dto.ProjectId);
10    if (!projectExists)
11    {
12        return NotFound(
13            $"Project with ID {dto.ProjectId} not found.");
14    }
15
16    bool employeeExists = await _context.Employees.AnyAsync(
17        e => e.EmployeeId == dto.EmployeeId);
18    if (!employeeExists)
19    {
20        return NotFound(
21            $"Employee with ID {dto.EmployeeId} not found.");
22    }
23
24    bool alreadyExists = await _context.EmployeeProjects.AnyAsync(
25        ep => ep.ProjectId == dto.ProjectId &&
26            ep.EmployeeId == dto.EmployeeId);
27    if (alreadyExists)
28    {
29        return BadRequest(
30            "The employee is already assigned to the project.");
31    }
32
33    _context.EmployeeProjects.Add(newEmployeeProject);
34    await _context.SaveChangesAsync();
35
36    return Ok(
37        $"Employee {dto.EmployeeId} added to project {dto.ProjectId}.");
38 }
```

Listing 5.5: Adding an employee to a project via POST request.

## 5.3 Controller

---

### 5.3.3 The PUT method

The PUT method is most often used to update an existing resource when you need to modify a specific entry. See Listing 5.6

## 5.3 Controller

---

```
1 [HttpPut("UpdateEmployee/{employeeId}")]
2 public async Task<IActionResult> EditEmployee(
3     string employeeId, [FromBody] EmployeeDto updateDto)
4 {
5     if (!Guid.TryParse(employeeId, out Guid parsedEmployeeId))
6     {
7         return BadRequest("Invalid employee ID format.");
8     }
9
10    var existingEmployee = await _context.Employees
11        .FindAsync(parsedEmployeeId);
12    if (existingEmployee == null)
13    {
14        return NotFound($"Entry with ID {employeeId} not found.");
15    }
16
17    // Update employee details
18    existingEmployee.FirstName = updateDto.FirstName;
19    existingEmployee.LastName = updateDto.LastName;
20    existingEmployee.PhoneNumber = updateDto.PhoneNumber;
21    existingEmployee.Email = updateDto.Email;
22    existingEmployee.Status = updateDto.Status;
23    existingEmployee.EmployeeRole = updateDto.EmployeeRole;
24
25    try
26    {
27        _context.Employees.Update(existingEmployee);
28        await _context.SaveChangesAsync();
29        return Ok($"Employee with ID {employeeId} has been
30            successfully updated.");
31    }
32    catch (Exception ex)
33    {
34        return StatusCode(500,
35            $"An error occurred while updating the employee.
36            Error details: {ex.Message}");
37    }
38 }
```

Listing 5.6: PUT method for updating an employee's details.

## 5.3 Controller

---

### 5.3.4 The DELETE method

The DELETE method is used to delete a resource. In listing 5.7, it is used to delete an employee's hours or absence records if an employee has registered something wrong.

## 5.3 Controller

---

```
1  [HttpDelete("RemoveEntry/{logId}")]
2  public async Task<IActionResult> RemoveEntry(string logId)
3  {
4      try
5      {
6          if (!Guid.TryParse(logId, out Guid parsedLogId))
7          {
8              return BadRequest("Invalid format for log ID.");
9          }
10
11         var timeEntry = await _context.Timeregisters
12             .FindAsync(parsedLogId);
13         if (timeEntry != null)
14         {
15             _context.Timeregisters.Remove(timeEntry);
16             await _context.SaveChangesAsync();
17             return Ok($"Time register entry with ID {logId} has been
18                 successfully deleted.");
19         }
20
21         var absenceEntry = await _context.Absenceregisters
22             .FindAsync(parsedLogId);
23         if (absenceEntry != null)
24         {
25             _context.Absenceregisters.Remove(absenceEntry);
26             await _context.SaveChangesAsync();
27             return Ok($"Absence register entry with ID {logId} has been
28                 successfully deleted.");
29         }
30
31         return NotFound($"Entry with ID {logId} not found in both time and
32             absence registers.");
33     }
34     catch (Exception ex)
35     {
36         return StatusCode(500, $"An error occurred while
37             deleting the entry.
38             Please try again.
39             Error details: {ex.Message}");
40     }
41 }
```



## 5.4 MediatR

---

### 5.4 MediatR

MediatR is designed for implementing the Mediator pattern in .NET applications, serving to decrease direct dependencies among components by employing a mediator object to manage communications.

As seen in the pipeline at Listing 5.8, I used IRequest interface to request for the GUIDs of projects linked to a specific employee, then the IRequestHandler interface to specify the logic for handling this request and returning the relevant ProjectIds for that employee.

### 5.5 Pipelines

The pipelines in the application connects to the database . Data entries stored in the database are retrieved, deleted or updated with these pipelines. Listing 5.8 is one of the pipelines I used, that retrieves projects linked to an employee.

## 5.5 Pipelines

---

```
1 public class GetProjectIds
2 {
3     public record Request(Guid employeeId) : IRequest<List<Guid>> { }
4
5     public class Handler : IRequestHandler<Request, List<Guid>>
6     {
7         private readonly Trcontext _db;
8
9         public Handler(Trcontext db)
10        {
11            _db = db;
12        }
13
14        public async Task<List<Guid>> Handle(Request request,
15            ↪ Cancellation token cancellationToken)
16        {
17            var projectIds = await _db.EmployeeProjects.Where(ep =>
18                ↪ ep.EmployeeId == request.employeeId).Select(ep =>
19                ↪ ep.ProjectId).ToListAsync();
20            return projectIds;
21        }
22    }
23 }
```

**Listing 5.8:** Pipeline to fetch ProjectIds linked with a given employeeId

## 5.5 Pipelines

---

```
1 [HttpGet("GetProjectsByEmployee/{employeeId}")]
2 public async Task<ActionResult<List<Guid>>> GetProjectsByEmployee(Guid
   ↳ employeeId)
3     {
4         var request = new GetProjectIds.Request(employeeId);
5         var projectIds = await _mediator.Send(request);
6         if (projectIds == null || !projectIds.Any())
7             return NotFound("No projects found for this employee.");
8
9         return Ok(projectIds);
10    }
```

**Listing 5.9:** Code used to retrieve projectIds for a specified employee using MediatR.

## Chapter 6

# Front-End Implementation

In this chapter, I show the implementation of the front-end for the application, and how the front-end works with the back-end. The front-end is developed using React Vite, and the folder structure is as follows: Components and Pages, the two folder consists of different elements which together builds up the application. Additionally, outside these two folders, there are three more important files: Main.tsx, App.tsx and AuthContext.tsx.

### 6.1 Components

The components are responsible for both rendering elements on the front-end, and also handling interactions with the back-end. As can be seen in the example Listing 6.1 and Listing 6.2 which is the ‘SignIn’ component, responsible for employee authentication. The component manages the employees input, handles state for input fields and error messages, and communicates with the authentication API to verify the employees credentials.

## 6.1 Components

---

```
1 export default function SignIn() {
2   const [emailUsername, setEmailUsername] = useState("");
3   const [password, setPassword] = useState("");
4   const [error, setError] = useState(false);
5   const [helperText, setHelperText] = useState('');
6
7   const handleSubmit = async (event) => {
8     event.preventDefault();
9     setError(false);
10    setHelperText('');
11    try {
12      const response = await
13        ↪ fetch('http://localhost:5163/api/Auth/login', {
14          method: "POST",
15          headers: {"Content-Type": "application/json"},
16          body: JSON.stringify({ Email: emailUsername, Password:
17            ↪ password })
18        });
19      if (response.ok) {
20        const data = await response.json();
21        navigate('/welcome');
22      } else {
23        setError(true);
24        setHelperText('Login failed. Please check your login.');
```

Listing 6.1: Key functionality of the SignIn component. Part 1.

## 6.1 Components

---

```
1   return (  
2     <div>  
3       <TextField  
4         label="Email or username"  
5         variant="outlined"  
6         value={emailUsername}  
7         onChange={(e) => setEmailUsername(e.target.value)}  
8         error={error}  
9       />  
10      <TextField  
11        label="Password"  
12        type="password"  
13        variant="outlined"  
14        value={password}  
15        onChange={(e) => setPassword(e.target.value)}  
16        error={error}  
17        helperText={helperText}  
18      />  
19      <Button onClick={handleSubmit}>Login</Button>  
20    </div>  
21  );  
22 }
```

**Listing 6.2:** Key functionality of the SignIn component. Part 2.

This application has a lot of components, varying in complexity, that I will be discussing in this section. Together, these components are used to create a complete page.

### 6.1.1 Register Hours

The Register Hours page is a crucial part of my application, created to allow employees to log their working hours. This page is composed of seven components, the components used can be seen in table 6.1 below.

## 6.1 Components

---

Component name
AdminMenu
DatePicker
LunchToggle
Project
TimePicker
WorkPackage
<b>TimeRegisterMain</b>

**Table 6.1:** List of components used to create the TimeRegisterPage

- The first component, **AdminMenu**, is exclusive to admin users and is accessible on most pages of the application. Further details about this menu is provided in Section 6.1.2 below.
- The second component, **DatePicker**, allows employees to select the date for registration.
- The third component, **LunchToggle**, is a toggle button for employees to select if they took a lunch break that day.
- The fourth component, **Project**, employees can select which project they worked on.
- The fifth component, **TimePicker**, is used for logging start and end times of the workday.
- the sixth component, **WorkPackage**, allows employees to select the specific work package they contributed to, if it's used by the company.
- the seventh component, **TimeRegisterMain**, integrates all the components above to build the complete page.

## 6.2 Pages

---

### 6.1.2 Admin Menu

Component name
AddEmployee
AddEmployeesProject
AddEmployeeWorkPackage
EditEmployee
EmployeeOverviewMain
ProjectDetail
ProjectsManage
WorkPackageDetail
WorkPackageManage

**Table 6.2:** List of components used for the Admin Role

## 6.2 Pages

Pages are structured as components that correspond to routes within the application. They are used for organizing the application into navigable sections, as can be seen in Listing 6.3, where the ‘AddEmployeePage’ component works as the interface for user interactions related to when an admin wants to register an employee.

```
1 import AddEmployee from "../Components/Admin/AddEmployee";
2
3 const AddEmployeePage = () => {
4   return (
5     <div>
6       <AddEmployee/>
7     </div>
8   )
9 }
10 export default AddEmployeePage;
```

**Listing 6.3:** The AddEmployeePage



## 6.3 HTTP-requests

---

### 6.3 HTTP-requests

The front-end of the application sends HTTP requests to the back-end, such as retrieving data and submitting updates. The Fetch API handles the HTTP requests, which provides an easy way to fetch resources from the database. Below, in Listing 6.4 is an example, that demonstrates on how I've added the implementation of an HTTP POST request to add an employee to a project.

```
1  const addEmployeeToProject = async (employeeId: string) => {
2    if (!projectId || addedEmployeeIds.has(employeeId)) return;
3
4    try {
5      const response = await
6        ↪ fetch(`http://localhost:5163/api/EmployeeProject/AddEmployee`,
7        ↪ {
8          method: 'POST',
9          headers: {
10             'Content-Type': 'application/json',
11           },
12          body: JSON.stringify({ projectId, employeeId }),
13        });
14      if (!response.ok) {
15        throw new Error('Failed to add employee to project');
16      }
17
18      setAddedEmployeeIds(new Set([...addedEmployeeIds, employeeId]));
19      setSnackbarOpen(true);
20
21    } catch (err) {
22      setError(err.message);
23    }
24  };
25
```

**Listing 6.4:** A POST HTTP-request to the back-end for adding an employee to a project.

## 6.4 Routing

---

## 6.4 Routing

This section demonstrates how to route data between the various endpoints and establishing a connections between them. To achieve this, React Hooks and React Router has been used.

### 6.4.1 React Hooks

React explains that hooks let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own. [React, nd].

Among the various hooks provided by React, the most commonly used once are:

**useState:** This hook lets you add state to components. For example, see Listing 6.5, which demonstrates initializing the hadLunch toggle state to false by default when employees enter the page where they register their hours.

```
1  const [hadLunch, setHadLunch] = React.useState(false);
```

**Listing 6.5:** useState hook initializing state to false.

**useParams:** is a hook that extracts parameters from the URL, such as IDs, allowing components to access variable parts of route paths, which can be seen in Listing 6.6, and is further discussed in Section 6.4.2.

```
1  const { employeeId } = useParams();
```

**Listing 6.6:** useParams hook that extracts the employeeId from the URL.

## 6.4 Routing

---

**useEffect:** allows you to perform side effects in function components, such as data fetching, and it runs after the render is committed.

```
1  useEffect(() => {
2    const fetchEmployee = async () => {
3      try {
4        const response = await fetch(`http://localhost:5163/api/Employee`
5          ↪ /EditEmployee/${employeeId}`,
6          ↪ {
7            headers: { 'Accept': 'application/json' },
8          });
9        if (!response.ok) throw new Error('Failed to fetch employee
10         ↪ data');
11        const data = await response.json();
12        setFormData({
13          email: data.email || '',
14          firstName: data.firstName || '',
15          lastName: data.lastName || '',
16          phoneNumber: data.phoneNumber.toString() || '',
17          role: Object.keys(roleMapping).find(key => roleMapping[key]
18            ↪ === data.employeeRole) || '',
19          status: data.status,
20        });
21      } catch (error) {
22        console.error(error.message);
23      }
24    };
25
26    fetchEmployee();
27  }, [employeeId]);
```

**Listing 6.7:** `useEffect` to fetch employee data based on a changing `employeeId`, and updating form state with the fetched data.

**useNavigate:** is a hook from that allows the user to navigate or change routes within the application.

## 6.4 Routing

---

```
1  const BackButton: React.FC<BackButtonProps> = () => {
2    const navigate = useNavigate();
3
4    const handleBack = () => {
5      navigate(-1);
6    };
};
```

**Listing 6.8:** useNavigate hook that is used to navigate back to the previous page.

**useLocation:** is a hook from React Router that provides access to the location object, which represents where the app is currently, allowing you to query and use information about the current URL in your component.

```
1  const location = useLocation();
2
3  const projectName = location.state?.projectName || 'Unknown Project';
```

**Listing 6.9:** useLocation hook retrieving a ‘projectName’ from the navigation state with a default value.

### 6.4.2 React Router

Since React lacks native page routing features, I’ve used React Router, to enable navigation between the different pages.

Among the various router components, BrowserRouter is the one utilized in this application. React Router describes the following on BrowserRouter: A `<BrowserRouter>` stores the current location in the browser’s address bar using clean URLs and navigates using the browser’s built-in history stack [BrowserRouter, nda]. Router enables Client Side Routing, which React Router provides the following explanation for: Client side routing allows your app to update the URL from a link click without making another request for another document from the server. Instead, your app can immediately render some new UI and make data requests with fetch to update

## 6.4 Routing

---

the page with new information [BrowserRouter, ndb]. See Listing 6.10 that shows some of the routes that are implemented.

## 6.4 Routing

---

```
1 interface ProtectedRouteProps {
2   children: ReactNode;
3 }
4
5 const ProtectedRoute: React.FC<ProtectedRouteProps> = ({ children }) => {
6   const { isAuthenticated } = useAuth();
7   return isAuthenticated ? <>{children}</> : <Navigate to="/" replace />;
8 }
9
10 function App() {
11   return (
12     <AuthProvider>
13       <Router>
14         <Routes>
15           <Route path="/" element={<HomePage />} />
16           <Route
17             path="/welcome"
18             element={
19               <ProtectedRoute>
20                 <WelcomePage />
21               </ProtectedRoute>
22             }
23           />
24           <Route
25             path="/registerhours"
26             element={
27               <ProtectedRoute>
28                 <TimeRegisterMain />
29               </ProtectedRoute>
30             }
31           />
32           <Route
33             path="/editemployee/:employeeId"
34             element={
35               <ProtectedRoute>
36                 <EditEmployeePage />
37               </ProtectedRoute>
38             }
39           />
```

Listing 6.10: Example of some of the routings used.

## 6.4 Routing

---

### ProtectedRoutes

As shown in Listing 6.10, the ‘ProtectedRoute’ component increases the application security by ensuring that only authenticated users can access specific routes, which is provided by the AuthContext component for authentication checks, which can be seen in the Listings 6.11 and Listing 6.12.

## 6.4 Routing

---

```
1 interface AuthContextType {
2   isAuthenticated: boolean;
3   isAdmin: boolean;
4   setIsAuthenticated: React.Dispatch<React.SetStateAction<boolean>>;
5   login: (role: string) => void;
6   logout: () => void;
7   setRole: (role: string) => void;
8 }
9 const AuthContext = createContext<AuthContextType | undefined>(undefined);
10
11 export const AuthProvider = ({ children }: { children: ReactNode }) => {
12   const [isAuthenticated, setIsAuthenticated] = useState(false);
13   const [role, setRole] = useState('');
14
15   const updateRole = useCallback((newRole: string) => {
16     setRole(newRole);
17     setIsAuthenticated(true);
18   }, []);
19
20   const login = useCallback((role: string) => {
21     updateRole(role);
22   }, [updateRole]);
23
24   const logout = useCallback(() => {
25     setIsAuthenticated(false);
26     setRole(''); // Clear the role on logout
27   }, []);
28
29   const isAdmin = role === 'Admin';
30 }
```

**Listing 6.11:** ‘AuthContext’ and ‘AuthProvider’ for managing authentication state.



## 6.4 Routing

---

```
1   return (
2     <AuthContext.Provider value={{ isAuthenticated, setIsAuthenticated,
3       ↪ isAdmin, login, logout, setRole: updateRole }}>
4       {children}
5     </AuthContext.Provider>
6   );
7
8   export const useAuth = () => {
9     const context = useContext(AuthContext);
10    if (context === undefined) {
11      throw new Error('useAuth must be used within an AuthProvider');
12    }
13    return context;
14  };
```

Listing 6.12: Continuation of the ‘AuthProvider‘.

# Chapter 7

## Results

In this chapter, I present the finished front-end of the application and I discuss the difference between the design from Chapter 3 and the finalized product.

### 7.1 Final design

Video demonstration of the application: <https://www.youtube.com/watch?v=LY3f17LMSmk>

A couple of clarifications in regards to the video:

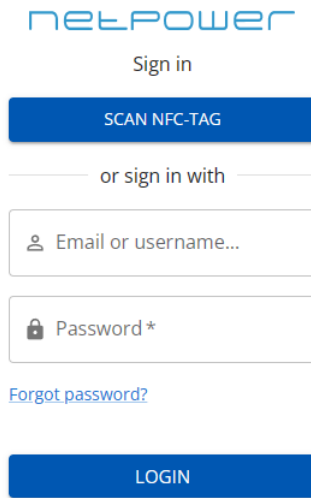
At the 1:12 mark in the video, an error appeared in the console log when the inspect window was opened. This error has not recurred since.

Regarding the repetition of my name when adding employees to projects or work packages, this arises from multiple registrations under the same name.

There are some differences between the design in Chapter 3 and the finalized application, this is due to a miscommunication under one of the meetings, so the designer used Material Design instead of Material UI in the design process.

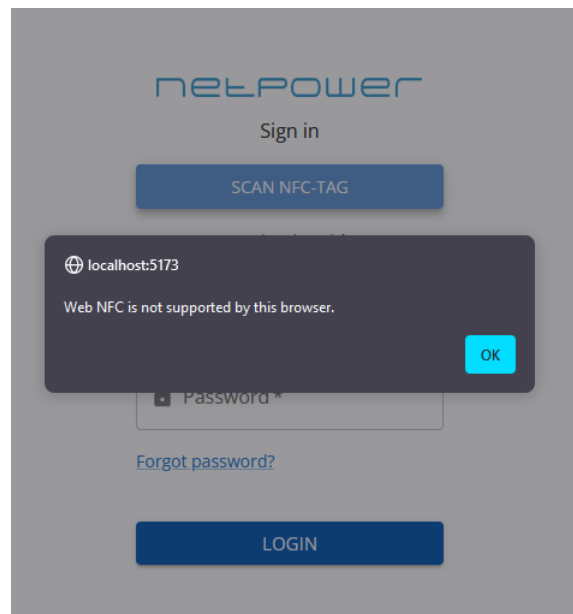
## 7.1 Final design

---



The screenshot shows the NetPower login interface. At the top is the 'netPOWER' logo in blue. Below it is the text 'Sign in'. A prominent blue button labeled 'SCAN NFC-TAG' is centered. Underneath, the text 'or sign in with' is flanked by horizontal lines. There are two input fields: the first is labeled 'Email or username...' with a person icon, and the second is labeled 'Password \*' with a lock icon. A blue link 'Forgot password?' is positioned below the password field. At the bottom is a large blue button labeled 'LOGIN'.

**Figure 7.1:** Login page



**Figure 7.2:** NFC-error because the browsers on my PC does not support NFC.

## 7.1 Final design

---

The screenshot shows the NetPower login interface. At the top is the 'netPOWER' logo. Below it is the text 'Sign in'. A blue button labeled 'SCAN NFC-TAG' is present. Below that is the text 'or sign in with'. There are two input fields: 'Email or username...' containing 'test@test.no' and 'Password\*' containing masked characters. A red error message 'Login failed. Please check your login.' is displayed below the password field. A blue link 'Forgot password?' is located below the error message. At the bottom is a blue button labeled 'LOGIN'.

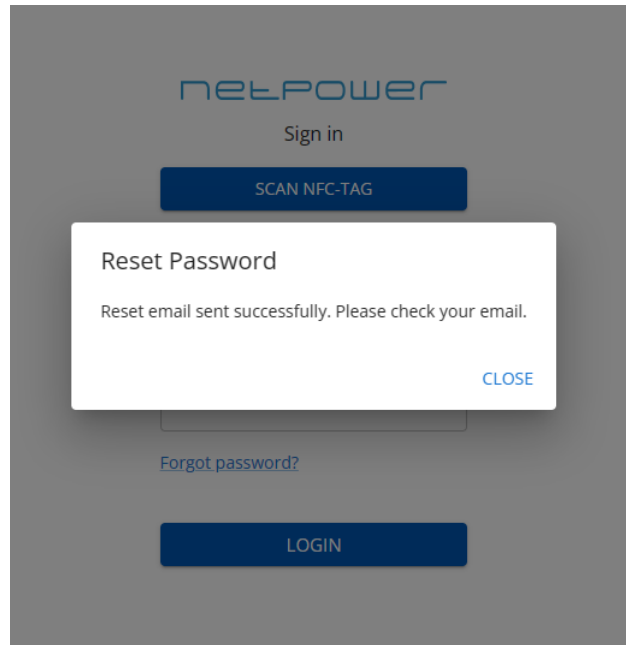
**Figure 7.3:** Login failed, shows error on both for security reasons.

The screenshot shows a 'Reset Password' dialog box overlaid on the login page. The dialog has a title 'Reset Password' and a message: 'To reset your password, please enter your email address here. We will send you a reset link.' Below the message is an input field labeled 'Email Address'. At the bottom right of the dialog are two buttons: 'CANCEL' and 'SEND'. The background login page is dimmed, showing the 'netPOWER' logo, 'Sign in' text, and a blue 'LOGIN' button.

**Figure 7.4:** Forgot password

## 7.1 Final design

---



**Figure 7.5:** Reset password email sent if email is found in the database



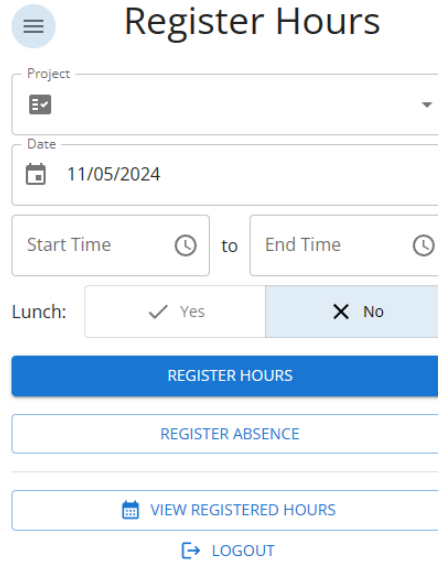
**Welcome, string!**

Redirecting you to the next page in a few seconds...

**Figure 7.6:** Welcome page after logging in, the "Welcome, string!" takes in the first name, so if a employee with the name "Anders" logs in it will be "Welcome, Anders!"

## 7.1 Final design

---



Register Hours

Project: [E] ✓

Date: 11/05/2024

Start Time [🕒] to End Time [🕒]

Lunch:  Yes  No

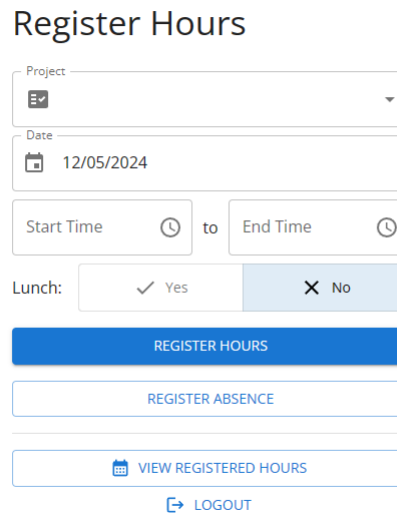
REGISTER HOURS

REGISTER ABSENCE

VIEW REGISTERED HOURS

LOGOUT

**Figure 7.7:** Register hours page with admin privileges(the admin menu is up in the left hand side corner)



Register Hours

Project: [E] ✓

Date: 12/05/2024

Start Time [🕒] to End Time [🕒]

Lunch:  Yes  No

REGISTER HOURS

REGISTER ABSENCE

VIEW REGISTERED HOURS

LOGOUT

**Figure 7.8:** Register hours page without the admin menu.

## 7.1 Final design

---

The screenshot shows the 'Register Hours' form with the following fields and options:

- Project:** Project A
- Work Project:** Pakke 2
- Date:** 11/05/2024
- Start Time:** [Clock icon] to **End Time:** [Clock icon]
- Lunch:**  Yes  No
- Buttons:** REGISTER HOURS (blue), REGISTER ABSENCE (light blue), VIEW REGISTERED HOURS (light blue with calendar icon), LOGOUT (light blue with arrow icon)

**Figure 7.9:** Work Package menu showing up because this employee has it available.

The screenshot shows the 'Register Hours' form with a date picker open over the 'Date' field. The date picker displays the month of May 2024, with the 11th of May selected. The form fields are the same as in Figure 7.9.

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

**Figure 7.10:** Date select on a PC browser.

## 7.1 Final design

---

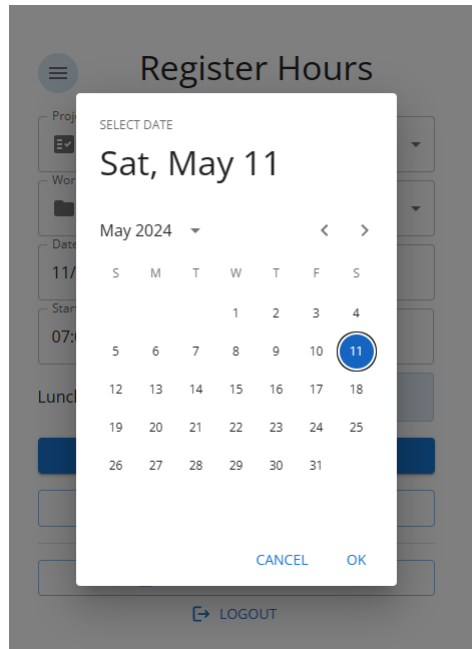


Figure 7.11: Date select on a phone.

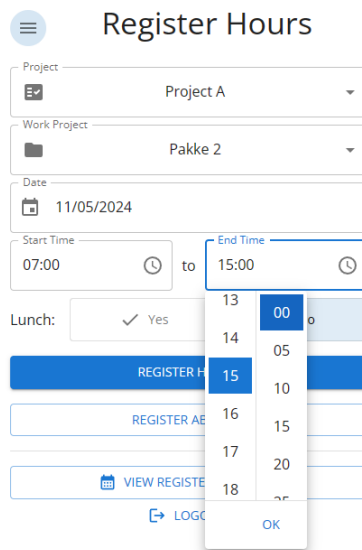
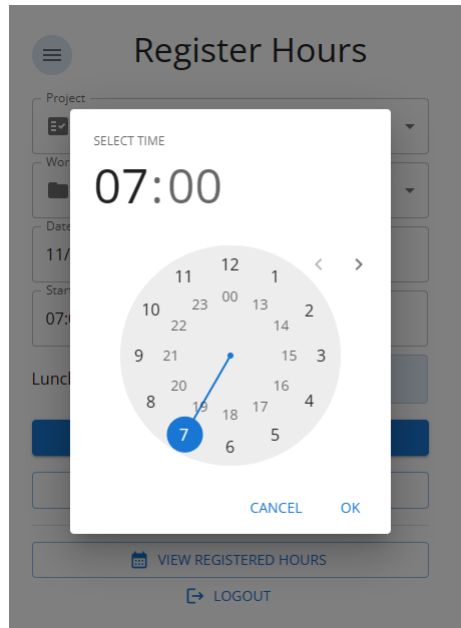


Figure 7.12: Clock select on a PC browser.

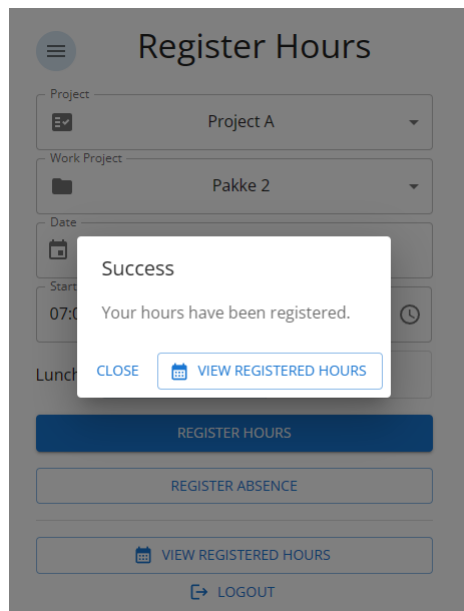


## 7.1 Final design

---



**Figure 7.13:** Clock select on a phone.



**Figure 7.14:** Hours registered.

## 7.1 Final design

---

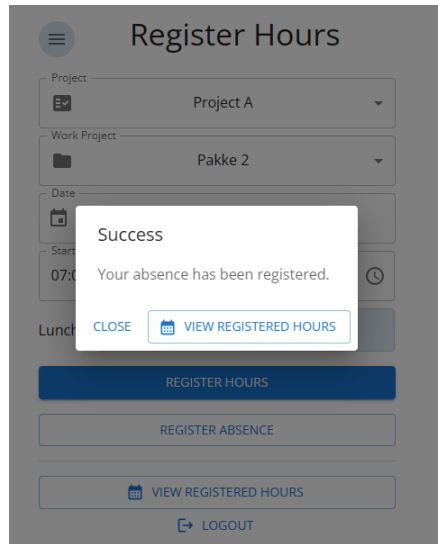


Figure 7.15: Absence registered.

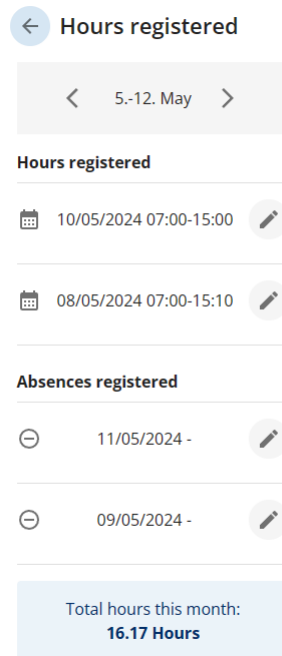


Figure 7.16: List of registered hours and absence.

## 7.1 Final design

---

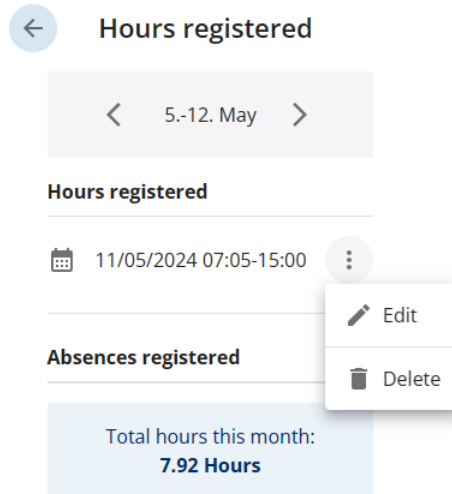


Figure 7.17: Edit or delete option for the hours registered.

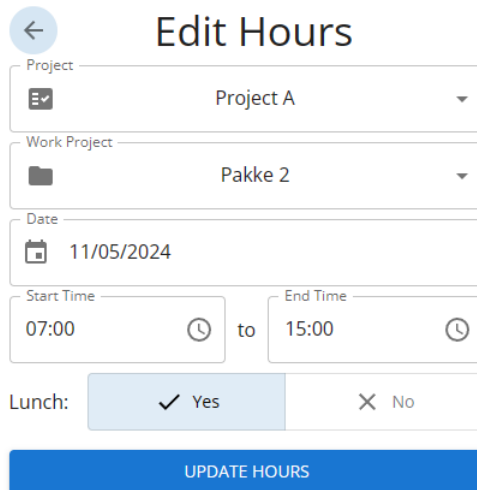
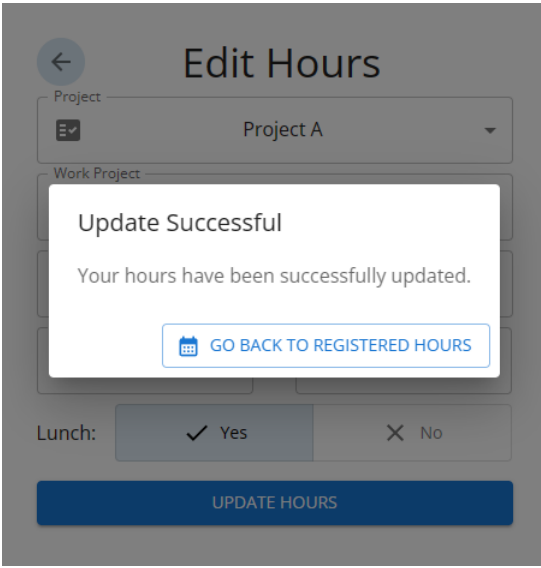


Figure 7.18: Editing the hours.

## 7.1 Final design

---



**Figure 7.19:** Editing hours successfully.

## 7.2 Admin Menu

---

## 7.2 Admin Menu

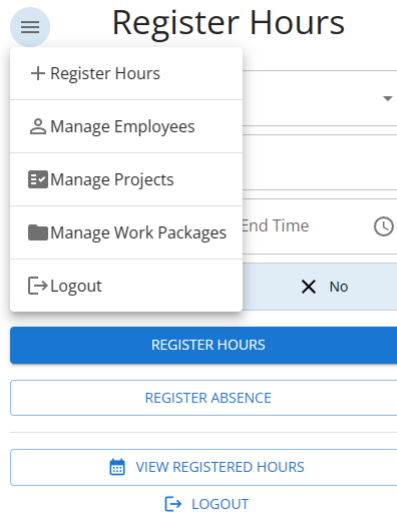


Figure 7.20: The admin menu

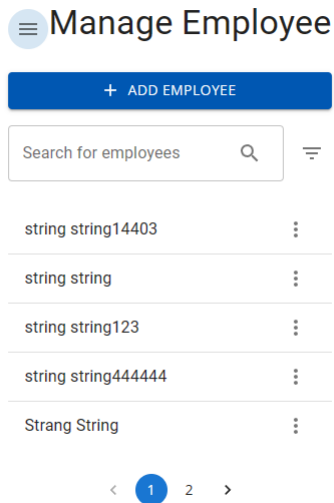
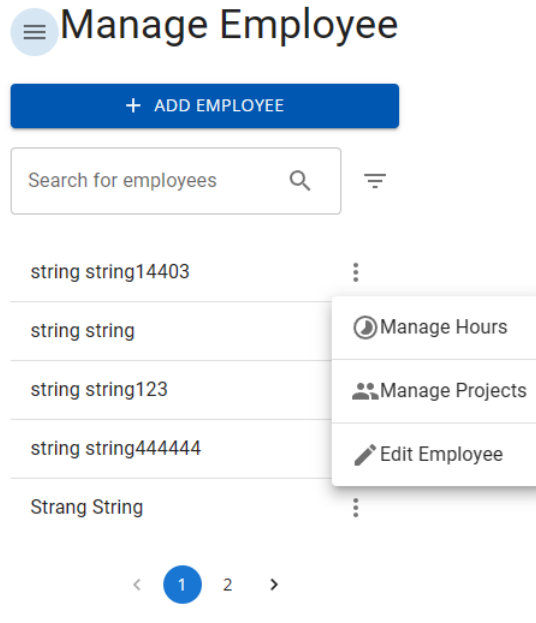


Figure 7.21: Overview of the employees.

## 7.2 Admin Menu

---




**Figure 7.22:** Options on employees.

The screenshot shows a form titled "Edit employee" with a back arrow icon. The form contains five input fields and one dropdown menu. The first field is "Email Address \*" with the value "stringstringstringstring@string.com". The second field is "First Name \*" with the value "string". The third field is "Last Name \*" with the value "string14403". The fourth field is "Phone Number \*" with the value "0". The fifth field is "Role" with a dropdown menu showing "Admin". At the bottom of the form is a blue button labeled "EDIT EMPLOYEE".

**Figure 7.23:** Editing employee.

## 7.2 Admin Menu

---

 Register new employee

Email Address \*

Password \*

Confirm Password \*

First Name \*

Last Name \*

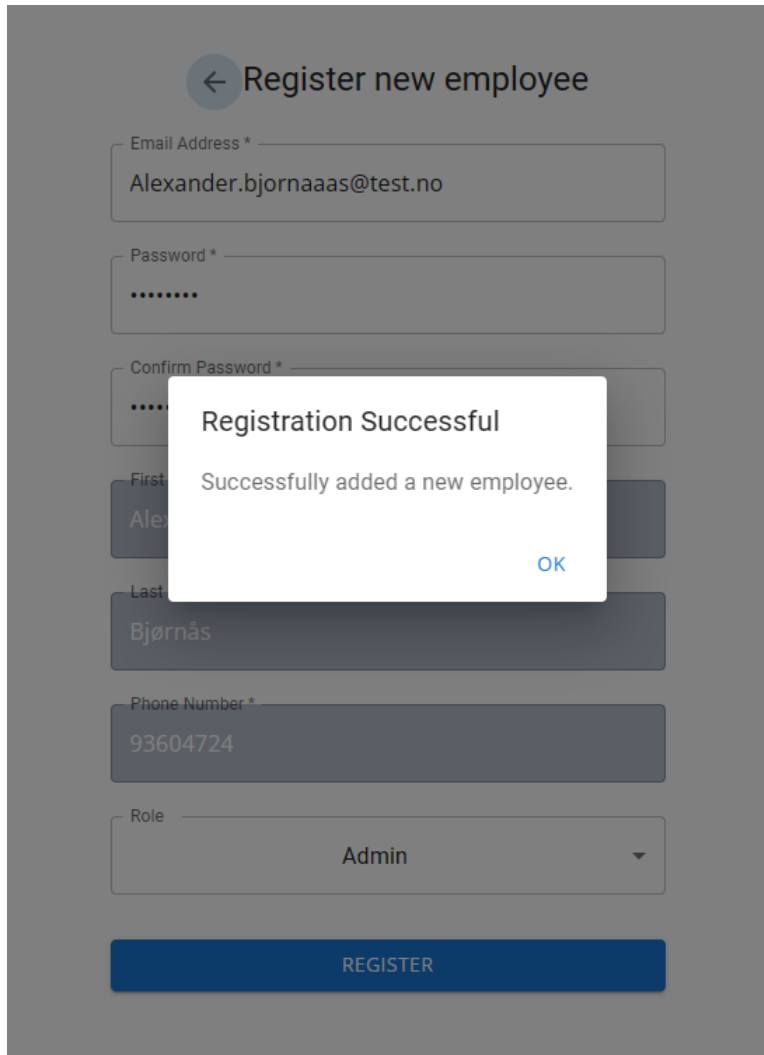
Phone Number \*

Role

**Figure 7.24:** Registering a new employee.

## 7.2 Admin Menu

---



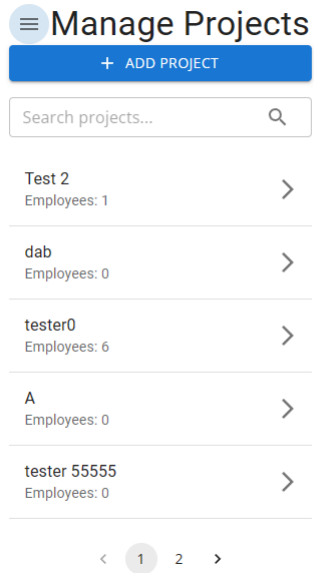
The image shows a mobile application interface for registering a new employee. The form is titled "Register new employee" and includes several input fields: "Email Address \*" (containing "Alexander.bjornaas@test.no"), "Password \*" (masked with dots), "Confirm Password \*" (masked with dots), "First Name" (containing "Ale"), "Last Name" (containing "Bjørnås"), "Phone Number \*" (containing "93604724"), and a "Role" dropdown menu (set to "Admin"). A large blue "REGISTER" button is at the bottom. A white modal dialog box is overlaid on the form, displaying the message "Registration Successful" and "Successfully added a new employee." with an "OK" button.

**Figure 7.25:** Successfully added a new employee.

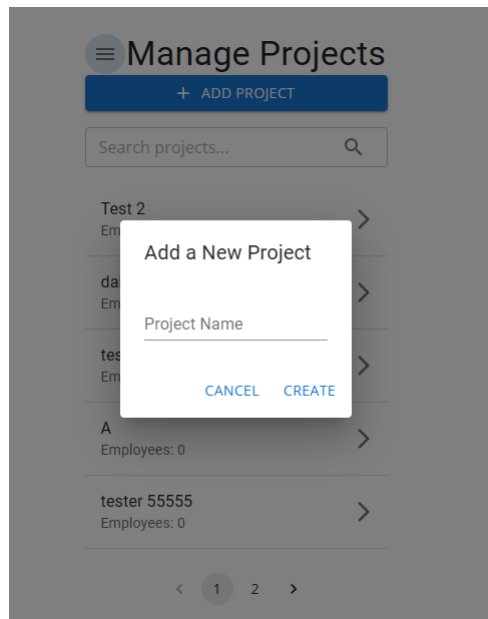


## 7.2 Admin Menu

---



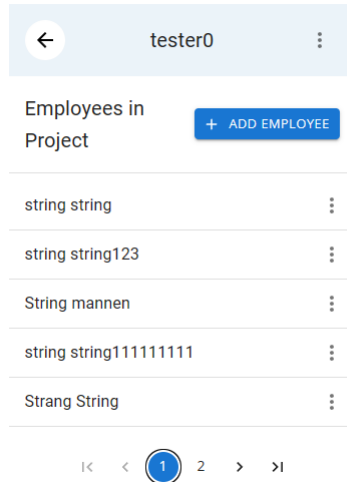
**Figure 7.26:** Projects overview.



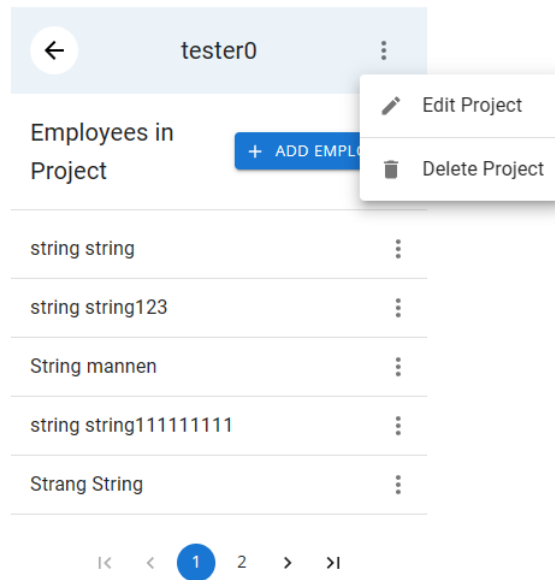
**Figure 7.27:** Adding a new project.

## 7.2 Admin Menu

---



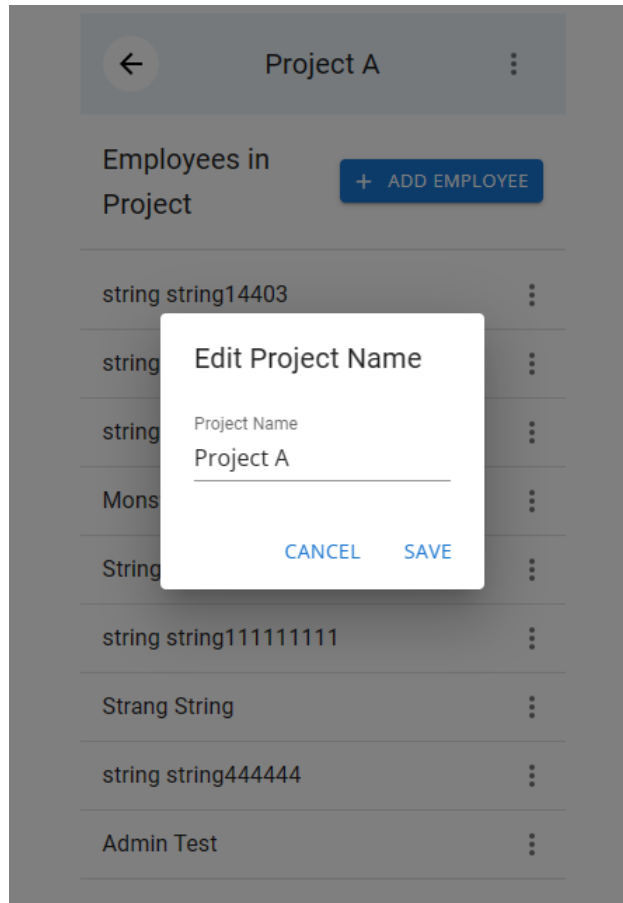
**Figure 7.28:** Managing a project.



**Figure 7.29:** Edit or delete options on the project.

## 7.2 Admin Menu

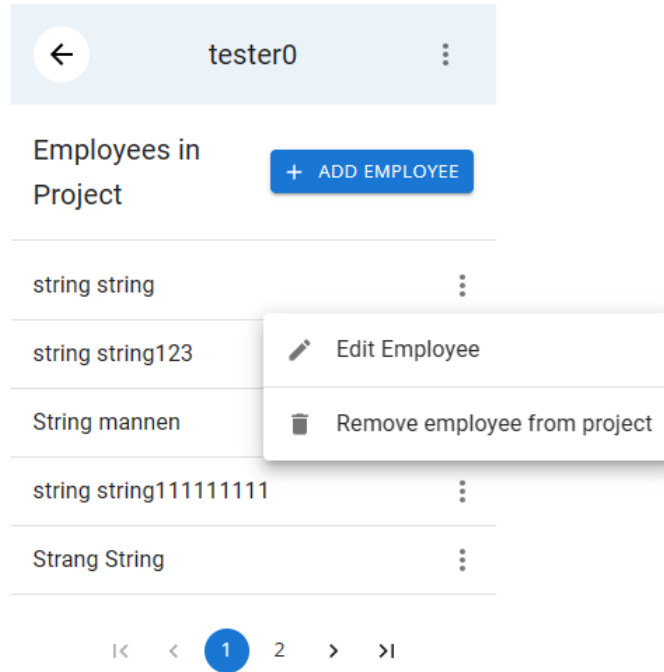
---



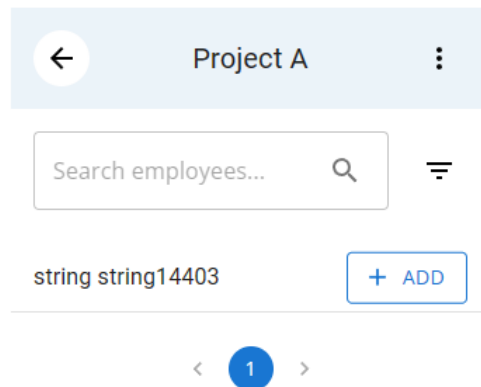
**Figure 7.30:** Editing the project

## 7.2 Admin Menu

---



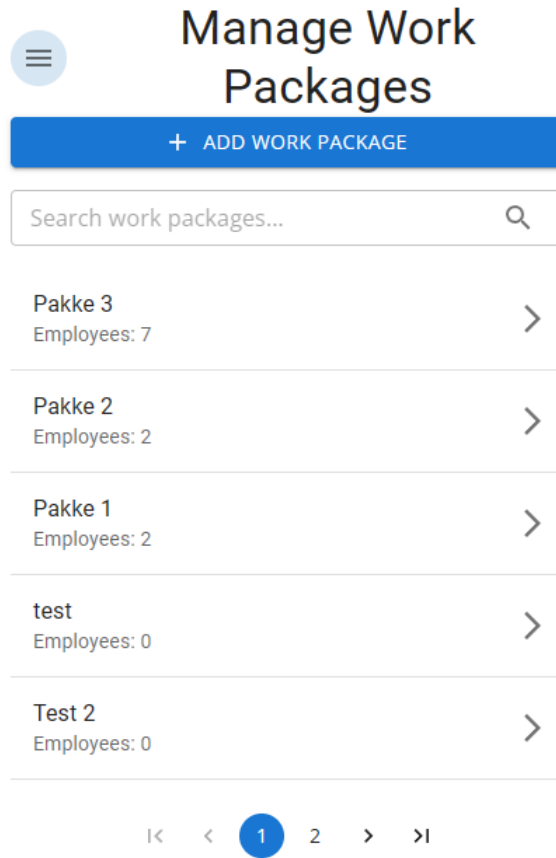
**Figure 7.31:** Options for employees in a project.



**Figure 7.32:** Adding an employee to a project.

## 7.2 Admin Menu

---



**Figure 7.33:** Work package overview.

The structure of the work package setup mirrors that of the project setup. It can be seen in the video at the beginning of the section 7.1.

## Chapter 8

# Discussion

In this chapter, I discuss what I would add or change in my application if I had more time. I talk about my ideas for making it better, fixing any issues the application has, and also adding new features.

### 8.1 Next Steps

This was a fairly large project for one person. Because of that, there were a few functionalities and ideas that I wanted to implement but did not have the time for. This opens up many possibilities for the future development of this project.

- **Disable or activate employees:** In Figure 3.10, it's shown that deleting an employee is an option. However, due to potential complications this could cause, it might be more practical to include a disable/activate button on the employee.
- **Approving or declining hours:** A feature that is work in progress, and can be seen in Figure 3.10 is the ability to review and manage an employees recorded hours. This functionality would allow administrators to approve, decline, or edit logged hours, and also provide employees with a status of their logged hours.

## 8.2 Looking Back

---

- **Project creation options:** Project creation only involves naming the project, However, it would be beneficial to include options for adding employees based on their roles or linking the project to a specific work package. Similarly, expanding the options available for work packages would enhance the overall functionality of the application.
- **More defined roles:** There are only three roles, Admin, Internal and External. Introducing more defined roles would simplify employees filtering, and the potential to implement more features based on what role they have.
- **Improved filtering:** Improving the filtering system would be beneficial for administrators to more effectively sort through employees, projects, or work packages.
- **See what projects or work packages an employee is on:** It should be possible to go into an employees projects or work packages to make it easier for an administrator to view and edit.
- **Dynamic branding:** Implement functionality to dynamically adjust the application's logo and color scheme according to the company using the application.
- **Dark Mode:** A feature we talked about was implementing dark mode in the application, offering users the flexibility to switch between light and dark modes for improved usability in various lighting conditions.

## 8.2 Looking Back

Due to my limited experience with projects of this scale, I did not have much experience with good coding practices and clean code principles. As a result of this, there are valuable lessons learned, and certain aspects I would have handled differently that I will discuss in this section.

- **Optimize the components:** Certain components could benefit from optimization to improve their functionality and efficiency.
- **Scaling issues:** Some pages in the application do not scale correctly with each other. This could be fixed by reworking some of the components as written above.

## 8.2 Looking Back

---

- **Tests:** With tests, it would have provided better documentation on how the code is running, increased maintainability, and offered the potential for faster development on the project.
- **Back-end rework:** Rewrite certain code segments and update pipelines that require attention.



## Chapter 9

# Conclusion

The goal of this thesis was to develop a standalone time register web application that does not require users to use SharePoint, Power Apps or other similar technologies which requires that the users have a Microsoft or guest account. Through using modern technologies such as React and Vite with MUI as the front-end, and ASP.NET Core 8 as the back-end, the application will have support and updates for a long time.

This application is designed with simplicity in mind, especially for regular users without admin privileges. By minimizing the number of pages, users can log in, record their hours, or view them without unnecessary complexity. While administrators have access to menus for making necessary adjustments or modifications to, for example employees or projects. This makes it user-friendly for all users, while equipping administrators with the tools they need for management and customization.

# Bibliography

- [Anderson, 2024] Anderson, R. (2024). Introduction to identity on asp.net core. Accessed on May 13, 2024. <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-8.0>.
- [BrowserRouter, nda] BrowserRouter (n.d.a). Browserrouter v6.23.0. Accessed on May 9, 2024. <https://reactrouter.com/en/main/router-components/browser-router>.
- [BrowserRouter, ndb] BrowserRouter (n.d.b). Feature overview. Accessed on May 10, 2024. <https://reactrouter.com/en/main/start/overview>.
- [Figma, nd] Figma (n.d.). What is figma? Accessed on April 19, 2024. <https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma>.
- [Larabee, 2016] Larabee, D. (2016). Best practice - an introduction to domain-driven design. Accessed on May 13, 2024. <https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design>.
- [Microsoft, 2023] Microsoft (2023). Overview of asp.net core. Accessed on January 26, 2024. <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0>.
- [Microsoft, nd] Microsoft (n.d.). Entity framework documentation hub. Accessed on May 13, 2024. <https://learn.microsoft.com/en-us/ef/>.

## BIBLIOGRAPHY

---

- [Miro, 2023] Miro (2023). What is miro? Accessed on April 19, 2024. <https://help.miro.com/hc/en-us/articles/360017730533-What-is-Miro>.
- [MUI, nd] MUI (n.d.). Overview. Accessed on January 21, 2024. <https://mui.com/material-ui/getting-started/>.
- [Netpower, nd] Netpower (n.d.). Om netpower. Accessed on January 18, 2024. <https://www.netpower.no/om-netpower/>.
- [React, nd] React (n.d.). Built-in react hooks – react. Accessed on May 8, 2024. <https://react.dev/reference/react/hooks>.
- [Vite, nda] Vite (n.d.a). Getting started. Accessed on January 26, 2024. <https://vitejs.dev/guide/>.
- [Vite, ndb] Vite (n.d.b). Why vite. Accessed on February 7, 2024. <https://vitejs.dev/guide/why>.
- [W3Schools, nd] W3Schools (n.d.). Typescript introduction. Accessed on May 13, 2024. [https://www.w3schools.com/typescript/typescript\\_intro.php](https://www.w3schools.com/typescript/typescript_intro.php).

## Attachments A

# Programlisting

For access to the source code, please contact Anders Endresen.

# Attachments B

## Data sheet

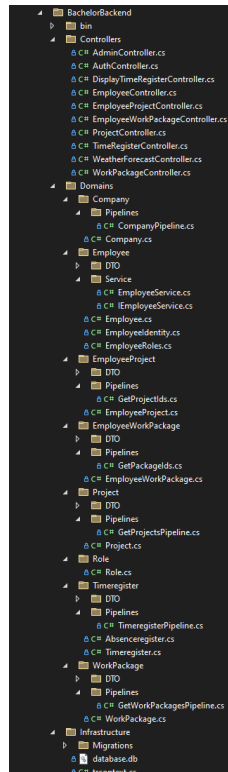


Figure B.1: Back-end folder structure

## Data sheet

---

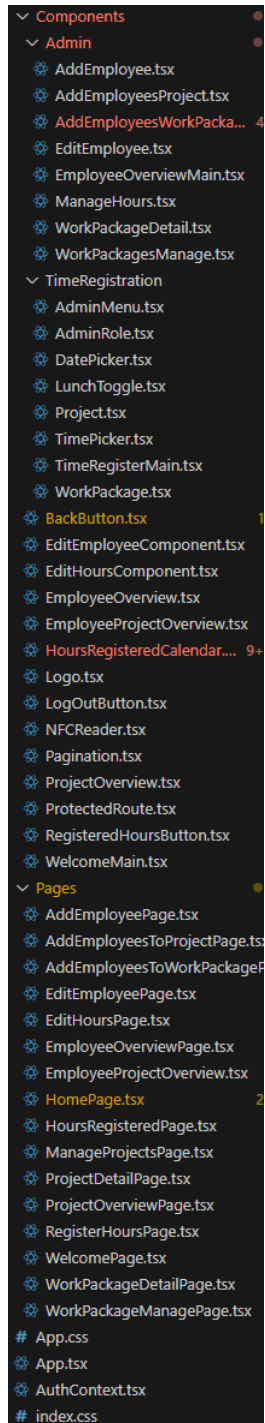


Figure B.2: front-end folder structure