



VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Mai Nguyen Tan Phat

Developing a Timesheet Application

Technology and Communication
2024

ABSTRACT

Author	Phat Mai
Title	Developing a Timesheet Application
Year	2024
Language	English
Pages	53
Name of Supervisor	Kenneth Norrgård

This thesis presents the development of Mercury 2.0, an updated version of the employee working hours management system originally developed by Indevit Ab Oy over a decade ago. The project aimed to improve user experience and functionality by redesigning the interface and adding new features.

The project was designed and developed using Razor, Blazor, C#, HTML/CSS and Fusion. Backend and frontend part were developed at the same time to ensure they worked well together. Developer also constantly tested the website to catch and fix any errors or bugs that could affect the rest of the development process.

The final application is fully functional and easy to use, allow employee to input, edit, delete and track their working hour.

Keywords
tion

Software engineering, Fusion, timesheet, C#, web applica-

CONTENTS

ABSTRACT

1	INTRODUCTION	8
2	BACKGROUND AND PROJECT PURPOSE	9
3	WORK PROCESS	11
4	THEORETICAL BACKGROUND	12
4.1	Backend.....	12
4.2	.NET ASP.NET Fusion	12
4.3	Frontend.....	13
4.4	RAZOR BLAZOR HTML CSS	13
4.5	DATABASE	14
4.5.1	MICROSOFT SQL SERVER.....	14
4.5.2	MS SQL MANAGEMENT STUDIO	14
4.6	VISUAL STUDIO 2022	15
4.7	INSTALLing NuGet Packages	15
5	IMPLEMENTATION.....	19
5.1	Project Structure	19
5.2	Backend.....	20
5.2.1	Backend.Contracts	20
5.2.2	BackEnd.Core	22
5.2.3	Backend.Host	26
5.3	Frontend.....	27
5.3.1	Css	28
5.3.2	Pages	29
5.3.3	Smart Components	31
5.3.4	Migrations	43
5.4	Deployment.....	43
5.4.1	IIS.....	44

5.4.2 Azure Pipeline	45
6. OUTCOME OF PROJECT	47
7. CONCLUSIONS	50
REFERENCES	52

LIST OF FIGURES AND TABLES

Figure 1 Microsoft SQL Server	14
Figure 2 Microsoft Visual Studio (C# IDE)	15
Figure 3 Project Structure	19
Figure 4 The "Backend.Contracts"	21
Figure 5 The "ICustomerService" file	21
Figure 6 The "DbEffort" file.....	23
Figure 7 Effort entity in Database	24
Figure 8 The "EffortService" file.....	24
Figure 9 The "UserService" file	26
Figure 10 Backend.Host files tree	26
Figure 11 The "Frontend" files tree	27
Figure 12 The "Css" files tree	28
Figure 13 Part of css file	28
Figure 14 The "Reset.css" file.....	29
Figure 15 The "Pages" files tree	29
Figure 16 The "Home.razor" file	30
Figure 17 All component that include in "Home.razor" file display in the browser	31
Figure 18 The "Calendar.razor" file.....	32
Figure 19 The "Calendar.razor" file.....	32
Figure 20 The "calendar.razor" display in the browser	33
Figure 21 The "Calendar.razor.cs" file	34
Figure 22 The "Calendar.razor.cs" file	34
Figure 23 The "CreateEffort.razor" file	35
Figure 24 This is how "CreateEffort" file	35
Figure 25 The "CreateEffort.razor.cs" file.....	36
Figure 26 EditDeleteModal.razor file.....	37
Figure 27 The "EditDeleteModal.razor.cs" file	38
Figure 28 EffortList.razor	39

Figure 29 The "EffortList.razor" file	40
Figure 30 The "EffortList.razor.cs" file	41
Figure 31 GetEffortsByDateAndCustomer method	42
Figure 32 Azure pipeline system	44
Figure 33 Azure pipelines setting.....	45
Figure 34 Login page	47
Figure 35 Mercury-timesheet page	48
Figure 36 EditDeleteModel popup.....	49
Table 1: Smart frontend convert case	36

1 INTRODUCTION

During my internship at Indevit Ab Oy, I gained practical experience and valuable insights into the world of web development. Throughout this period, I was exposed to a variety of technologies, including HTML, CSS, Gatsby, React, and C#. A significant part of my internship involved developing a web application to help employees track their working hours efficiently. This project provided me with hands-on experience in building web applications from scratch, implementing key functionalities, and ensuring user-friendly interfaces.

In addition to mastering technical skills such as coding and debugging, this experience taught me the importance of adaptability and continuous learning in the fast-paced field of software development. I learned to tackle challenges head-on, think critically, and find innovative solutions to complex problems. My internship at Indevit Ab Oy not only enhanced my technical proficiency but also equipped me with invaluable practical knowledge and problem-solving abilities.

Mercury, a software developed more than 10 years ago by the founder of Indevit, required further development due to the rapid advancements in society and technology. The old Mercury had several problems: the interface was difficult to see, the arrangement of features was not convenient, it was challenging to operate, it was difficult to edit tasks if entered incorrectly, and there was no feature to track total working hours in a day or for a customer. Therefore, Indevit needed a new application for hour recording. During my internship, my task was to develop this new application, and I also chose this project for my thesis.

These experiences have significantly contributed to the research and development of my thesis, providing me with a solid foundation to explore and address real-world challenges in the field of software engineering.

2 BACKGROUND AND PROJECT PURPOSE

Indevit is a Finnish company specializing in providing advanced IT solutions and services for various industries. With a strong focus on digital transformation and innovation, Indevit aims to empower businesses by leveraging cutting-edge technology.

Founded in 2010, Indevit has grown to become a trusted partner for organizations seeking to optimize their operations and enhance their competitiveness in the digital age. The company offers a wide range of services, including software development, IT consulting, cloud services, and system integration.

Indevit's team consists of highly skilled professionals with expertise in areas such as software development, cloud computing, cybersecurity, and data analytics. By combining industry knowledge with technical proficiency, Indevit delivers tailored solutions that address the unique challenges and objectives of each client.

The old Mercury had the following problems: the old interface was difficult to see, the arrangement of features was not convenient, it was difficult to operate, it was difficult to edit tasks if entered incorrectly, there was no feature of total working hours in a day, no features total working hours for a customer.

To solve these problems, Mercury 2.0 has the same functions as the old Mercury, such as entering working hours (effort), which include work description, working time, date, customer project, project task, and task category. It also allows users to edit hours, delete hours, and sort days by week or by day more efficiently and easily. Moreover, it comes with many upgrades: users can now choose the week to display by selecting the week in the calendar, it shows the total hours of the week, total hours in a day, and total hours that an employee has worked on a specific customer project. It will also auto-update whenever a user adds or deletes working hours.

In addition to the existing functions, Mercury 2.0 also has a better and easier user interface than the old version. Users do not need to go through as many steps to add or edit hours, making it easier to track working hours. Mercury 2.0 uses Azure for login, they also integrate with Azure to send tasks or assign tasks separately to employees, making it easier for both managers and employees to handle task management efficiently.

3 WORK PROCESS

I collaborated closely with the Indevit IT team, which included senior software engineers and the Chief Technology Officer (CTO), who served as my supervisor. The work followed a structured timetable using Azure DevOps as the project management tool, adhering to Agile methodology. While I primarily worked independently on the Mercury 2.0 project, I regularly participated in weekly meetings with the Indevit team to report progress to both the CEO and CTO.

The most challenging aspect of the project was ensuring the smooth flow of application data between components, which was crucial for maintaining accurate timesheet records. The Mercury 2.0 underwent testing by my supervisor, my teammates, and other Indevit employees.

According to feedback from my supervisor (CTO), the application is free of bugs, with all functions operating perfectly. They also praised the new user interface, describing it as aesthetically pleasing, user-friendly, and ready for deployment. Additionally, another employee commented on the smooth performance of the web application, highlighting that all functions work flawlessly. They also appreciated the attractive user interface and its ease of use.

4 THEORETICAL BACKGROUND

4.1 Backend

Backend is a server-side of the web application, which is responsible for managing and processing data, as well as handling requests from the client-side (front end). In a typical web application, the backend includes the server, application logic, database, and any other components necessary to support the functionality of the application.

4.2 .NET ASP.NET Fusion

.NET is a software development framework developed by Microsoft. It provides a platform for building, deploying, and running applications across various operating systems, devices, and architectures. NET supports various programming languages, including C#, Visual Basic .NET, and F#. It offers a wide range of functionalities such as web development, desktop application development, mobile application development, cloud-based services, and more. Additionally, .NET includes a large class library known as the .NET Framework Class Library (FCL), which provides reusable code for common programming tasks. (Meghan, 2023)

ASP.NET is a web application framework developed by Microsoft for building dynamic web pages and web applications. It is an extension of the .NET framework and allows developers to use languages such as C# or Visual Basic to create web applications. It provides many of feature and tools for web applications: Web form, ASP.NET MVC (Model-View-Controller), ASP.NET Web API, ASP.NET Core, ASP.NET Razor Pages. (Rick, 2017)

Fusion is a .NET library that implements Distributed REActive Memoization (DREAM), which is a novel abstraction designed to address various challenges encountered in building real-time web applications. It's somewhat similar to other state management libraries like MobX or Flux but is tailored to handle large states

across backend microservices, API servers, and client applications. (Alexyakunin, 2020)

4.3 Frontend

Frontend is an interface of the webpage or application that client interact with. If client have any action or request, the frontend will send it to the backend. Backend will resolve and send it result back and display it in the frontend. For example: when client want to view the shoe and click on it, the browser will pop up the shoe that client want to see.

4.4 RAZOR BLAZOR HTML CSS

RAZOR is a markup syntax used in ASP.NET, a web application framework developed by Microsoft. RAZOR allows developers to embed server-based code (C# or VB.NET) directly into HTML markup, making it easier to create dynamic web pages. It's particularly popular for building web applications with the ASP.NET MVC (Model-View-Controller) architecture. (Mike, 2018)

Blazor is a modern front-end web framework based on HTML, CSS, and C# that helps you build web apps faster. With Blazor, build web apps using reusable components that can be run from both the client and the server so that you can deliver great web experiences. (Microsoft, 2019)

HTML (HyperText Markup Language) is the standard markup language used to create and design documents on the World Wide Web. It's the foundation of web pages and web applications and provides the structure and layout for the content they display. (Wikipedia, 2001)

CSS (Cascading Style Sheets) is a fundamental component of web development, used to control the layout, formatting, and appearance of web pages. It allows developers to separate content from presentation, making it easier to create consistent and visually appealing websites. (W3Schools, 2013)

4.5 DATABASE

A database is an organized collection of structured information or data, typically stored electronically in a computer system. It is designed to efficiently manage, store, retrieve, and manipulate data according to specific requirements. Small databases can be stored on a file system, while large databases are hosted on computer clusters or cloud storage. (Wikipedia, 2022)

4.5.1 MICROSOFT SQL SERVER

Microsoft SQL Server is a relational database management system (RDBMS). Applications and tools connect to a SQL Server instance or database and communicate using Transact-SQL (T-SQL). (Wikipedia, 2022)



Figure 1 Microsoft SQL Server

4.5.2 MS SQL MANAGEMENT STUDIO

SQL Server Management Studio (SSMS) is an integrated environment for managing any SQL infrastructure, from SQL Server to Azure SQL Database. SSMS provides

tools to configure, monitor, and administer instances of SQL Server and databases. (Microsoft, 2018)

4.6 VISUAL STUDIO 2022

Visual Studio is a powerful developer tool that you can use to complete the entire development cycle in one place. It is a comprehensive integrated development environment (IDE) that you can use to write, edit, debug, and build code, and then deploy your app. Beyond code editing and debugging, Visual Studio includes compilers, code completion tools, source control, extensions, and many more features to enhance every stage of the software development process. (Microsoft, 2016)



Figure 2 Microsoft Visual Studio (C# IDE)

4.7 INSTALLING NuGet Packages

Microsoft.AspNetCore.Components.WebAssembly is a framework that enables to the building of interactive web applications using C# and .NET in the client-side browser environment.

Microsoft.AspNetCore.Authentication.MicrosoftAccount is a middleware that allows to the user authentication in ASP.NET Core applications using Microsoft accounts.

Microsoft.AspNetCore.Components.WebAssembly.DevServer is a development server that enables the debugging and testing of Blazor WebAssembly applications during development.

Microsoft.AspNetCore.Authentication.Google simplifies the process of implementing Google authentication in ASP.NET Core applications, providing a secure and user-friendly login experience.

Microsoft.AspNetCore.Authentication.Cookies: is essential for implementing cookie-based authentication and session management in ASP.NET Core applications, providing a secure and convenient way to authenticate users and manage their sessions.

Microsoft.AspNetCore.Components.WebAssembly.Server is a package that allows the hosting of Blazor WebAssembly applications on the server side.

Microsoft.Authentication.WebAssembly.Msal is a library that enables authentication in Blazor WebAssembly applications using Microsoft Authentication Library (MSAL).

Microsoft.EntityFrameworkCore.Sqlite is a library that enables SQLite database support in Entity Framework Core applications.

Microsoft.EntityFrameworkCore.SqlServer is a library that enables SQL Server database support in Entity Framework Core applications.

Microsoft.Extensions.Logging.Debug is a logging provider for .NET applications that writes log messages to the debug output window in Visual Studio.

Microsoft.Identity.Web is a library that simplifies the process of adding authentication and authorization to ASP.NET Core web applications using Azure Active Directory (Azure AD) and Microsoft identity platform.

Microsoft.NET.ILLink.Tasks is a NuGet package that provides tasks for the ILLink linker, which is used to perform link-time optimization (LTO) in .NET applications.

Microsoft.NET.Sdk.WebAssembly.Pack is a package that provides tools for packaging and publishing Blazor WebAssembly applications.

Stl.Fusion is a library that provides tools for building highly responsive and scalable .NET web applications.

Stl.Fusion.Blazor is a library that extends the capabilities of Blazor applications by providing tools for building highly responsive and scalable user interfaces.

Stl.Fusion.Blazor.Authentication is a library that extends the capabilities of Blazor applications by providing tools for integrating authentication and authorization functionalities.

Stl.Fusion.EntityFramework.Redis is a library that extends the capabilities of Entity Framework Core (EF Core) by providing integration with Redis for caching and distributed data synchronization.

Stl.Fusion.Ext.Contracts is a library that provides contract interfaces and attributes for defining and managing operations in Stl.Fusion-based applications.

Stl.Fusion.Ext.Services is a library that extends the capabilities of Stl.Fusion by providing additional services and utilities for building highly responsive and scalable .NET applications.

Stl.Fusion.Server is a library that extends the capabilities of Stl.Fusion by providing server-side components and utilities for building highly responsive and scalable .NET applications.

Stl.Generators is a library that provides utilities for generating code, such as source code files, during the build process of .NET applications.

Stl.Interception is a library that provides tools for intercepting method calls in .NET applications.

UAParser is a library that provides user-agent parsing functionality in .NET applications.

5 IMPLEMENTATION

5.1 Project Structure

The project includes 5 folders: `_`, Backend, Frontend, Migrations, and Solution Items, but the focus in this project is on Backend, Frontend, and Migrations.

Figure 3 will show the Project structure.

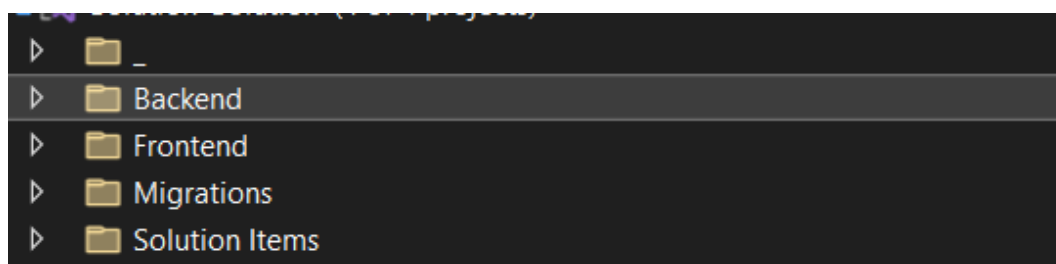


Figure 3 Project Structure

The Backend folder contains three important components:

- Service Interface defines how different parts of the application communicate with each other. It ensures smooth interaction between various services.
- Application Service is where the logic of the application lives. It handles all the behind-the-scenes operations, making sure everything works as it should.
- Hosting is responsible for deploying and managing the application, ensuring it's available and can handle as many users as needed.

The front ends contain many components that are used for creating and designing applications on the browser. Migrations ensure that the database structure evolves with the application.

5.2 Backend

Backend ("Server side") have 3 sub-folders: Backend.Contracts, Backend.Core, Backend.Host

- Backend.Contract contain interfaces defined for the backend functionality of an application.
- Backend.Core refers to the core implementation of the backend functionality. It contains the concrete implementations of the interfaces defined in Backend.Contracts.
- Backend.Host is the hosting or integration layer for the backend services. In Fusion and .NET development, this might involve components responsible for hosting backend services, such as API endpoints, or web services.

5.2.1 Backend.Contracts

In Fusion, backend.contracts is a namespace that typically contains interfaces defining contracts for backend services or functionalities. These contracts act as a set of rules that define what a particular service is supposed to do.

When working with Fusion and C#, backend.contracts would contain interfaces that define the communication protocols between the frontend and backend of an application. These interfaces help in decoupling the frontend and backend code, making the application more modular and easier to maintain.

Figure 4 displays all the file in the Backend.Contract

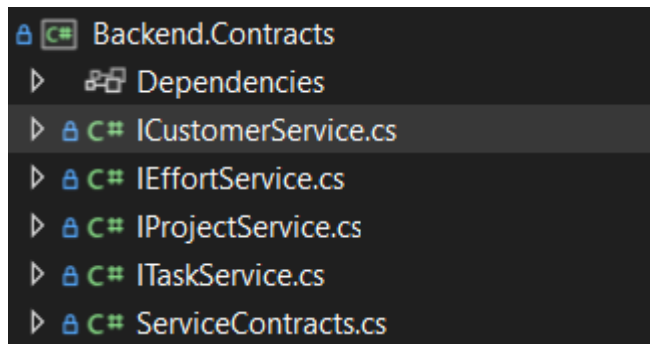


Figure 4 The "Backend.Contracts"

There are many files and it all the interface for the service in the Backend.Core for example: ICustomerService, IEffortService, IProjectService, ITaskService

All service interfaces have different methods and structures, depending on what the developer wants in that specific service interface. For the purpose of explanation, I have chosen the ICustomerService.

Figure 5 illustrates the ICustomerService which has the record type name "Customer" with list of the constructor include: CustomerId, CustomerName

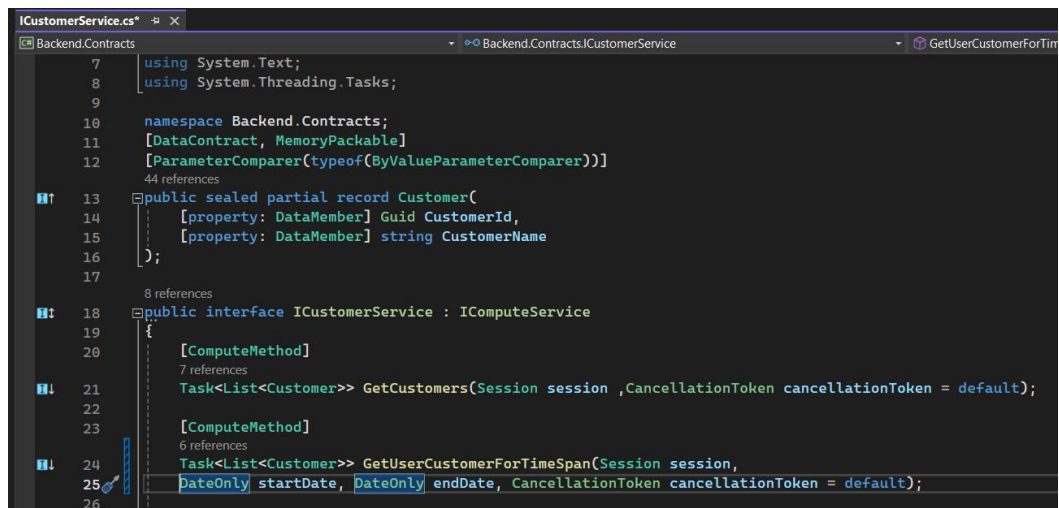


Figure 5 The "ICustomerService" file

The code in Figure 5 essentially provides a structure for managing customer data in a Blazor application, including defining the customer data structure (Customer) and methods to interact with this data (ICustomerService interface)

ICustomerService is an interface that provides methods for retrieving customer data. These methods are decorated with [ComputeMethod] attribute from Stl.Fusion, indicating that the results of these methods will be automatically cached and invalidated when necessary. This helps in optimizing the performance of the application by caching the results and reducing the number of calls to the server.

The ICustomerService interface serves as a blueprint for all CustomerService implementations. It ensures that each method in CustomerService follows the same structure defined in this interface. For example, the ICustomerService interface includes a method called GetCustomers which requires a session and a cancellationToken to execute, and a GetUserCustomerForTimeSpan method which requires a session, startDate, endDate, and a cancellationToken to execute. Any class implementing ICustomerService must provide an implementation for the GetCustomers method that accepts these parameters.

For services like IEffortService, ITaskService, and IProjectService, the workflow would be the same as ICustomerService.

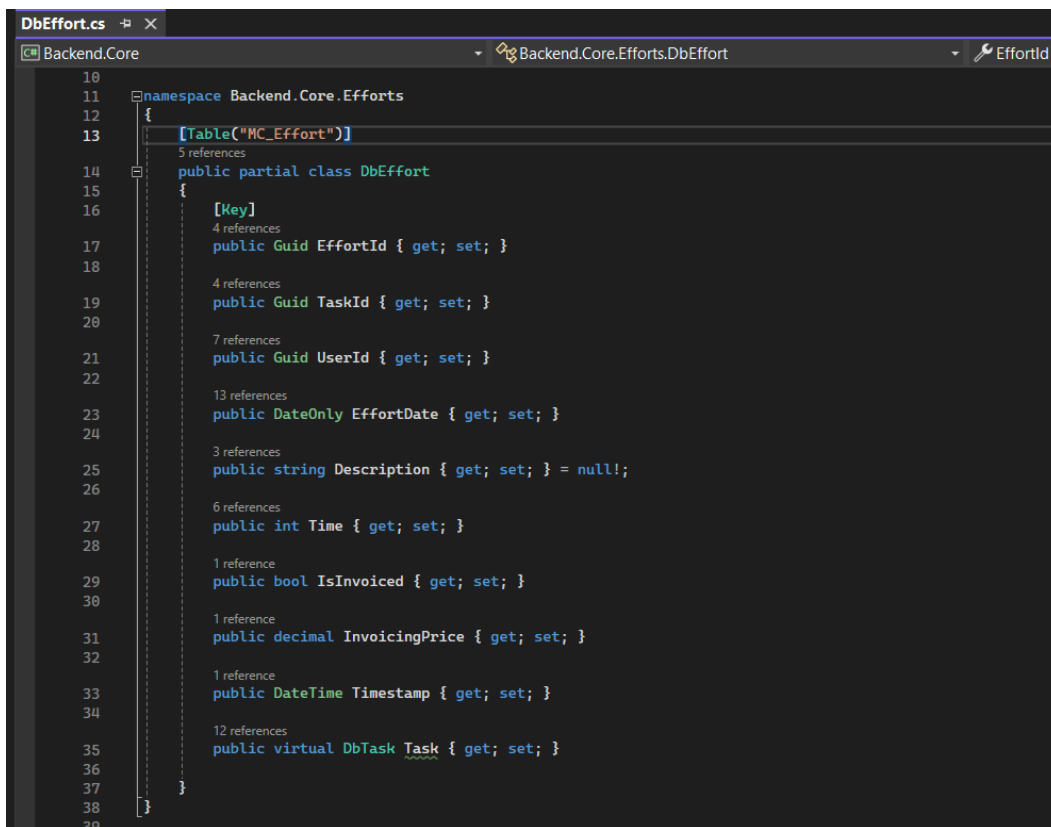
Each interface would define methods that specify the functionality related to efforts, tasks, and projects.

5.2.2 BackEnd.Core

The core functionality of the backend in a Fusion application includes the implementation of business logic, data access logic, and other essential components.

In this web application, developers use data models such as Category, Task, Customer, Effort, and Project. All entities within these models must be defined to correspond with the database schema.

Figure 6 show the model entities in the implementation part as an example.



```
10
11 namespace Backend.Core.Efforts
12 {
13     [Table("MC_Effort")]
14     public partial class DbEffort
15     {
16         [Key]
17         public Guid EffortId { get; set; }
18
19         public Guid TaskId { get; set; }
20
21         public Guid UserId { get; set; }
22
23         public DateOnly EffortDate { get; set; }
24
25         public string Description { get; set; } = null!;
26
27         public int Time { get; set; }
28
29         public bool IsInvoiced { get; set; }
30
31         public decimal InvoicingPrice { get; set; }
32
33         public DateTime Timestamp { get; set; }
34
35         public virtual DbTask Task { get; set; }
36     }
37 }
38
39
```

Figure 6 The "DbEffort" file

The DbEffort class creates a blueprint for storing information about these efforts in a database. Each effort has a unique identifier (EffortId), and might be linked to specific tasks (TaskId) and people (UserId) involved. It also tracks the date (EffortDate), a description (Description), and the time spent (Time). The system can also manage invoicing by keeping track of whether the effort has been billed (IsInvoiced) and potentially the associated price (InvoicingPrice). Finally, it records when the effort was logged (Timestamp) and allows linking it to a specific task within the project (Task).

An example of the Effort entity in Databa is shown in Figure 7.

	Column Name	Data Type	Allow Nulls
▶	EffortId	uniqueidentifier	<input type="checkbox"/>
	TaskId	uniqueidentifier	<input type="checkbox"/>
	UserId	uniqueidentifier	<input type="checkbox"/>
	EffortDate	date	<input type="checkbox"/>
	Description	nvarchar(200)	<input type="checkbox"/>
	Time	int	<input type="checkbox"/>
	IsInvoiced	bit	<input type="checkbox"/>
	InvoicingPrice	decimal(14, 2)	<input type="checkbox"/>
	Timestamp	datetime	<input type="checkbox"/>
			<input type="checkbox"/>

Figure 7 Effort entity in Database

To ensure that the application can correctly send and retrieve data from the database, we need to make sure that the data types match the entity data types.

After defining data entity, service implementation is the next part for that model, in this case is EffortService.

Figure 8 is an example of EffortService.

```

EffortService.cs*
Backend.Core
16
17
18 namespace Backend.Core.Todos;
19
20 public class EffortService(IDbContextFactory<AppDbContext> dbFactory, UserService users) : IEffortService
21 {
22
23     5 references
24     public virtual async Task<List<Guid>> GetCustomerIdsWithEffortsDuring(Session session,
25         DateOnly startDate, DateOnly endDate, CancellationToken cancellationToken)
26     {
27         var db = await dbFactory.CreateDbContextAsync();
28         MercuryUser user = await users.GetCurrentUser(session, cancellationToken);
29         var customerIdsWithEffort = await db.Efforts
30             .Where(x => x.UserId == user.AspNetUserId && x.EffortDate >= startDate && x.EffortDate <= endDate)
31             .Select(x => x.Task.Category.Project.CustomerId)
32             .Distinct()
33             .ToListAsync(cancellationToken);
34
35         await PseudoGetCustomerIdsWithEffortsDuring(session);
36         return customerIdsWithEffort;
37     }
38

```

Figure 8 The "EffortService" file

In the EffortService, we implement functions related to the Effort model that are necessary for the application development. For example, in a timesheet application, creating an effort is one of the essential functions required. Each method in the EffortService must be defined in IEffortService because IEffortService is the interface of EffortService, ensuring that EffortService follows the rules or blueprint defined in the interface.

For some methods used for data invalidation, as this is a timesheet application, data must always be fresh and up to date. For example, after creating, editing, or deleting an effort, the UI or data must be refreshed so that the latest data can be sent to the frontend or user interface using the GetEffort method.

For some method that can not invalidate in the normal way, we create a Pseudo method and invalidate it. PsuedoGet is used to trigger the invalidation of other computed methods such as GetEffortsByDateAndCustomer and GetTimeByDates. This ensures that these methods recalculate their results with fresh data from the database.

For each data model, such as Task, Project, and Customer, model entities need to be defined similarly to the DbEffort model. The choice of which model entities to implement depends on the requirements set by the developer. In this case, Task, Project, and Customer data models implemented, and several services were created using these data models for the application's purposes.

The problem is we need to use the old Mercury data which is login by the system username and password and the new mercury using Azure to login. To solve this problem, we have UserService.cs to filter and connect 2 different accounts become one with specific UserId.

Figure 9 is an example of UserService.cs

```

4
5 namespace Backend.Core.Users
6 {
7     public class UserService(IDbContextFactory<AppDbContext> dbFactory, IAuth auth) : IComputeService
8     {
9         [ComputeMethod]
10        public virtual async Task<MercuryUser> GetCurrentUser(Session session, Cancellation token cancellationToken)
11        {
12            var db = await dbFactory.CreateDbContextAsync();
13            User user = await auth.RequireUser(session, cancellationToken);
14
15            DbUserToAspNetUser? aspNetUser = await db.UserToAspNetUsers.
16                FirstOrDefaultAsync(x => x.UserId == user.Id.Value);
17
18            //var aspNetUser = await db.AspNetUsers.ToListAsync();
19            // var aspnet = aspNetUser.FirstOrDefault(u => u.UserName == user.Name);
20
21            if (aspNetUser == null)
22            {
23                throw new NullReferenceException("No UserToAspNetUser");
24            }
25
26            return new MercuryUser(aspNetUser.AspNetUserId);
27        }
28    }
29 }

```

Figure 9 The "UserService" file

5.2.3 Backend.Host

Backend.Host usually refers to the host or server where the backend services of an application are hosted. Figure 10 shows the Backend.Host tree

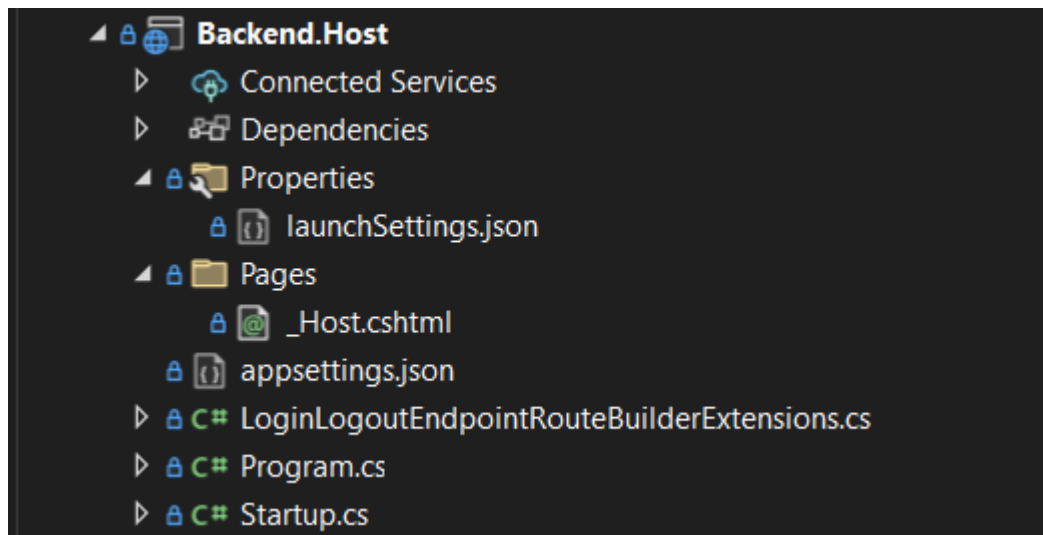


Figure 10 Backend.Host files tree

The launchSettings.json file is used to configure how the application will be launched. It allows specification of various settings such as the application profile, environment variables, the web server settings, and the URL to launch the application.

The appsettings.json file is used to store configuration settings for your application. It's used to store a wide range of configuration options, such as database connection strings, logging configuration.

The program.cs file contains the entry point for the application. The Main method in Program.cs is builds the web host and starts the application.

The Startup.cs file is used to configure services and the application's request pipeline.

5.3 Frontend

The front end is the user interface (UI) because it is where users interact with the application. Figure 11 illustrates the "Frontend" file tree.

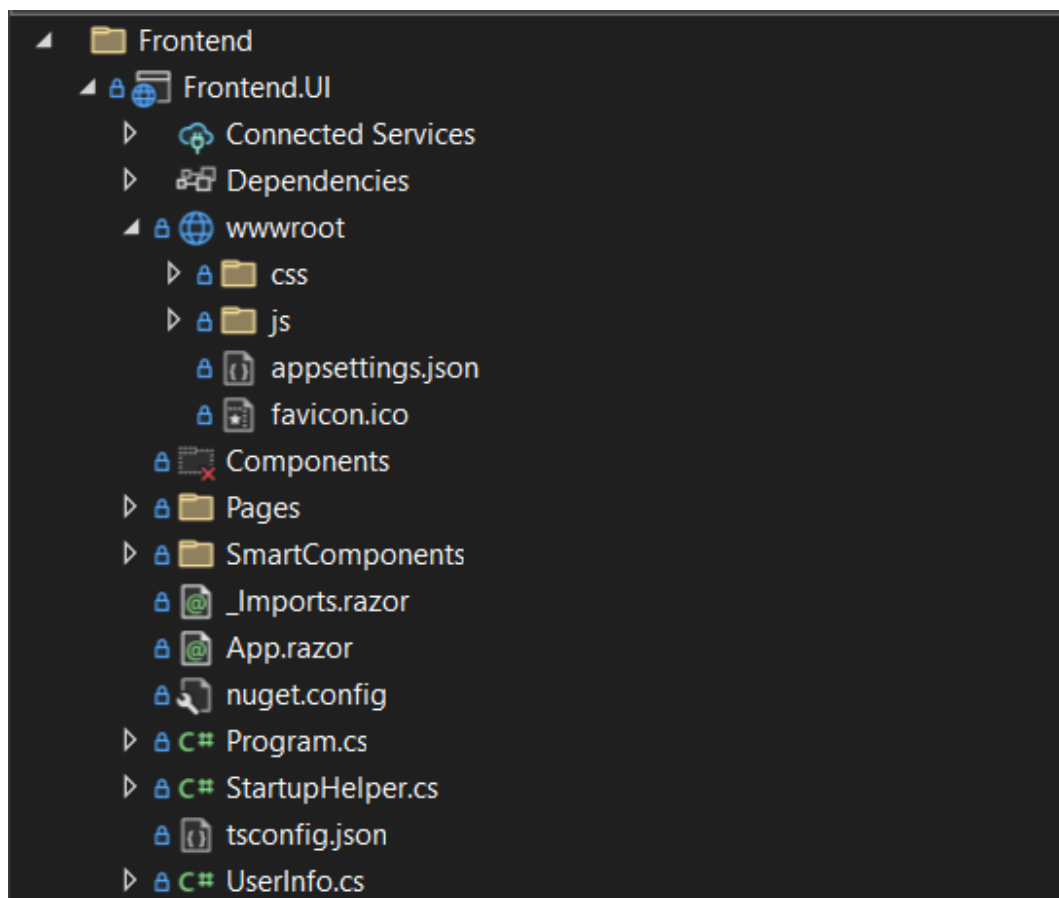


Figure 11 The "Frontend" files tree

The wwwroot folder stores static files like CSS, and images, which are important for the look and feel of the app and for running code in the browser. The Program.cs file is crucial for setting up the application; it configures services, handles routing, and starts the application. In Blazor WebAssembly apps, Program.cs starts the WebAssembly host, while in Blazor Server apps, it sets up the server. This organized structure helps developers easily manage and grow their Blazor applications, using the power of .NET and C# to create modern, interactive web interfaces.

5.3.1 Css

Where set the styling for the whole page with 2 files: app.css, reset.css. The App.css file is the place for setting up application text font, text style, text size for displaying. Figure 12 shows the CSS file tree and Figure 13 how to configure code in the CSS file.

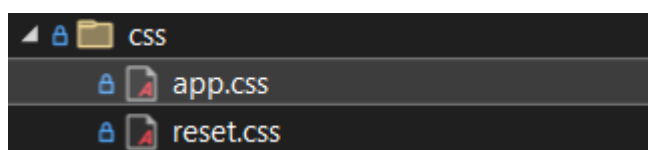


Figure 12 The "Css" files tree

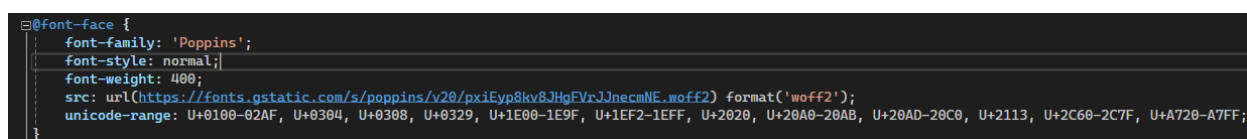
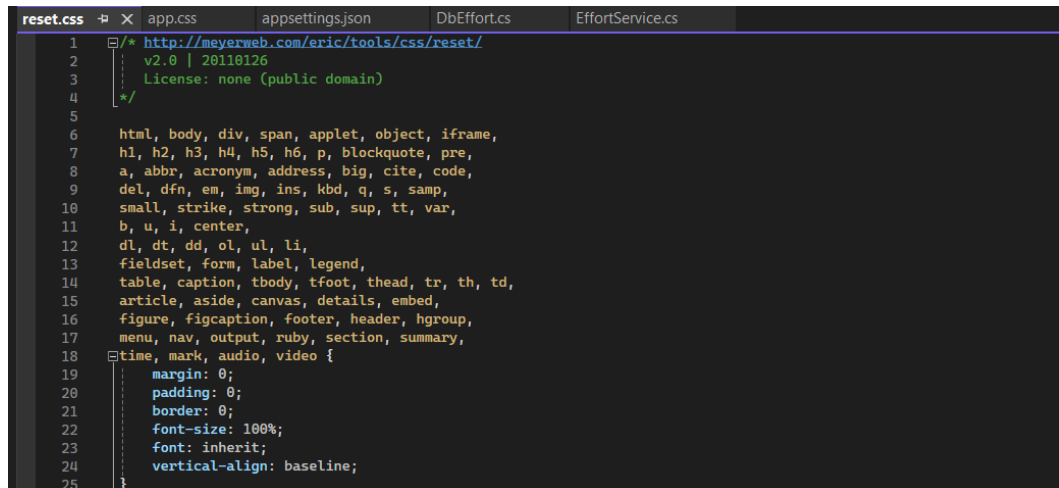


Figure 13 Part of css file

This CSS @font-face rule defines the "Poppins" font, specifying its style, weight, source URL, format, and the range of Unicode characters it supports. This allows the font to be used throughout the website.

Reset.css reset stylesheet normalizes default styles for HTML elements, ensuring consistent rendering across different browsers. It resets margins, paddings, borders, and sets default font sizes. Additionally, it normalizes HTML5 display properties for older browsers.

Figure 14 demonstrates how to configure code in Reset.css.



```

1  /* http://meyerweb.com/eric/tools/css/reset/
2     v2.0 | 20110126
3     License: none (public domain)
4  */
5
6  html, body, div, span, applet, object, iframe,
7  h1, h2, h3, h4, h5, h6, p, blockquote, pre,
8  a, abbr, acronym, address, big, cite, code,
9  del, dfn, em, img, ins, kbd, q, s, samp,
10 small, strike, strong, sub, sup, tt, var,
11 b, u, i, center,
12 dl, dt, dd, ol, ul, li,
13 fieldset, form, label, legend,
14 table, caption, tbody, tfoot, thead, tr, th, td,
15 article, aside, canvas, details, embed,
16 figure, figcaption, footer, header, hgroup,
17 menu, nav, output, ruby, section, summary,
18 time, mark, audio, video {
19     margin: 0;
20     padding: 0;
21     border: 0;
22     font-size: 100%;
23     font: inherit;
24     vertical-align: baseline;
25 }

```

Figure 14 The "Reset.css" file

5.3.2 Pages

Figure 15 illustrates the working tree of "Pages" folder

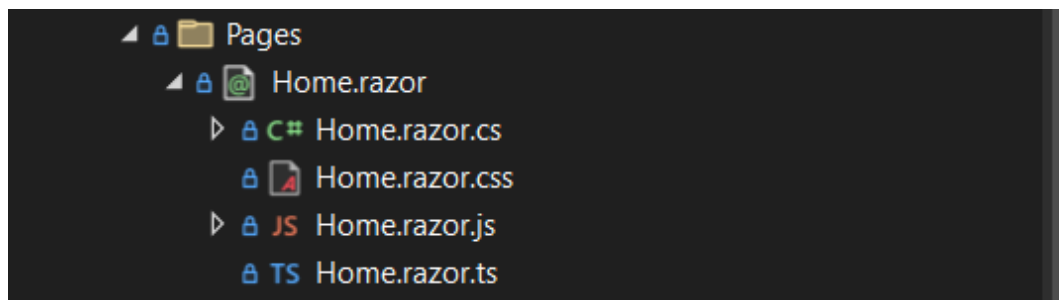
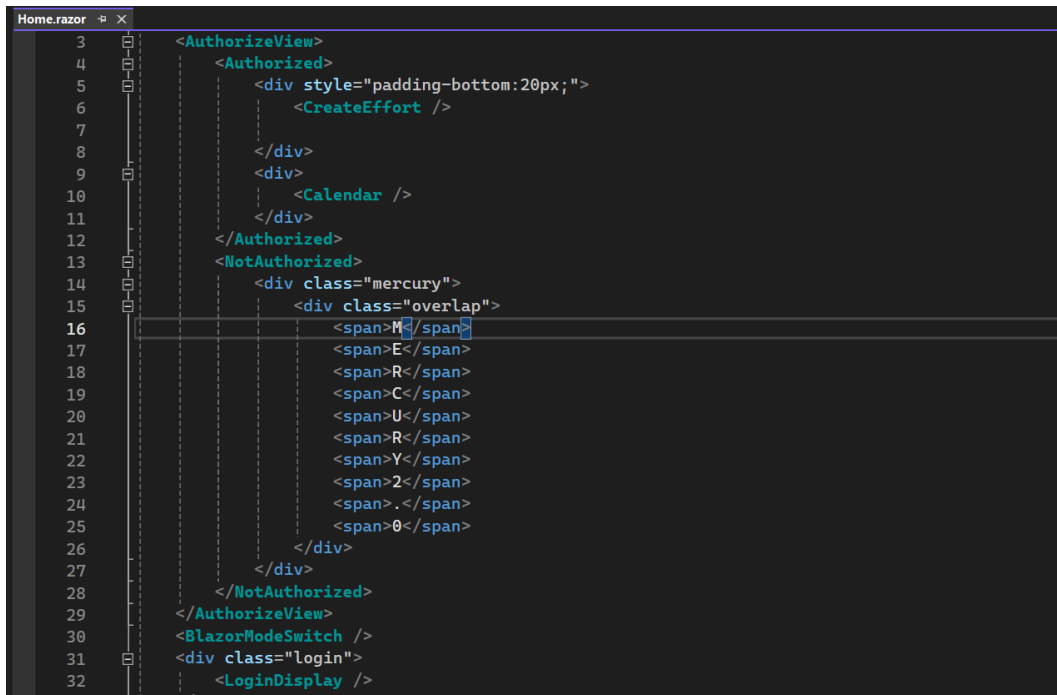


Figure 15 The "Pages" files tree

The Home.razor file is a Razor component in a Blazor application. It is designed to render different content based on the user's authorization status. For example, if

the user does not login it would display the home login page. After login or authorized with would display the home page of the web application which have 2 components: CreateEffort and Calendar.

An example of "Home.razor" file is shown in Figure 16.



```

3 <AuthorizeView>
4 <Authorized>
5 <div style="padding-bottom:20px;">
6 <CreateEffort />
7
8 </div>
9 <div>
10 <Calendar />
11 </div>
12 </Authorized>
13 <NotAuthorized>
14 <div class="mercury">
15 <div class="overlap">
16 <span>M</span>
17 <span>E</span>
18 <span>R</span>
19 <span>C</span>
20 <span>U</span>
21 <span>R</span>
22 <span>Y</span>
23 <span>2</span>
24 <span>.</span>
25 <span>0</span>
26 </div>
27 </div>
28 </NotAuthorized>
29 </AuthorizeView>
30 <BlazorModeSwitch />
31 <div class="login">
32 <LoginDisplay />

```

Figure 16 The "Home.razor" file

The AuthorizeView component handles the login and logout process, covering both authorized and unauthorized. When the user is not logged in yet, it would display all component inside the NotAuthorized tag. Upon successful authorization, two components, CreateEffort and Calendar, representing the Mercury application, are shown.

Figure 17 shows a successful login.

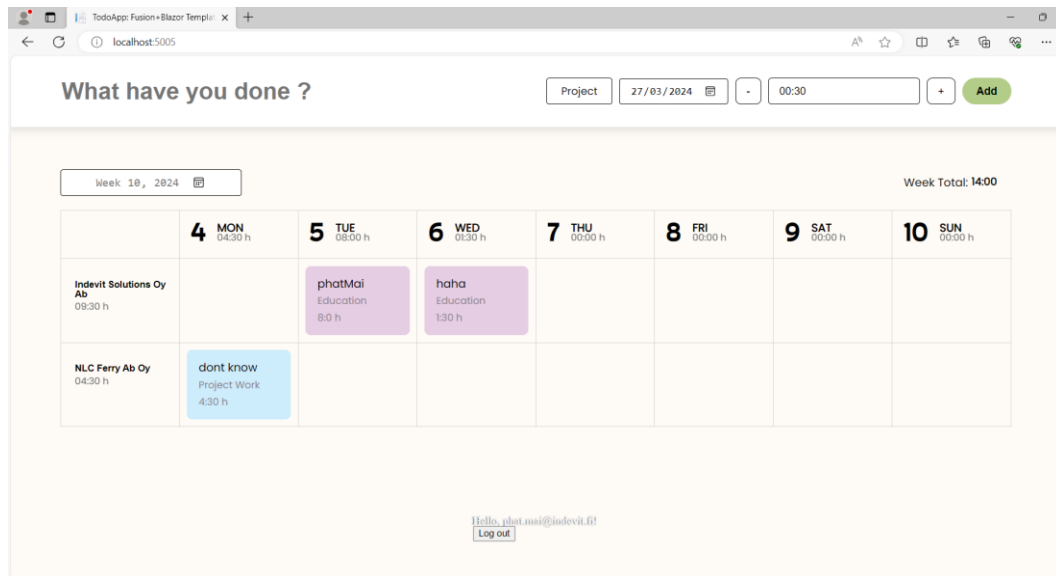


Figure 17 All component that include in "Home.razor" file display in the browser

5.3.3 Smart Components

For each smart component, this package would include three files: Component.razor, Component.razor.cs, and Component.razor.css. For example: Calendar.razor, Calendar.razor.cs, Calendar.razor.css. The purpose of each file type:

Figure 18 and Figure 19 how to implement the code in the calendar.razor file

```

Calendar.razor* X
6  <div class="calendar-container">
7  <div class="edit-modal">
8  <EditDeleteModal ModalVisible="ModalVisible" EffortId="EditingEffortId"
9  CurrentEffort="CurrentEffort" TaskId="TaskId" OnModalClosed="HandleModalClosed" />
10 </div>
11 <div class="week-container">
12 <input class="week-picker" type="week" @bind="SelectedWeek" @oninput="OnWeekChanged" />
13 <h class="week-total">
14 <WeekTimeTotal StartDate="StartDate" EndDate="EndDate" />
15 </h>
16 </div>
17
18 <div class="calendar">
19 <table>
20 <tr>
21 <td></td>
22 @foreach (DateOnly date in Dates)
23 {
24 <td>
25 <section>
26 <div class="calendar-date"> @date.Day</div>
27 <div class="date-detail">
28 <span style="font-weight: 700;">@date.ToString("ddd",
29 CultureInfo.InvariantCulture).ToUpper()</span>
30 <div class="date-total"><DateTimeTotal CalDate="date" /></div>
31 </div>
32 </section>
33 </td>
34 }
35 </tr>

```

Figure 18 The "Calendar.razor" file

```

Calendar.razor* X
36
37 @foreach (var customer in Customers)
38 {
39 <tr>
40 <td>
41 <div class="customer">
42 <CustomerCalendar Customer="customer" />
43 <h1><CustomersTimeCalculate StartDate="StartDate"
44 EndDate="EndDate" CustomerId="customer.CustomerId" /></h1>
45 </div>
46 </td>
47
48 @foreach (DateOnly date in Dates)
49 {
50 <td>
51 <EffortList CalDate="date" CustomerId="customer.CustomerId"
52 OnEffortClicked="HandleEffortClicked" />
53 </td>
54 }
55 </tr>
56 }
57 </table>
58 </div>
59
60
61
62 </div>

```

Figure 19 The "Calendar.razor" file

The purpose of Calendar.razor file is to display a calendar table which is the combination from many components: EditDeleteModel, WeekTimeTotal, DateTimeTotal, CustomerCalendar, CustomerTimeCalculate, EffortList.

With the loop Date, it helps this component to render the date which represents 7 days in the week.

EditDeleteModal pops up a modal where the user can edit the effort value (Description, Time, Date, Task) after clicking on the effort in the calendar.

WeekTimeTotal calculates and displays the total amount of time for the client throughout the week.

DateTimeTotal calculate and display the total amount of time in the day for the client.

CustomerCalendar render the customer's name in the calendar.

CustomersTimeCalculate calculate and display the amount of time that customer in a week.

EffortList display the efforts that the client has in the database.

Figure 20 show how calendar.razor display in the browser

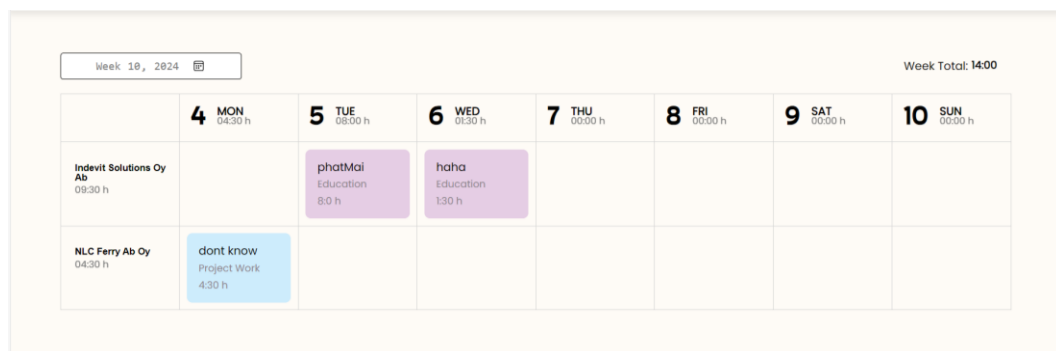


Figure 20 The "calendar.razor" display in the browser

Figure 21 depicts a part of Calendar.razor.cs

```

Calendar.razor.cs
Frontend.UI
Frontend.UI.SmartComponents.Calendar
OnInitializedAsync()

0 references
protected override async Task OnInitializedAsync()
{
    await base.OnInitializedAsync();
    if (string.IsNullOrEmpty(SelectedWeek))
    {
        var currentWeek = GetCurrentWeek();
        SelectedWeek = currentWeek;
    }

    UpdateDates(SelectedWeek);
}

1 reference
private string GetCurrentWeek()
{
    var currentDate = DateTime.Today;
    var weekNumber = CultureInfo.CurrentCulture.Calendar.GetWeekOfYear(currentDate,
        CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
    var currentWeek = $"{currentDate.Year}-W{weekNumber.ToString("00")}";
    return currentWeek;
}

1 reference
private async Task OnWeekChanged(ChangeEventArgs e)
{
    SelectedWeek = (string)e.Value;
    UpdateDates(SelectedWeek);
}

```

Figure 21 The "Calendar.razor.cs" file

The purpose of this component is to handle the logical part for the frontend (Calendar.razor). It renders week dates and binds parameters from another component so it can take values from it and display them on the front end.

The Calendar.razor.css component is used for styling how the Calendar.razor component is displayed. For example, it set width and height for calendars, calendar column with these elements: "width", "height".

Figure 22 show is example of Calendar.razor.css

```

Calendar.razor.css
Calendar.razor.cs

1
2 table {
3     width: 100vw;
4     border-collapse: collapse;
5 }
6
7 table, th {
8     border: 1px solid lightgrey;
9     width: 100%;
10    text-align: center;
11 }

```

Figure 22 The "Calendar.razor.cs" file

The purpose of CreateEffort.razor component is to render the input fields for the application. For example, this application has four input values: Description, Task (SearchButton.css component), Date, and Time. For the add time, user can both type

value in the input or click “+” or “-” to add hour, for each click it would plus or minus 30 minutes. All the input values in the front end would bind value to lower component which is CreateEffort.razor.cs. Figure 23 shows the code file of Create.Effort.razor and Figure 14 how it is displayed in the browser.

```

1 <form @onsubmit="Save">
2   <div class="adjust">
3     <input class="description" @bind="@Description" placeholder="What have you done ?" />
4     <SearchButton TaskClicked="HandleTaskClicked" SelectedTask="@SelectedTask" />
5     <input class="date" @bind="Date" type="date" placeholder="Date" />
6     <button class="time-button" @onclick="MinusTime">-</button>
7     <input class="time" @bind="@TimeInMinutesAsString" type="text" />
8     <button class="time-button" @onclick="PlusTime">+</button>
9     <button class="add" type="submit">Add</button>
10  </div>
11
12  @* <button @onclick="BustCache">Bust da cache</button> *@
13 </form>
14

```

Figure 23 The "CreateEffort.razor" file

Figure 24 This is how "CreateEffort" file

The purpose of CreateEffort.razor.cs component is to send all the values that the client has provided in the input form to the backend. It uses TaskService and EffortService in the backend, specifically the method in CreateEffort.cs. It takes the values (@bind) from CreateEffort.razor so that the component would have the values and can send them to the backend with CreateEffort method in EffortService. Figure 25 demonstrates CreateEffort.razor.cs code file.

```

78
79
80 public async Task Save()
81 {
82     if (TaskId != Guid.Empty && !string.IsNullOrEmpty(Description) && TimeInMinutes > TimeSpan.Zero)
83     {
84         var createEffortCommand = new CreateEffortCommand(Session, TaskId, CustomerId, Description, (int)TimeInMinutes.TotalMinutes, Date);
85         await EffortService.CreateEffort(createEffortCommand, CustomerId);
86         Description = "";
87         TimeInMinutes = TimeSpan.FromHours(0.5);
88         Date = new DateOnly(DateTime.Now.Year, DateTime.Now.Month, DateTime.Now.Day);
89         StateHasChanged();
90     }
91 }
    
```

Figure 25 The "CreateEffort.razor.cs" file

This component also sets smart values for the front end. For example: if client input are:

Table 1: Smart frontend convert case

Case	Input	Return
':' between number	9: 15	09: 15
' ' (space) between number	9 15	09: 15
1 Number only	10	10 (hour)
1 Number only	30	30 (minutes)

Figure 26 display the EditDeleteModal.razor code file

```

EditDeleteModal.razor.cs  EditDeleteModal.razor  CreateEffort.razor.cs  CreateEffort.razor  Calendar.razor.cs
1  @inherits ComputedStateComponent<MTask>
2
3  <div>
4      @if (ModalVisible)
5      {
6          <div class="blur-layout" @onclick="CloseModal"></div>
7      }
8      <div class="modal-dialog">
9          <dialog open="@ModalVisible">
10             <div>
11                 <form @onsubmit="Save">
12                     <h1>Task</h1>
13                     <input @bind="@Description" type="text" />
14                     <h1>Project</h1>
15                     <SearchButton TaskClicked="@HandleTaskClicked" SelectedTask="@SelectedTask" />
16                     <h1>Time</h1>
17                     <div class="time">
18                         <div>
19                             <button type="button" class="time-button" @onclick="MinusTime"></button>
20                         </div>
21                         <input @bind="@DisplayTime" type="text" />
22                         <div>
23                             <button type="button" class="time-button" @onclick="PlusTime">+</button>
24                         </div>
25                     </div>
26                     <h1>Date</h1>
27                     <input @bind="@Date" type="date" placeholder="Date" />
28                     <div class="button-container">
29                         <button class="delete" type="button" @onclick="Delete">Delete</button>
30                         <button class="save" type="submit">Save</button>
31                     </div>
32                 </form>
33             </div>
34         </dialog>
35     </div>
36 </div>

```

Figure 26 EditDeleteModal.razor file

The purpose of the EditDeleteModal.razor is displaying Edit or Delete Model for the Effort that selected in the calendar. In this model it would show the value of the selected effort. Users can edit effort value (Description, Time, Date, Task) in this model or user can Delete that effort it depends on user purpose. After user action, it would send or bind user input from the front end (EditDeleteModal.razor) to lower layer (EditDeleteModal.razor.cs).

In figure 27 you will see how EditDeleteModal.razor.cs look like as the code file

```

122     }
123
124     private async Task Delete() //Delete
125     {
126         var deleteEffortCommand = new DeleteEffortCommand(Session, EffortId);
127         if (Guid.TryParse(EffortId.ToString(), out Guid effortId))
128         {
129             await EffortService.DeleteEffort(deleteEffortCommand, effortId);
130         }
131         await OnModalClosed.InvokeAsync(false);
132     }
133
134     private async Task Save() //Edit
135     {
136         var editEffortCommand = new EditEffortCommand(Session, TaskId, Description, (int)TimeInMinutes.TotalMinutes, Date);
137
138         if (Guid.TryParse(EffortId.ToString(), out Guid effortId))
139         {
140             await EffortService.EditEffort(editEffortCommand, effortId, dateBefore);
141         }
142         await OnModalClosed.InvokeAsync(false);
143         StateHasChanged();
144     }

```

Figure 27 The "EditDeleteModal.razor.cs" file

The purpose of EditDeleteModal.razor.cs component is handling the logical part for the EditDeleteModal.razor, it takes value binding from the front end and resolves it before sending it to the back end.

After Edit or Delete, it would send the value to the backend through 2 Task: Delete (DeleteEffort method in the EffortService) and Save (EditEffort method in EffortService).

In this component also have the smart input value. It checks that if the input has ":" or " ", if it does split the input string into 2 parts: hour and minute.

Figure 28 shows what smart convert method look like

```

3 references
private string DisplayTime
{
    get => TimeInMinutes.ToString(@"hh\:mm");
    set
    {
        if (value.Contains(":") || value.Contains(" "))
        {
            string[] strings = value.Split(':', ' ');
            if (strings.Length == 2)
            {
                if (int.TryParse(strings[0], out int hour) && (int.TryParse(strings[1], out int minute)))
                {
                    if (hour < 15)
                    {
                        TimeInMinutes = TimeSpan.FromHours(hour) + TimeSpan.FromMinutes(minute);
                    }
                    else
                    {
                        TimeInMinutes = TimeSpan.FromMinutes(hour * 60 + minute);
                    }
                }
            }
        }
        else
        {
            if (int.TryParse(value, out int hour))
            {
                if (hour < 15)
                {
                    TimeInMinutes = TimeSpan.FromHours(hour);
                }
                else
                {
                    TimeInMinutes = TimeSpan.FromMinutes(hour);
                }
            }
        }
    }
}

```

Figure 28 EffortList.razor

It then converts the hours and minutes into an integer and checks if the conversion was successful.

If the hour is less than 15, it converts the time to minutes using `TimeSpan.FromHours(hour) + TimeSpan.FromMinutes(minute)`.

If the hour is 15 or more, it converts the time to minutes using `TimeSpan.FromMinutes(hour * 60 + minute)`.

Figure 29 gives an example of EffortList.razor

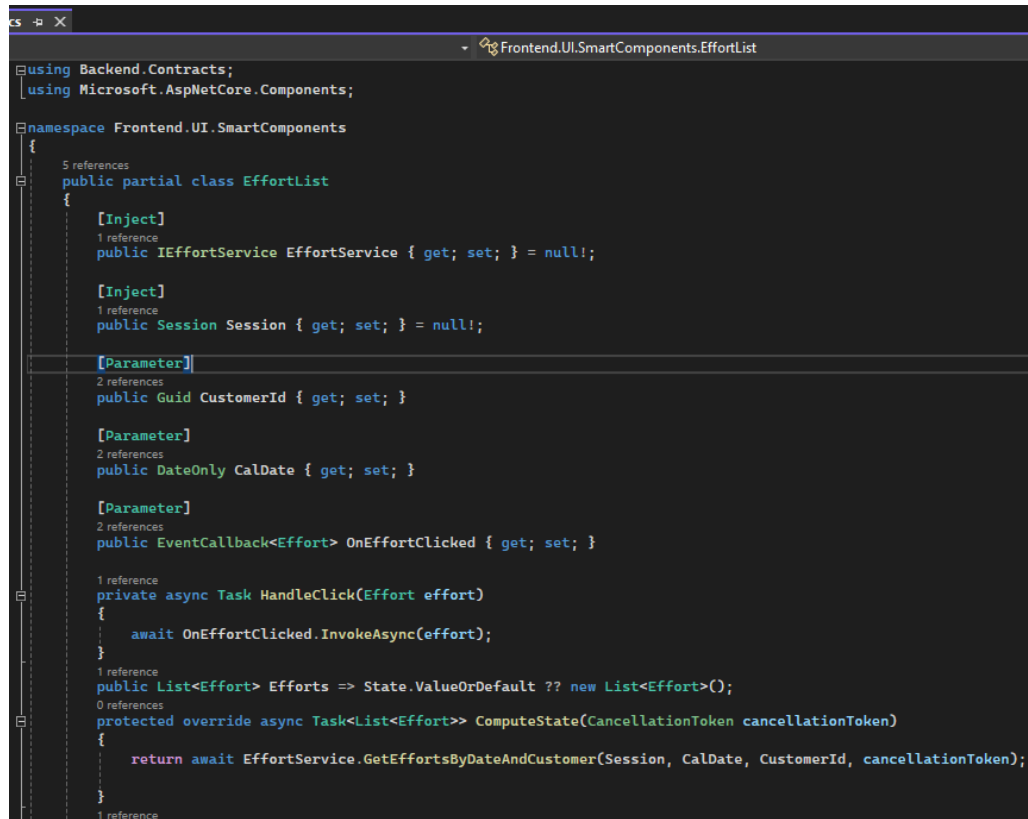
```
EffortList.razor* # x
1  @inherits ComputedStateComponent<List<Effort>>
2
3  <div class="">
4      @foreach (var effort in Efforts)
5      {
6          <div @onclick="(C) => HandleClick(effort)" class="card" style="background-color:
7              @(GetCardColor(effort.CustomerId))">
8              <ol>
9                  <li class="effort-header">@effort.Description</li>
10                 <li class="effort-body">@effort.Name</li>
11                 @if
12                 {
13                     var hours = effort.Time / 60;
14                     var minutes = effort.Time % 60;
15                     var time = $"{hours:D2}:{minutes:D2}";
16                 }
17                 <li class="effort-body">@time</li>
18             </ol>
19         </div>
20     }
21 </div>
```

Figure 29 The "EffortList.razor" file

The purpose of EffortList is to render the efforts obtained from the lower component (EffortList.razor.cs). It displays three values of each effort: Description, Time, and Name (CustomerName). The Time value is displayed as hours and minutes.

Additionally, this component includes a HandleClick method which sends the value of the selected effort to the EditDeleteModal, allowing the user to edit or delete the selected effort.

Figure 30 shows an example of EffortList.razor.cs



```

cs  X
Frontend.UI.SmartComponents.EffortList

using Backend.Contracts;
using Microsoft.AspNetCore.Components;

namespace Frontend.UI.SmartComponents
{
    5 references
    public partial class EffortList
    {
        [Inject]
        1 reference
        public IEffortService EffortService { get; set; } = null!;

        [Inject]
        1 reference
        public Session Session { get; set; } = null!;

        [Parameter]
        2 references
        public Guid CustomerId { get; set; }

        [Parameter]
        2 references
        public DateOnly CalDate { get; set; }

        [Parameter]
        2 references
        public EventCallback<Effort> OnEffortClicked { get; set; }

        1 reference
        private async Task HandleClick(Effort effort)
        {
            await OnEffortClicked.InvokeAsync(effort);
        }

        1 reference
        public List<Effort> Efforts => State.ValueOrDefault ?? new List<Effort>();
        0 references
        protected override async Task<List<Effort>> ComputeState(CancellationTokentoken cancellationTokentoken)
        {
            return await EffortService.GetEffortsByDateAndCustomer(Session, CalDate, CustomerId, cancellationTokentoken);
        }
    }
    1 reference

```

Figure 30 The "EffortList.razor.cs" file

This component's purpose is to fetch effort for specific customer and date by passing parameter: CalDate, CustomerId, each effort always has different date and customerId, so with CalDate, customerId parameter it would render effort

for each day and customer. For example: 3/4/2024 have 2 efforts with 2 different customers (Mirka, Entia).

In figure 31 you will see GetEffortsByDateAndCustomer method

```
public List<Effort> Efforts => State.ValueOrDefault ?? new List<Effort>();  
0 references  
protected override async Task<List<Effort>> ComputeState(CancellationTok  
{  
    return await EffortService.GetEffortsByDateAndCustomer(Session, CalDate, CustomerId, cancellationToken);  
}
```

Figure 31 GetEffortsByDateAndCustomer method

This code purpose is getting the effort from the database with specific date and customerId then the front end displays it with 3 values: Description, Time, Name

It also defined customer card color by GetCardColor, each customer has specific color with color hex code.

There are many more components that are related to calculating working time for calendar: WeekTimeTotal, DateTimeTotal, CustomersTimeCalculate.

WeekTimeTotal: The GetTimeForWeek method in EffortService requires a StartDate and an EndDate to calculate the total hours in a week for a user. These StartDate and EndDate parameters are provided in Calendar.razor.cs. The calculation begins when the user chooses the week to display, ensuring that the data is always up-to-date whenever the user adds, edits, or deletes effort.

DateTimeTotal: it requires CalDate to calculate total hour in a date of customer. CalDate is provided in Calendar.razor.cs. The calculation happens for the date

column that has effort inside, ensuring that the data is always up to date whenever the user adds, edits, or deletes effort. It is using `GetTimeInDay` method to calculate.

`CustomersTimeCalculate`: requires `CustomerId` to calculate total hours of the customer that user works on. The calculation happens for the customer row when there has effort appear in the week. Using `GetTimeByDates` to calculate.

5.3.4 Migrations

`CreateUserTable.sql`

This SQL script creates database tables and is used to synchronize the development and test databases. If developers make changes to the database schema, such as adding new columns, they need to include those changes in this script to ensure synchronization between the databases.

`UserToAspNetUser.sql`

The purpose of this table is to establish a relationship between a `UserId` from another table (likely the `Users` table) and an `AspNetUserId` from an ASP.NET Identity related table. This could be useful for linking user accounts between different systems or services.

5.4 Deployment

During application development, Indevit uses Azure to manage the development process, so deploying web applications also utilizes Azure. After setting up IIS, developers click "Save & Queue" to build the package that the server needs to run. Once the build is successful, developers can create a new pipeline to send the package to the IIS server.

In figure 32 you will see the example of Azure pipeline system.

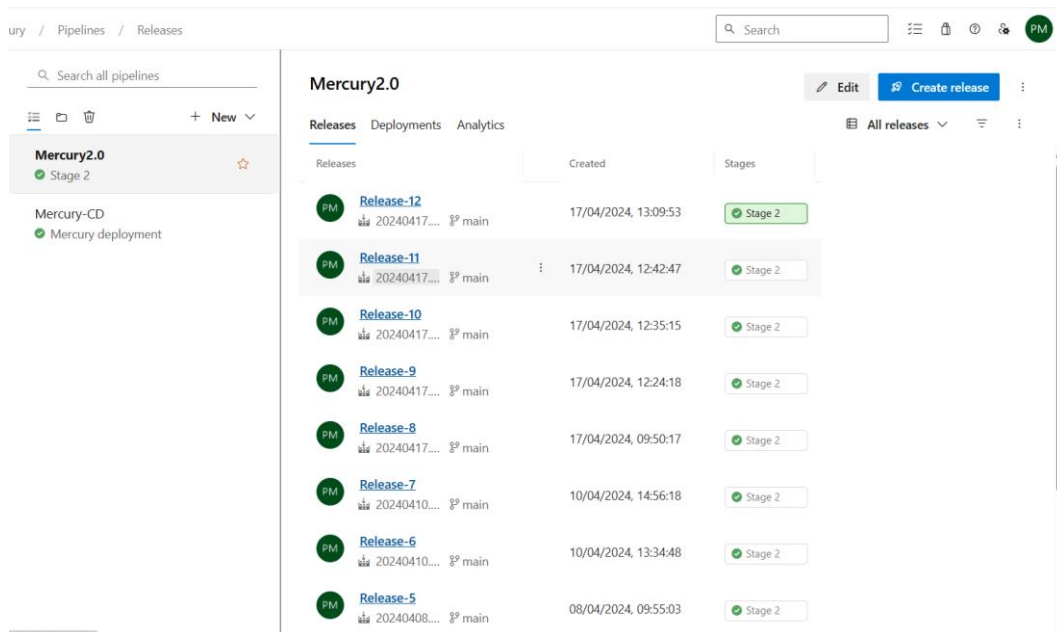


Figure 32 Azure pipeline system

Azure Pipeline Releases automate the deployment of application updates across different environments, ensuring a consistent and controlled process. This service in Microsoft Azure handles the transition from development to production, reducing manual effort and errors.

Key components include artifacts (build outputs), stages (environments like development and production), jobs (deployment steps), and approvals (reviews before and after deployments). By automating these processes, Azure Pipelines enhance reliability, speed, and compliance with organizational policies. Adopting best practices like using templates and securing data further improves efficiency. Overall, Azure Pipeline Releases streamline deployments, improving software quality and delivery speed.

5.4.1 IIS

Internet Information Services, also known as IIS, is a Microsoft web server that runs on Windows operating system and is used to exchange static and dynamic web content with internet users. IIS can be used to host, deploy, and manage web applications using technologies such as ASP.NET and PHP.

5.4.2 Azure Pipeline

Azure Pipelines automatically builds and tests code projects. It supports all major languages and project types and combines continuous integration, continuous delivery, and continuous testing to build, test, and deliver your code to any destination.

Before setting up the release pipeline, a build pipeline is configured to build the application and produce artifacts for deployment. Once the build pipeline is set up, the release pipeline is created. This involves adding the artifact produced by the build pipeline and configuring tasks such as creating/updating the website, configuring the application pool, and starting the website.

In figure 33 is a example of Azure pipeline setting

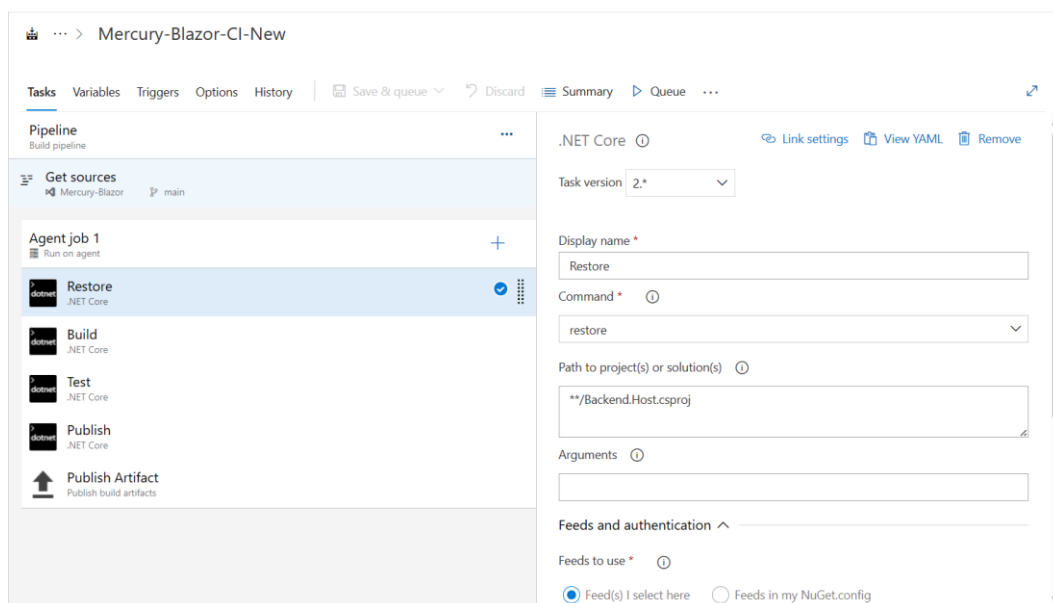


Figure 33 Azure pipelines setting

Deployment stages are configured to define the deployment workflow, such as Dev, QA, and Production environments. Finally, the release is triggered, and the

deployment process is automated, ensuring rapid and consistent delivery of code changes to production.

Azure Pipeline Release streamlines the deployment process, ensuring that every code change goes through the same build, test, and deployment process, thereby enhancing the efficiency and reliability of the deployment process.

6. OUTCOME OF PROJECT

The application has three page states: the login page, the homepage, and the EditDeleteModal popup page. When the application starts, the login page is the initial page.

In the figure 34 you will see how the login page looks like

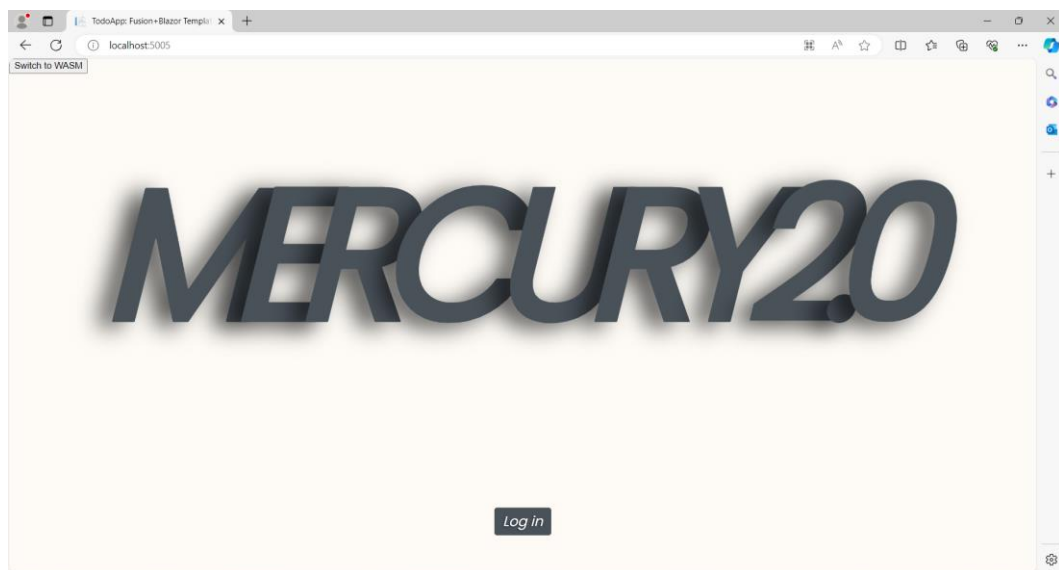


Figure 34 Login page

After logging in with an Azure account, the user is redirected to the homepage, where the create effort bar and calendar are displayed. Users can select the week they want to display in the calendar by clicking on the Week select.

To add a new effort to calendar, user must fill value into create bar. Type your comment in “What have you done” part which is Description. Click into Project Button then it would popup the modal where user can search for specific task name, customer name, task category or user can search with those 3 values. For example: type “Entia webdev publish” and it would appear the task that user wants to create.

Users can select the date by choosing the date in the date field or user can type the date they want into that field.

Users can set the time value by click “+” or “-” button to change the value or can type specific time into the input field.

After filling all those input fields, users can create a new effort with those values.

In figure 35 you will see an example of how application display with effort card.

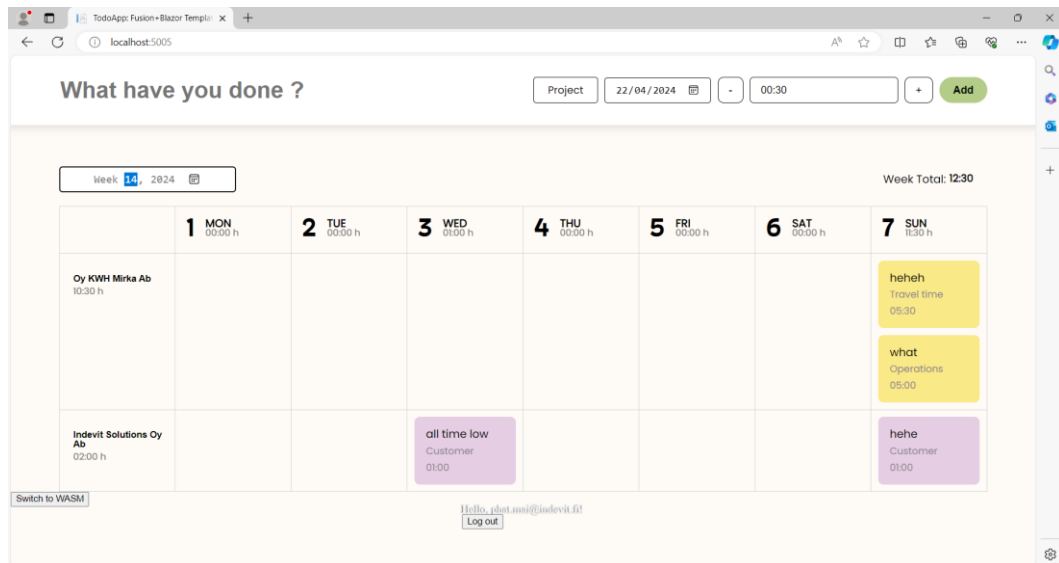


Figure 35 Mercury-timesheet page

Users can edit or delete efforts by clicking on the effort displayed in the calendar. After clicking on the effort, they want to change, a modal will pop up displaying the selected effort's details. Users can then make whatever changes they want and click "Save" to save the changes or "Delete" to delete the effort. Once saved or deleted, the effort will be updated or removed immediately, ensuring that the data is always up to date for display, thanks to the EffortService's validation process.

In figure 36 you will see how EditDeleteModal look like after clicked into Effort card

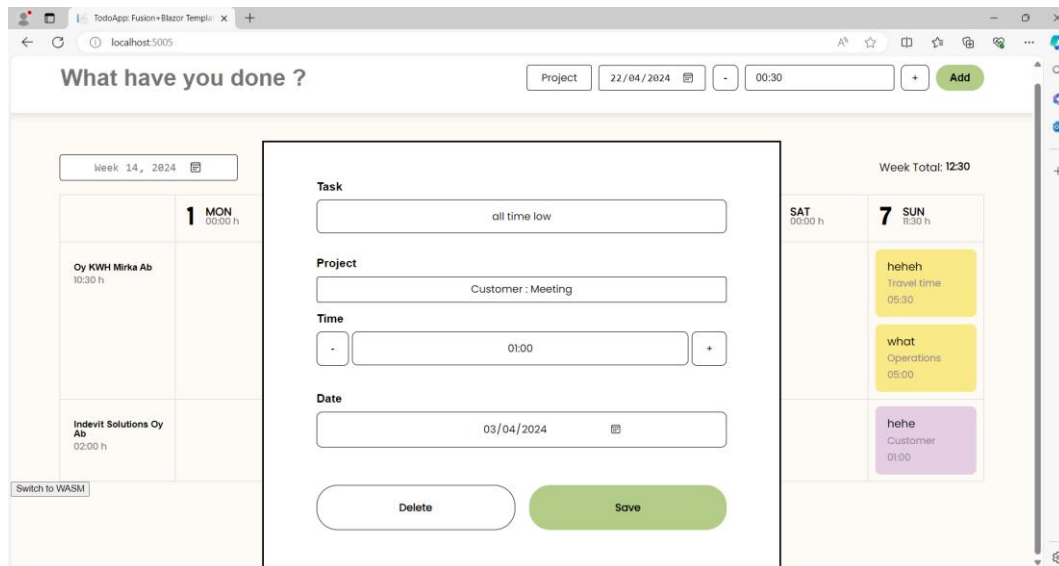


Figure 36 EditDeleteModal popup

User can type or edit value in input field of modal, user need to click Save to save the change to backend. they also can delete the effort by click DELETE. Whenever you click outside the modal, it auto close without saving change.

The application is ready to use, but it is only intended for internal use by employees of Indevit AB Oy. While there are many new features planned, Mercury 2.0 is ready for use based on the current application requirements.

7. CONCLUSIONS

During my internship at Indevit Ab Oy, I had the opportunity to work on the development of Mercury 2.0, a web application designed to streamline the process of recording, tracking, and editing employee working hours. Throughout this experience, I gained valuable skills in web development, including HTML/CSS, Blazor, Razor, React, and C#.

I was guided to develop the application according to a certain plan by dividing tasks on Azure devops, so that my working process was also easier in planning and dividing time appropriately for each function. Based on that, the company can easily monitor and evaluate my work process. Starting the development process, I had to spend time learning new tools, such as Fusion, Blazor, Razor. After gaining the necessary knowledge, I started developing from the smallest things needed in a web application. During the development process, I worked alone, so encountering difficulties was completely expected. But thanks to the experience shared from my teammates and my boss, I have learned many things from debugging and fixing bugs in a more systematic and logical way. The biggest challenge is data flow between components because the developer must know which components need that data and which do not use it, which components receive it, and which components need to be invalidated.

Mercury 2.0 was designed with a new and improved user interface that retains the functionality of the original system while making it easier to use. New features were added to simplify the process of adding, editing, and deleting hours, improving overall usability and efficiency.

After extensive testing by developers and Indevit employees, Mercury 2.0 is complete and performing excellently. The application runs smoothly with fast-loading pages, no bugs, and a highly intuitive, user-friendly interface. Future includes adding more impressive features to further expand its functionality.

Through the development of Mercury 2.0, I gained valuable experience as a software engineer, learning the importance of continuous improvement and innovation in software development. I enhanced my skills in learning new technologies, coding, debugging, and problem-solving, which will be invaluable in my future career.

My supervisor, the CTO, was thrilled with the results, confirming that the application is bug-free and that all features work perfectly. They commended the new user interface for its attractive design and ease of use, stating that it is ready for a successful launch. Another Indevit employee also praised the application's smooth performance and beautifully designed, user-friendly interface, highlighting the outstanding work done.

There are still many improvements that could be made to the application in the future, making it more user-friendly and functional. For example, connecting with Powerpi would make it easier for international business teams to work with the database. Implementing an employee payment invoicing system would allow employees to know how much salary they receive each month. Hourly record notifications would help users keep track of their latest actions. Additionally, a day-off reservation feature would greatly assist companies in managing employee holidays.

REFERENCES

- Alexyakunin. (2020). *Stl.Fusion*. Retrieved from github:
<https://github.com/servicetitan/Stl.Fusion>
- Brind, M. (2018). *Learn Razor Pages*. Retrieved from learnrazorpages:
<https://www.learnrazorpages.com/>
- Meghan, S. &. (2023). *.NET Framework: Advantages and Disadvantages*. Retrieved from softjourn: <https://softjourn.com/insights/net-framework-advantages-and-disadvantages>
- Microsoft. (2016). Retrieved from <https://learn.microsoft.com/en-us/visualstudio/productinfo/vs-roadmap>
- Microsoft. (2018). Retrieved from <https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16>
- Microsoft. (2019). *Blazor - Build client web apps with C# | Microsoft Docs*. Retrieved from Microsoft: <https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor>
- Mike. (2018). *Learn Razor Pages*. Retrieved from learnrazorpages:
<https://www.learnrazorpages.com/>
- Mike. (2019). *Learn Blazor*. Retrieved from learnblazor:
<https://www.learnblazor.com/>
- Mike. (2019). *Learn Blazor*. Retrieved from learnblazor:
<https://www.learnblazor.com/>
- Rick, D. &. (2017). *Introduction to ASP.NET Core*. Retrieved from aspnetcore:
<https://aspnetcore.readthedocs.io/en/stable/intro.html>

W3Schools. (2013). *Introduction to CSS*. Retrieved from W3Schools.in:
<https://www.w3schools.in/css3/introduction-to-css>

Wikipedia. (2001). Retrieved from <https://en.wikipedia.org/wiki/HTML>

Wikipedia. (2022). Retrieved from
https://en.wikipedia.org/wiki/Microsoft_SQL_Server

Wikipedia. (2022). *Database*. Retrieved from wikipedia:
<https://en.wikipedia.org/wiki/Database>