Scheduling of accounting data in a planning tool

Development of a work schedule

Jakob Topholt Jensen, Karl Barnholdt Kragsaa, Ibrahim Alaiddin Abu Rached, Hans Richard martinussen, Christian Ingemann Otte, Rune Gøttsche Aagaard and Søren Sønderholt Christiansen

Software Engineering, 2026-12

Third Semester Project

Overleaf and LaTeX is used for writing the report. The implementation of the solution is in the programming language C# and ASP.NET 5.0. Blazor will be used to render the front-end.

Department of Computer Science
Aalborg University
https://www.aau.dk

Title:
Scheduling of accounting data in a planning tool

Theme:
Software Engineering

Project Period:
Fall Semester 2021

Project Group:
cs-21-sw-3-15

Participant(s):
Christian Ingemann Otte
Hans Richard Martinussen
Ibrahim Alaiddin Abu Rached
Jakob Topholt Jensen
Karl Barnholdt Kragsaa
Rune Gøttsche Aagaard
Søren Sønderholt Christiansen

Supervisor(s):
Imran Riaz Hazrat

Copies: 1

Page Numbers: 77

Date of Completion:
February 18, 2026

Abstract:

The group has worked with the accounting firm Attiri and has been in close contact with two representatives from the firm. The program which Attiri has requested is a program which can extract all projects from their database (such as VAT or earnings reports) in a format similar to a spreadsheet. This is so that it becomes easier to get an overview of the work load in different periods rather than struggle to maintain an overview a few days into the future.

To provide a solution, we had to develop a time management tool so that an overview of projects and the creation of sub-tasks for projects can be facilitated.

The result of the development is an object oriented web-based application in C# which uses the Blazor framework.

Contents

Chapter 1

Introduction

Attiri is an auditing firm located in Aalborg. The firm has seven employees and services many companies of varying sizes. In total, Attiri works with about 700 customers.

The project group approached Attiri regarding identifying a potential problem that we could use for our 3rd semester project. The project group met with the company for a discussion about what problems they were facing and how collaboration between the group and the firm might be established. The current system that Attiri is using, involves Excel spreadsheets for time management, paper planning and regular meetings to distribute workloads. This brings forth issues in work delegation and detail planning, and sometimes causes the company to miss important deadlines. Attiri is looking for a better way to manage and plan their time, so that every employee has a clear overview of what tasks they have scheduled for the day, and which important deadlines are approaching. The proposed solution is a data visualizing and time management tool, that runs off of the company's current solution, Uniconta. The tool is meant to help provide a better overview of the company's available time, work pressure, task management and delegation. Additionally the solution is meant to alert the users to relevant and urgent tasks and helping the company miss fewer deadlines and spread the workload more efficiently as to avoid working overtime.

Acquiring an understanding of the problem is necessary to develop a solution to their issues. By contacting Attiri on a weekly basis throughout the project period, we ensured that all expectations were met.

1.1 Methodology

Accounting is a profession that no member of the group has experience with. We are to achieve a thorough understanding of the inner workings and the daily processes of the firm that that we are collaborating with, in order to be able to make a solution that befits them. Due to the complexity of our task being low and the uncertainty being high, the iterative design process has been chosen as our preferred model of designing, implementing and documenting our solution.

The design and implementation will follow general principles and guidelines from the book "Object Oriented Design Analysis" as this is the book that we use throughout our course in Systems Development [10]. This helps us ensure a high quality of our system architecture, whilst applying the curriculum of a course to the project. Furthermore, as we are to create a Mental Model of our problem domain and our application domain, we will use methods and strategies from the book "Designing User Experience" for information gathering, usability testing and specification of requirements [2].

Chapter 2

Problem Analysis

2.1 Problem Statement

The accounting firm Attiri currently uses a system for time management that revolves around a person with scheduling responsibility that plans and delegates projects and tasks to the other employees. As the firm has 600 – 700 customers it can be difficult to manage deadlines, upcoming tasks and gain a broad overview of all employees and their work.

Based on this, the problem statement will be formulated as:

"How can we make a tool that provides the person in charge of planning with an overview of individual employees' workload and makes it manageable to meet deadlines across multiple projects and tasks?"

2.2 Current System

Attiri uses an excel sheet to plan and delegate tasks to their employees, as they don't have an actual system for their planning. The problem with their current management method, appears to be the missing ability to see the distribution of workload over a longer period of time. Only having the ability to plan short periods at a time, has the consequence of creating periods of uneven workloads. Therefore the employees have periods with a lot of overtime and other periods where they barely have enough tasks to fill out a time schedule.

The excel sheet consists of dates, tasks and employees. For each task, a time is estimated to solve it, and through this estimation they can estimate which employees have the available time to complete them. We have been asked to retain this part of the planning system in our solution. One of their largest problems with this domain of their planning system is the lack of overview of which subtasks are approaching their deadlines. The lack of an overview of approaching deadlines results in two scenarios; either missing a deadline entirely, which in this industry may result in a fine for the accounting firm or they may have to force the employees to work overtime to avoid missing said deadline. None of the results would be satisfying for Attiri or their customers, which is why they need a planning tool.

The excel sheet is a local file which is exchanged between the employees and CEO(s) responsible for the planning. This results in a waste of time where the employee has to wait to be delegated a task, in order to have work to do. This part of the system they would like to have visualized so that each employee, on their own, can find out which tasks they have the responsibility for.

2.3 System Definition

The system is defined for use in the daily work of managing the assignments of the employees and handling their workload, as well as alerting managers and employees of approaching deadlines. The system should be used by project managers for scheduling work and by employees to get an overview of their daily assignments. The system should enable both project managers and employees to get an overview of the specific assignments and projects that are relevant to them. The system is based on a single server which can be accessed by any device with a browser and a valid log-in. The development process will happen in close dialogue between a project manager from the accounting firm and the developers.

2.4 FACTOR

In order to get a clearer picture of the proposed solution, a FACTOR analysis has been done, which will help both the developers and the firm to stay consistent with the solution. This analysis helps define the functionalities and conditions which encapsulate the solution and broadly defines the system as a whole, and serves as a guideline for the development process.

FACTOR
This FACTOR analysis is based on two meetings with Attiri and the information that was gathered during and after them. With the analysis we can simplify the process and place the different aspects into groups to support the system-definition development. The FACTOR consists of six elements:

Functionality: The functions that support the application domain tasks

Application domain: Those parts of an organization that administrates, monitor or control the problem domain

Condidtions: The conditions under which the system will be developed and used

Technology: The technical platform used to develop and run the system

Objects: The main objects in the problem domain

Responsibility: The systems overall responsibility in relation to the context

Functionality
The main functionality of the program is to create an easy overview of the accountants' tasks, workflow and show which tasks are nearing their deadlines. The program will use an already existing application's API and through it gather data and create a login.

Application domain
A project manager, who is also an accountant, administrates the program and supplies the accountants with tasks. The accountants can visit their own profile to view which tasks they have been assigned, as well as their deadlines.

Conditions
The program will be developed by the project-group from Aalborg University as part of their curriculum. After submitting the solution, Attiri will be given ownership of the program no matter its state of operability.

Technology
The program can be accessed on any PC platform with a web-browser and an internet connection. The program itself will be created with the C# framework "Blazor" and Bootstrap CSS.

Objects
The main objects in the problem domain are the project manager, as he is the super-user, accountants using the program, projects, subtasks and alerts.

Responsibility
It is the program's responsibility to create an overview of the accountants' tasks and deadlines. It is also responsible for showing each accountant their tasks along with a customer evaluation.

2.5 MoSCoW Analysis

The MoSCoW analysis is a way to prioritize and plan the development of the program. In the MoSCoW analysis the discussed functionalities of the program is divided into four categories by importance.

Must have: Must have represents the minimum requirements that the system must fulfill.

Should have: should have represents requirements that should be fulfilled but are not necessary for the system to be functional.

Could have: Could have represents the requirements that have been discussed but will only be fulfilled if the time constraints allow it.

Wont have: Wont have represents the requirements which have been discussed but will not be fulfilled.

Must have

- A login system.
- An easy overview of tasks, alerts, unassigned tasks, customers and workload.
- Fast editing of subtasks in the existing database.

Should have

- Graphs displaying various data depending on the page.
- Reminders
- A way to review customers in the form of blog-posts.

Could have

- A mobile version of the system.
- An “annual wheel” giving the managers a fast overview of the important deadlines of the year.
- A drag and drop system for reassigning tasks.
- A task dependent time factor with hourly rate.

Won’t have

- Automated delegation of tasks.
- Integrated reading of call history for automated fee-charging.
- Drag and drop interaction with elements throughout the system.

Chapter 3

System Development

This chapter will cover analysis of the overall system design and documentation of the component, criterion and architecture. The analysis revolves around analysis of the problem domain and the appropriate methods used for the analysis. The methodology will be covered briefly during the introduction of each section followed by the results of the associated analysis.

3.1 Criterion Analysis

A criterion analysis is a method by which the developers of a system can evaluate the importance of several aspects of a system. e.g. a credit card terminal might prioritize security and efficiency over reusability, portability and interoperability, these terms are explained in [10] (p. 180).

Criterion	Very important	Important	Less Important	Irrelevant	Easily fulfilled
Usable	X				
Secure					X
Efficient			X		
Correct	X				
Reliable		X			
Maintainable					X
Testable		X			
Flexible				X	
Comprehensible			X		
Reuseable				X	
Portable					X
Interoperable	X				

Table 3.1: The Criterion analysis table which showcases which qualities are prioritized

3.1.1 Comments on the criterion analysis

Usable: It is essential that the employees find the system intuitive and suiting to their perception of what a management tool looks like. The system is to take inspiration from spreadsheets to mimic the systems that the employees regularly use, and have some recognizability towards.

Secure: Security refers to the steps taken to avoid unauthorized access to the system. The system treats sensitive data, however the API handles the security. Therefore handling of security becomes trivial, as it lies within the system from which the data is extracted, and is not directly handled by the development team.

Correct: It is important to fulfill the system requirements as they have been designed, since the system will not satisfy the MoSCoW model without doing so. The system has several aspects which are all 'must haves' and as such the system will be thoroughly lackluster without a high level of correctness.

Maintainable: Maintainability refers to the ease of resolving issues arising after the system is deemed finished. A close-strict format for the programming will be implemented, unit testing and several exception throws which should ensure that no maintenance will be needed, incorporating this structure should make maintenance easily fulfillable.

Interoperable: Interoperablity is what the cost of coupling the system to another system will be. A

high level of interoperablity is imperative to the system, since we're required to extract all data via Uniconta to create the schedule.

Flexible: Flexibility is deemed not important to the development process, since the system is to be made for Attiri specifically, without the need for features to be added as the system will satisfy the correctness criteria as noted above.

Reuseable: Reusability refers to the potential for reusing aspects of the system to other related systems. There will not be a focus on being able to reuse code in other systems, since the solution provided to Attiri is a one-time project.

Portable: Portability refers to the cost of porting the system to a new platform (e.g from PlayStation to PC or from Google Chrome to Firefox) Since we are using Blazor as our development framework, the system can be opened on any platform that has a browser and internet access, hence it is easily fulfilled.

3.2 Problem Domain Analysis

The purpose of problem domain analysis is to establish the classes for programming the solution and establish the correct architecture along with their attributes, to serve as a guidelines for the continued development.

3.2.1 Classes

A visualization of all these classes can be found at subsection 3.2.2

Employee
The employee is an abstract class from which "Accountant" and "Manager" inherit from. This class holds the essential information needed about both types of employees like their name and employment status.

Accountant
The "Accountant" role represents the employees at Attiri which have no administrative tasks. Employees with this role cannot view the tasks or information about their coworkers but can view their own tasks and report these to the system as finished as well as create posts for customers and themselves.

Manager
The "Manager" role represents the employees at Attiri which do have administrative tasks. The "Manager" class holds two attributes that are not shared with the accountant role; these are "AdminAccess" and "AuthorizedAccountant". These attributes give employees admin status, which gives them the ability to use every functionality in the program as seen in Table 3.3.

Customer
The "Customer" class represents the customers of Attiri and will not interact with the system in any way. This class holds information about the company, such as the company name, contact information of a company representative, a unique ID, an activity status and a list of projects.

Project
"Project" is connected to "Customer" and represents a customer task that needs completion. "Project" contains a list of subtasks which makes up the whole project and is only finished when all of the subtasks has been completed. This class also contains an ID, start date, deadline, a responsible

employee, and an attribute "TypeOfTask" which holds information about the type of project to be completed. Typically, the type of tasks dictates the nature of the subtasks. The responsible employee is tasked with ensuring the deadline of the project is not exceeded but is not responsible for completing the subtasks within a project as they can be assigned to different employees.

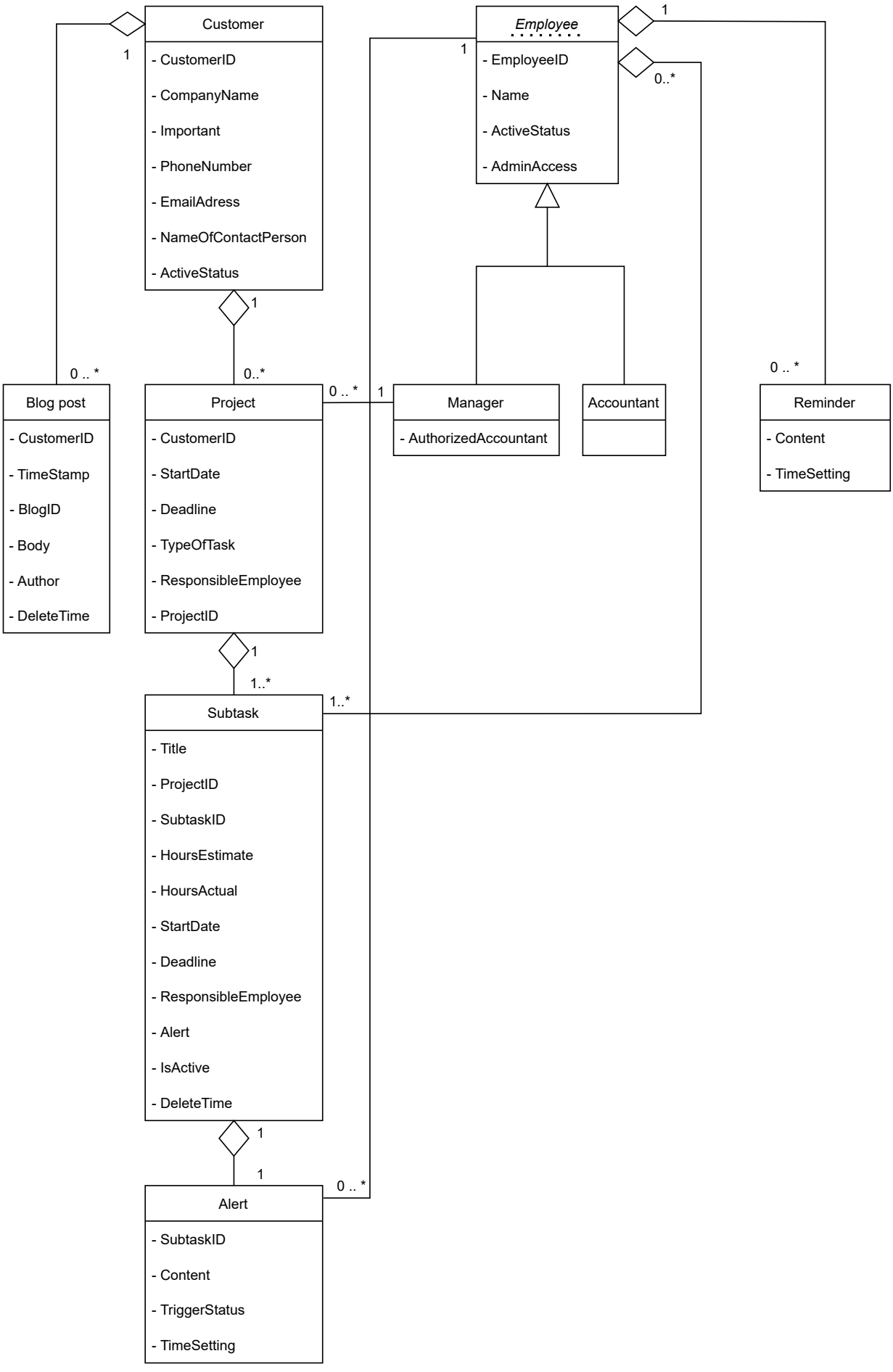
Subtask
The "Subtask" class represents the individual tasks to be completed within a project. A subtask contains a project ID, an "HoursEstimate" which is the estimated time to complete the task, based on experience. The attribute "HoursActual" is filled in after a subtask is classified as finished and represents the hours it took to complete the task, helping Attiri manage their time better the following year. "Subtask", like "Project", also contains a start date, deadline and a responsible employee. Uniquely for the subtask it also contains an "Alert" which is triggered when the deadline is approaching. The attribute DeleteTime is used for when subtasks are "deleted" and added to the trash bin, so that they can be sorted by deletion date within the trash bin.

Alert
The "Alert" class contains an alert text, a trigger status and a time setting. The time setting represents how long before a deadline the alert should trigger. When the alert is triggered the trigger status changes and sends the alert text to the correct employee.

Blog
The "Blog" class is connected to customer by a customer ID and is made up of a list of posts containing employee notes for the customer.

3.2.2 Class Diagram

After determining all the classes within the problem domain, a class diagram has been developed to visualize the relations and the attributes of the different classes.


Customer
- CustomerID
- CompanyName
- Important
- PhoneNumber
- EmailAdress
- NameOfContactPerson
- ActiveStatus
Employee
- EmployeeID
- Name
- ActiveStatus
- AdminAccess
Blog post
- CustomerID
- TimeStamp
- BlogID
- Body
- Author
- DeleteTime
Project
- CustomerID
- StartDate
- Deadline
- TypeOfTask
- ResponsibleEmployee
- ProjectID
Manager
- AuthorizedAccountant
Accountant
Reminder
- Content
- TimeSetting
Subtask
- Title
- ProjectID
- SubtaskID
- HoursEstimate
- HoursActual
- StartDate
- Deadline
- ResponsibleEmployee
- Alert
- IsActive
- DeleteTime
Alert
- SubtaskID
- Content
- TriggerStatus
- TimeSetting
1
0 .. *
0..*
1..*

An employee within the system can have different roles, which can either be as an accountant or a manager. The manager is usually working as an accountant within the firm and therefore has the same basic privileges to control the system as an accountant. As the manager is typically an accountant the manager can like the accountant be assigned subtasks, projects and has their own day schedule with tasks for each day. The difference between the employee roles is clarified in the Table 3.2 and state chart diagrams in the chapters below.

In Table 3.2 we see an "event table". The event table contains all the main events that can instantaneously happen in the system. It also contains all the classes that are prevalent in the system. The table shows which event is connected to which class either with an asterisk or a plus symbol. The table helps illustrate how we plan to construct the system. The plus sign symbolizes a one time event, where as the asterisk symbolizes an event that can occur multiple times.

Class/Event	Accountant	Manager	Customer	Project	Subtask	Blog	Post
Employed	+	+					
Resigned	+	+					
Project created		*	+	+	+		
Project assigned		*		*			
Project finished		*	+	+			
Subtask created		*		*	+		
Subtask assigned	*	*			*		
Subtask finished	*	*		*	+		
Customer created		*	+			+	
Customer deleted		*	+			+	
Customer active		*	*	+			
Customer inactive		*	*	+			
Alert triggered	*	*			+		
Blogpost created	*	*				*	+
Blogpost deleted	*	*				*	+

Table 3.2: Event table with the main classes and events in the system.

In Table 3.2 we can see that "Project created" is connected to the class "Manager" with an asterisk. This means that the class "Manager" can raise the event "Project created" multiple of times. We can also see that the event "Alert triggered" is connected to the class "Subtask" with a plus sign. This means that the class "Subtask" can only do the event "Alert triggered" once. This logic is drawn with the help of the diagrams in subsection 3.2.3 to easily visualize how the different classes interact with the different events in the system.

3.2.3 Statechart

In this section we will present the statechart diagrams which describe the general behavior of all objects in a specific class from our class diagram in subsection 3.2.2 and contain states and transitions between them. It is done by using the basic notation for the statechart diagram as described in the book Object Oriented Analysis & Design page 343 [10].

Project

The project class contains a unique ID, startdate and deadline which indicate the course of the project. Besides this, project has an attribute named "TypeOfTask" which categorizes the project. Finally we have the responsible employee attribute which describes which employee is responsible for the project handling.

When the project is created it goes into the active state. During that state, we can manipulate the values of the attributes. We can assign the responsible manager for the project, change the start date and deadline and create a subtask for the project. When the project "die" and that happens when it gets deleted or marked as finished.

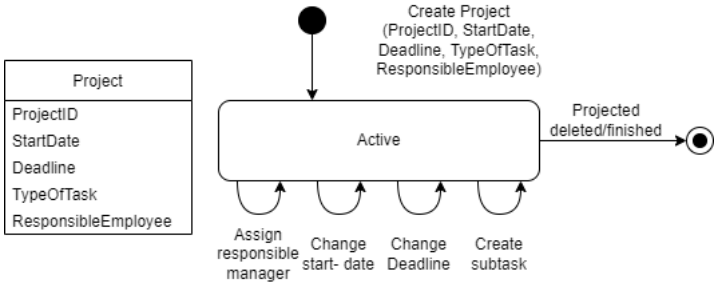


Figure 3.1: Statechart for the Project class

Subtask

The class has an object called subtask that becomes "alive" when it's been created as shown in Figure 3.2. When creating a subtask it comes with several attributes ProjectID, HoursEstimate, HoursActual, StartDate, Deadline, ResponsibleEmployee and Alert. A subtask should be part of one project with a specific ID. The subtask object has a responsible employee, start date and deadline as the project object, but the subtask object has three additional attributes:

- HoursEstimate contains the information about the estimated time to get the subtask done
- HoursActual describes the actual amount of time the subtask took to be solved
- Alerts give warning to the responsible employee.

When the subtask is created and goes into the active state, it is possible to assign the responsible employee, change Alert-date, change the expected time and change the deadline. It is possible to do all of that until the subtask become inactive or the subtask is finished, but the subtask will first completely "die" when the project is finished. While the unfinished subtask is still active and it's approaching its deadline, trigger the warning state which will create a new alert.

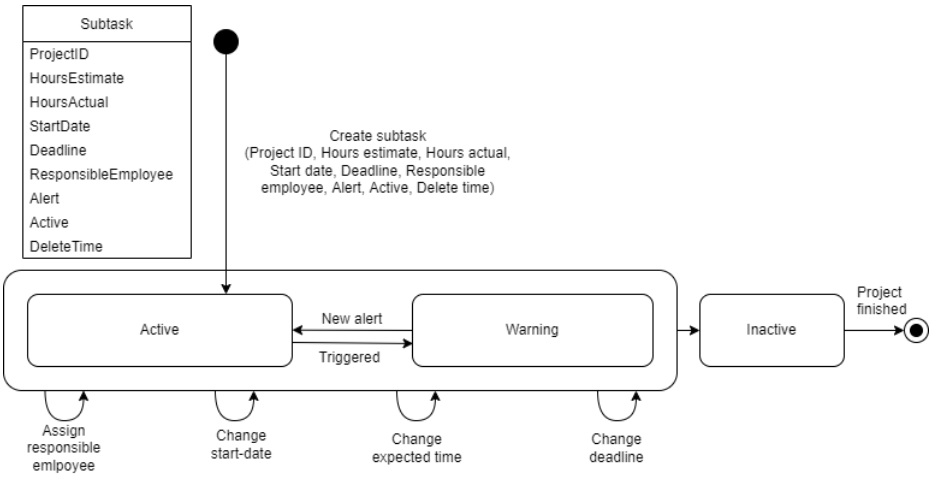


Figure 3.2: Statechart for the Subtask class

Customer

The class Customer contains an object customer that becomes “alive” when it’s been created as shown in Figure 3.3. It also has some attributes that contain information about the customer, such as Company name, Phone number, Email address, Name of contact person and ID. It also contains the IsActive with a boolean value that allows an overview about the customer’s status. When the customer object is created it is not necessary for it to be active for being “alive”, it can switch around between active and inactive and still events, like changing the customer information, creating a customer blog-post and deleting customer blog-posts can be used. Finally, the customer would “die” only if it becomes inactive within the system. This can only be done by manipulating the IsActive attribute in Uniconta.

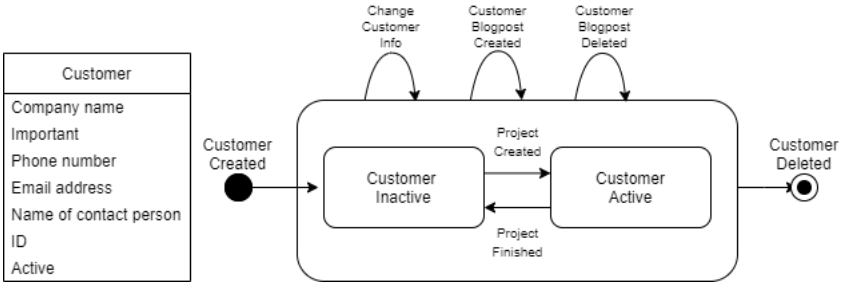


Figure 3.3: Statechart for the Customer class

Accountant

The Accountant class contain the attributes Name, Date, DaySchedule and ActiveStatus as shown in Figure 3.4. An instance of this class becomes "alive", when a new employee is added to the system. Every accountant has an ActiveStatus that can be switched between active and inactive, while still being employed at the same time. While the accountant is active they can be assigned to a subtask, finish the subtask and get alerts about the subtask when it is approaching its deadline. The accountant will “die” when they resign and set to inactive.

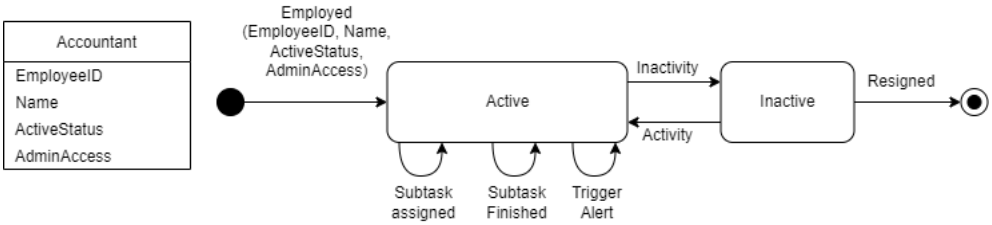


Figure 3.4: Statechart for the Accountant class

Manager

In the class Manager, an object manager starts being active when it gets employed, as shown in Figure 3.5. When the manager is active it has the same abilities as the accountant, but it is allowed to conduct additional operations, such as creating subtasks, adding new customers, deleting already existing customers and switching their statuses from active to inactive. Finally, the manager becomes inactive and "dies" when they resign.

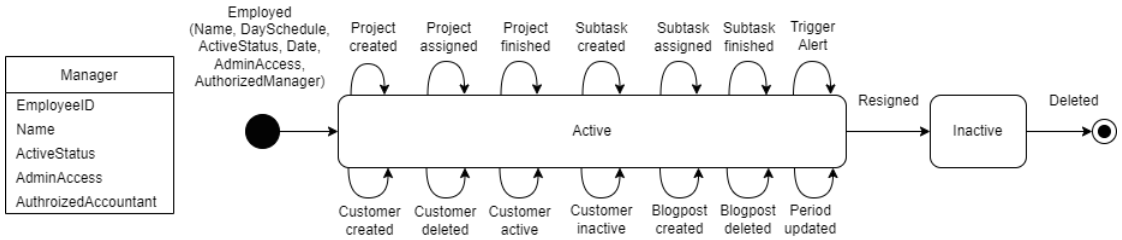


Figure 3.5: Statechart for the Manager class

It may appear by looking at the model that the system allows a user to assign/finish a project before even creating it. That is a misunderstanding, as UML(Unified Modelling Langauge) [10] does not directly distinguish between events which are contingent upon other events having occurred prior.

3.3 Application Domain Analysis

The purpose of the application domain analysis is to establish the actors that interact with the system, which commands and functionalities should be available, and which actors and systems are able to utilize the different functions, along with their objects and complexity.

3.3.1 Use Cases

This section is dedicated to the actors that use or interact with our system. The most relevant use cases will then be analyzed and explained. In Table 3.3 we see all the main use cases and actors.

Actor table	Accountant	Manager	UniConta
Login	x	x	x
Add/edit reminder	x	x	
Add/edit customer evaluation	x	x	
Add/edit subtask		x	x
Edit project		x	x
Edit customer		x	x
Assign subtask		x	x

Table 3.3: Actor table with each use case and their corresponding actor.

Login
The login event begins when the website is opened and the user has not previously logged in or if the user has logged out. When the user goes to the domain, a prompt will appear and ask if the user wants to log in. If the user clicks login the system will retrieve a unique login key from the user's Uniconta account. If the key is sufficient the user will log in. If not the system will try again and prompt the user that the login was unsuccessful.

Objects: Employee
Functions: Session storage, Uniconta browser login

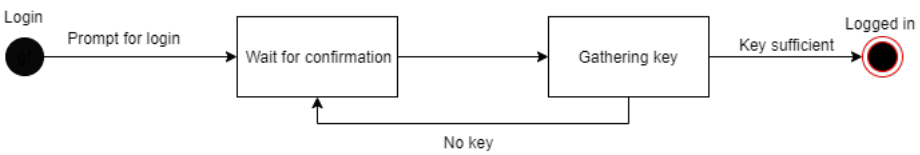


Figure 3.6: Use case diagram showing the process of logging in to our system.

Add/edit subtask
The employees with the correct credentials can go to the “Employee overview” tab. The tab will show a weekly work schedule for all accountants. If the employee would like to edit or add another subtask they simply have to click "Add/edit subtask". This bottom will allow the user to fill out a form, alike the one in Uniconta. After the changes have been confirmed the newly edited or added subtask will be filled out into Uniconta’s database. The updates will also be shown in our system wherever it is needed.

Objects: Manager
Functions: Time period, add/edit subtask, quick assign

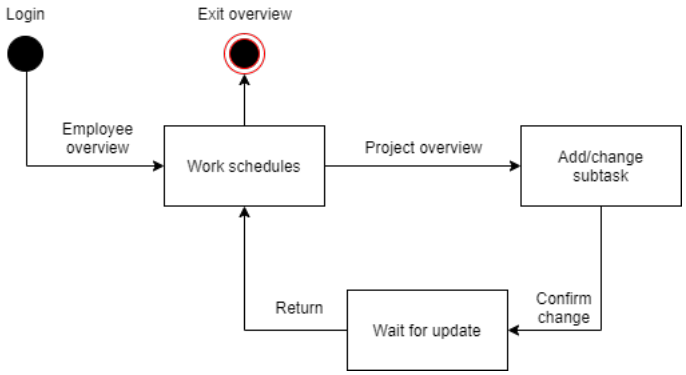


Figure 3.7: Use case diagram showing the process of editing or adding information to a subtask.

3.3.2 Functions

From our actor table and use case diagrams we can begin to map out which functions our system needs to have. Therefore, in this section we will list each main function, what the complexity is of that function and what type it is. The complexity scale goes from simple to medium to complex. We can rate a function by looking at its content, such as, how many hours do we think it will take, how much new information do we require to solve it, previous experiences and much more. Some of the functions require a further explanation and will be given in Table 3.4.

API GET/POST data
These two functions are the most trivial for our system to work. We need to extract data and post data into the other system - Uniconta. Therefore these two functions have been marked with complex. They are also utilized by most other functions, as they are likely to either extract data or post data. As an example the function "Add/edit subtask" needs to be able to able to write to Uniconta to be able to add a new subtask and the same goes with editing a subtask.

Employee and organization workload calculation
These two functions are marked as medium as we need to find a way to easily visualize what the employee and organization workload is and find a way to find and use that data. We also need to create these two functions from scratch as our version of a graph display is unique

Functions	Complexity	Type
API GET data	Complex	Read
API POST data	Complex	Write
UniConta browser login	Complex	Read
Session storage	Complex	Write
Employee workload calculation	Medium	Compute
Organisation workload calculation	Medium	Compute
Add/edit subtask form	Simple	Write/update
Add/edit blogpost	Simple	Write/update
Add/edit reminder	Simple	Write/update
Edit project form	Simple	Update
Edit customer form	Simple	Update
Time period	Simple	Read
Dashboard cycle	Simple	Read
Logout	Simple	Delete
Trigger alert	Simple	Read
Subtask finished	Simple	Write
Quick reassign button	Simple	Update

Table 3.4: Table over all of the main functions in our system, their complexity and type.

3.4 Component Architecture

This sections will introduce our system's component architecture. We will be following the generic architecture pattern [10]. This pattern is ideal as it allows us to split the work into different parts. This means that we can work on the front-end and back-end of the component simultaneously. The pattern separates the system into four main components: System Interface, Function, Model and Technical Platform.

As illustrated on Figure 3.8 the user is going to interact either with the Function, Model or directly with the Technical Platform. The user will e.g. interact directly with the Technical Platform whenever a user is going to login. The Model will interact with the Technical Platform every time new data is required.

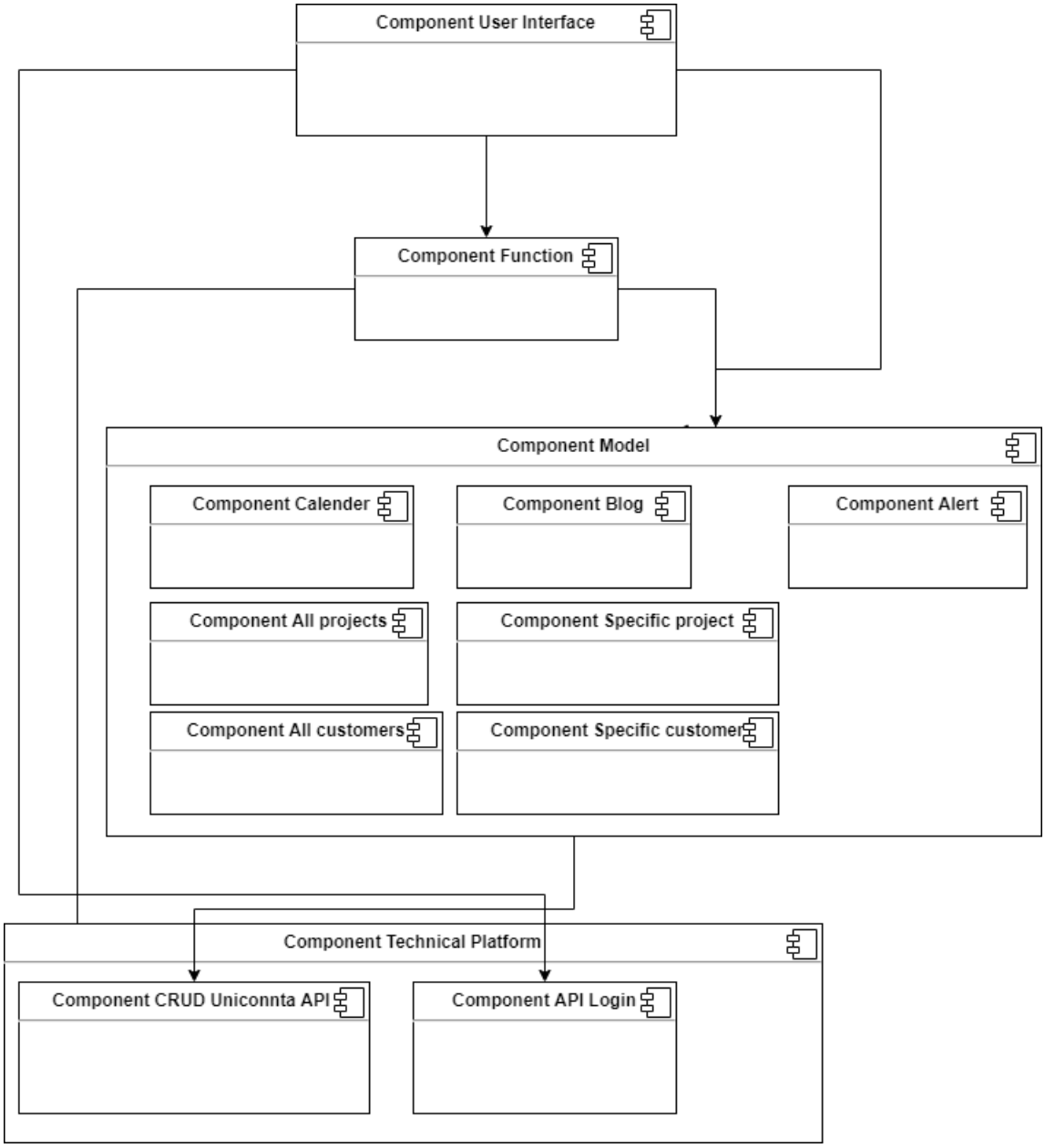


Figure 3.8: The Generic Diagram

We are not able to modify the technical platform which mainly consists of the database we are going to access through the API. As shown in the function Table 3.4 the API GET/POST functions are categorized as complex, mainly because of our lack of experience when it comes to creating API's. Challenges like this makes this component architecture really useful for us, because we are able to create some dummy access points and retrieve data, which means the possible challenges in creating the API wont stop our progress in creating the logic behind the components.

3.5 Component Design

In this section we have divided the generic component diagram from Figure 3.8 into smaller diagrams. We do this to simplify the components to get a better understanding of each component so they are easier to implement.

3.5.1 Calendar Component

The main goal of *Calendar component* is to give an overview over the work schedule for each of the employees in the firm during a given period of time, so the employee is able to see what subtask they have scheduled in that chosen period, and it can be e.g. a day or a week.

The *Calendar component*, shown on Figure 3.9, contains the components *Employee* and *Date*. The *Employee component* contains information about the specific employee, such as employee ID and working hours. The *Date component* is responsible for finding the period for the chosen date.

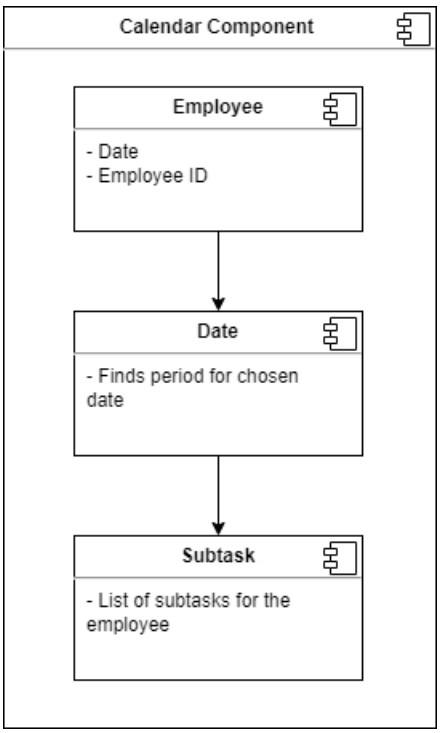


Figure 3.9: Calendar component

3.5.2 Alert Component

When a subtask is approaching its deadline, there should be sent an alert to the responsible employee. This is the main goal of the *Alert component*. When the alert is no longer necessary then it is deleted. The *Alert component*, shown on Figure 3.10, contains components *Fetching Subtask Details* and *Rendering Alert*. The *Fetching Subtask Details component* gathers information about a subtask via the API, such as its title and its description. The *Rendering Alert component* is responsible for rendering the alert and displays it only for the responsible employee.

The component also consists of two other components *New Alert Date* and *Dismiss alert* that contains two different functionalities. The *New Alert Date component* is responsible for overwriting the existing alert, while the *Dismiss alert component* is responsible for marking the subtask as finished. This is done by changing the alert date to NULL via API. When the alert is no longer needed, it should be removed, that is why both of the components *New Alert Date* and *Dismiss alert* are connected to the *Removing visualization alert component* that is responsible for deleting the alert.

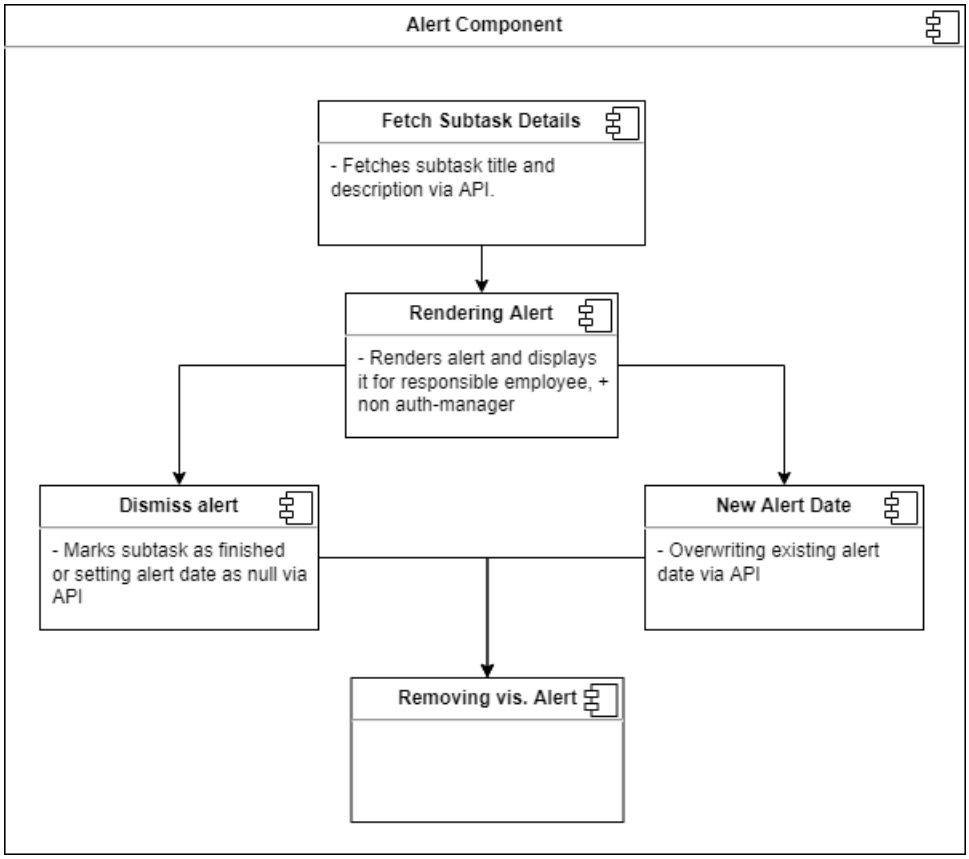


Figure 3.10: Alert component

3.5.3 Snackbar Component

The *Snackbar component* is used to generate and send notifications, with certain descriptions, depending on what type of task has been done. After some time it is set to automatically disappear. The *Snackbar component*, shown on Figure 3.11, contains components *Snackbar component*, *Render visualization* and *Remove visualization*. The *Snackbar component* contains information about the new snack-bar that will be created, such as the title and description. The *Render visualization component* is responsible for creating a visualization of the new snackbar that was created by the*Snackbar component*. The *Remove visualization component* is responsible for removing the visualized snackbar, it happens automatically after 30 seconds.

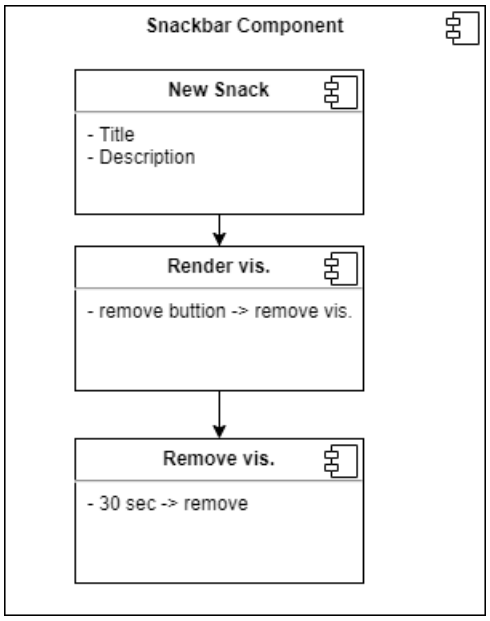


Figure 3.11: Snackbar component

3.5.4 Blog Component

The *Blog Component* has the function of allowing employees to post pieces of text in the form of blog posts, in order to create a place for internal communication in the company, allowing for custom notices in regards to customers. The *Blog component* as seen on Figure 3.12, contains several components that allows for its service. *Fetching blogpost* gathers the correct blog posts for a customer, using the customers ID in the database as a parameter. Next, the *Blogposts component* renders the blog posts and displays them to the user. The user interactability of the *Blog Component* lies in the *Add/Edit*, *Delete* and *Save* methods, which allow the user to post new blog posts, edit existing ones and once changes have been made, save them to the system. Lastly the *Snackbar component* handles notifying the changes made to the blog posts.

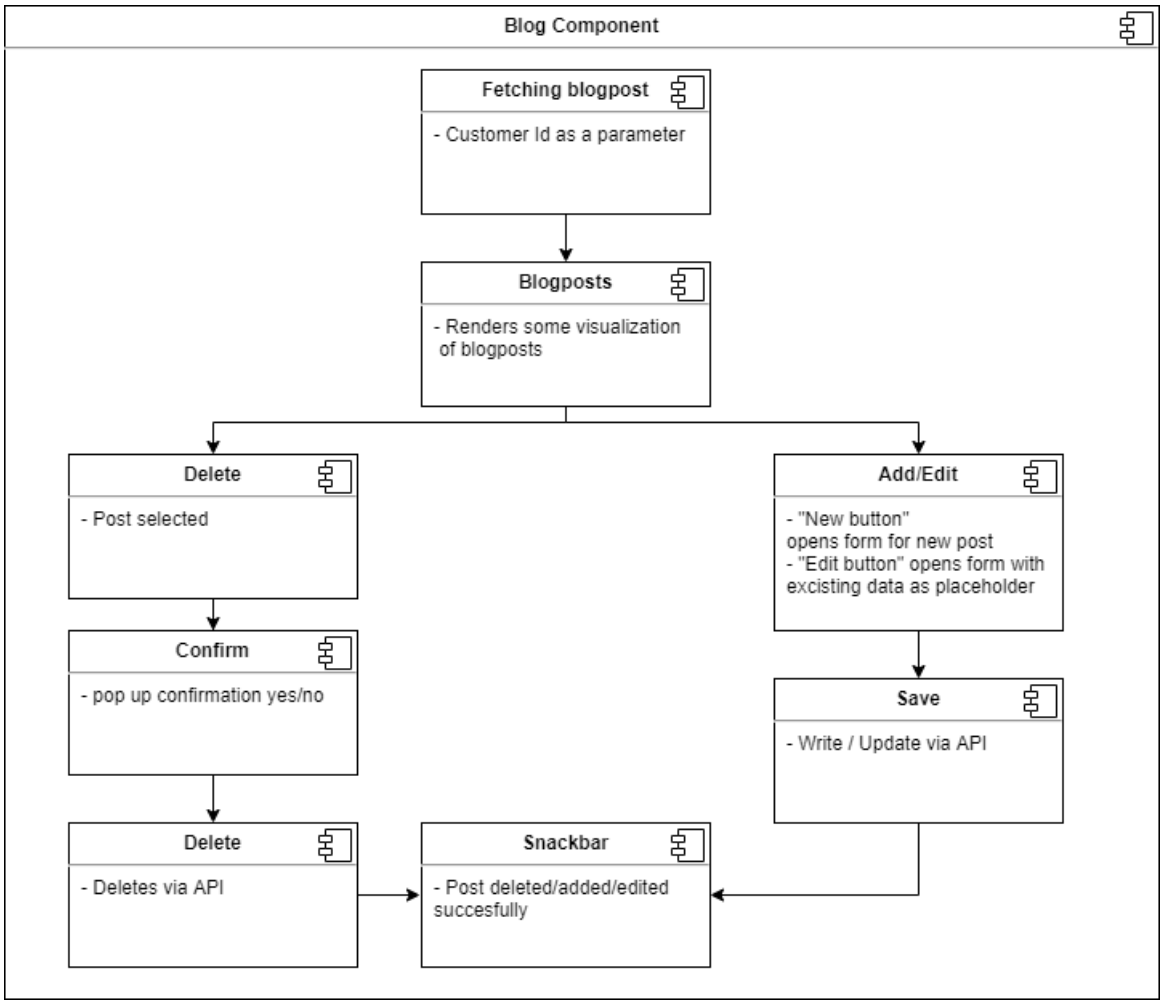


Figure 3.12: Blog Component

3.5.5 Specific Project Component

The *Specific Project Component* is in charge of fetching project data via the API, displaying the project and all of its relevant information, and allowed the user to add and edit subtasks that are attached to the project. *Fetch Data* and *Fetch Subtasks*, fetches the details from Uniconta via the API. The subtasks that are attached to a project, are connected via the ID of the project they belong to. The *Render Subtasks* component is in charge of displaying all of the subtasks that are attached to a project on the page. The *Add/Edit Subtasks* component allows the user to create a new subtask using a form that requires the user to enter all of the relevant data, such as deadline, type of task, responsible employee and date of which the work of the subtask can begin. Once subtasks have been completed, they can be marked as finished using the *Marking Subtask As Finished* component, which is a simple checkbox. Lastly, all changes made to a subtask can be saved using the *Save Subtask* component, which updates the changes in the Uniconta database, using the API.

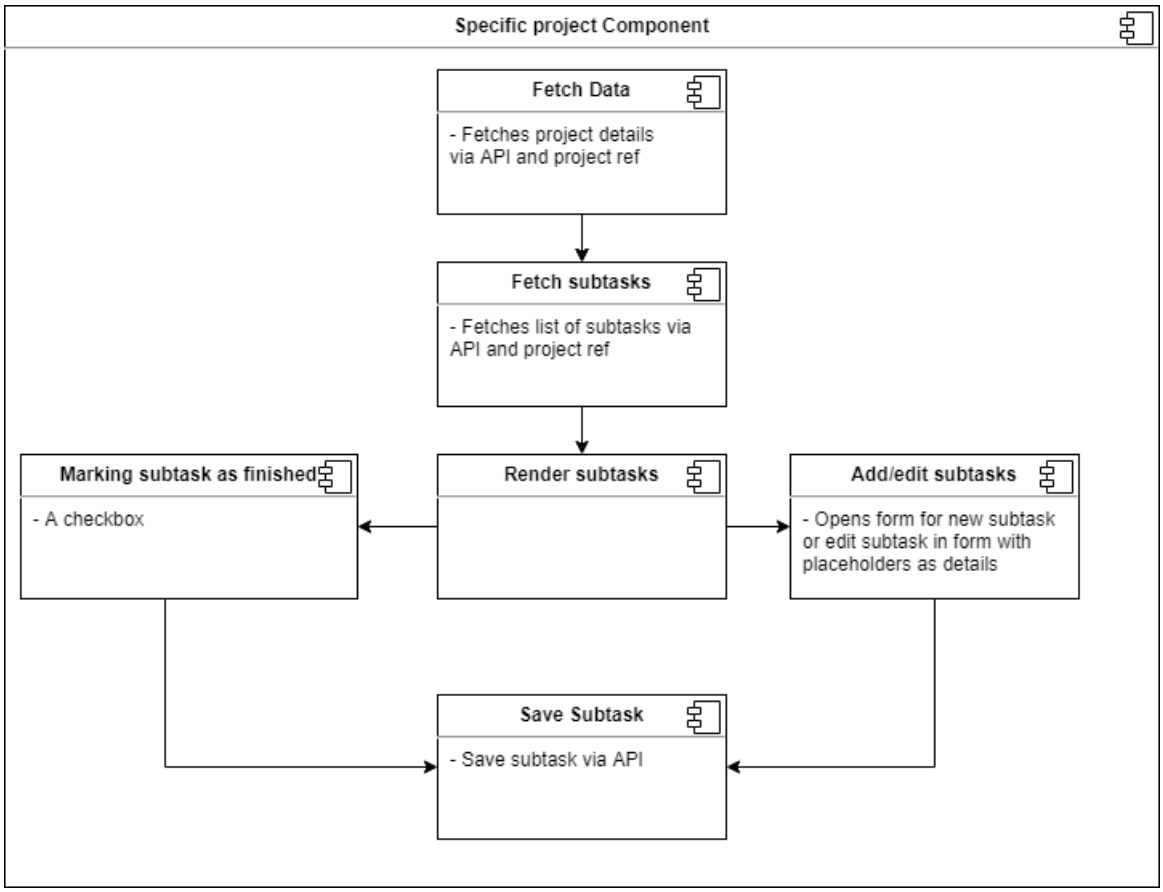


Figure 3.13: Specific Project Component

Chapter 4

Design of User Interface

This chapter focuses on the design sketch of the user interface and how Gestalt Laws are applied to make the interface more user-friendly and intuitive to the intended users of the system. The prototype has been shown and discussed in a meeting with one of the intended users and corrected accordingly to best reflect their needs. The section will also describe the layout choice and how it is intended to affect the user.

4.1 Prototype

The sketched web page is intended to be viewed on a large flat screen TV as well as on a normal desktop monitor. The manager and CEO of the company will be using the system and certain screens in one way, and the accountants have different pages that they have access to. The pages for the most part use some sort of table, in order to make the layout similar to their current methods of planning, which is mostly done in spreadsheets.

4.1.1 Dashboard View

The dashboard as seen in Figure 4.1, is intended to provide a quick overview of reminders, statuses and workloads. All the elements take up quite a lot of screen space, so that they are easily viewed at a distance. The web page also has a navigation menu on the left-hand side, that carries over between each screen. The color coding of the system is not shown in the sketches, which will mostly involve colors like green, red and yellow, due to their psychological associations. We use the associations of color like green for positivity, yellow for caution and red for critical. The layout color of the system has been chosen to be blue, due to its association with work and professionalism. We were tempted to select the layout color of green to match the logo of Attiri, but due to also wanting to use green in color coding, we felt as if it would drown out the intended use of the color.

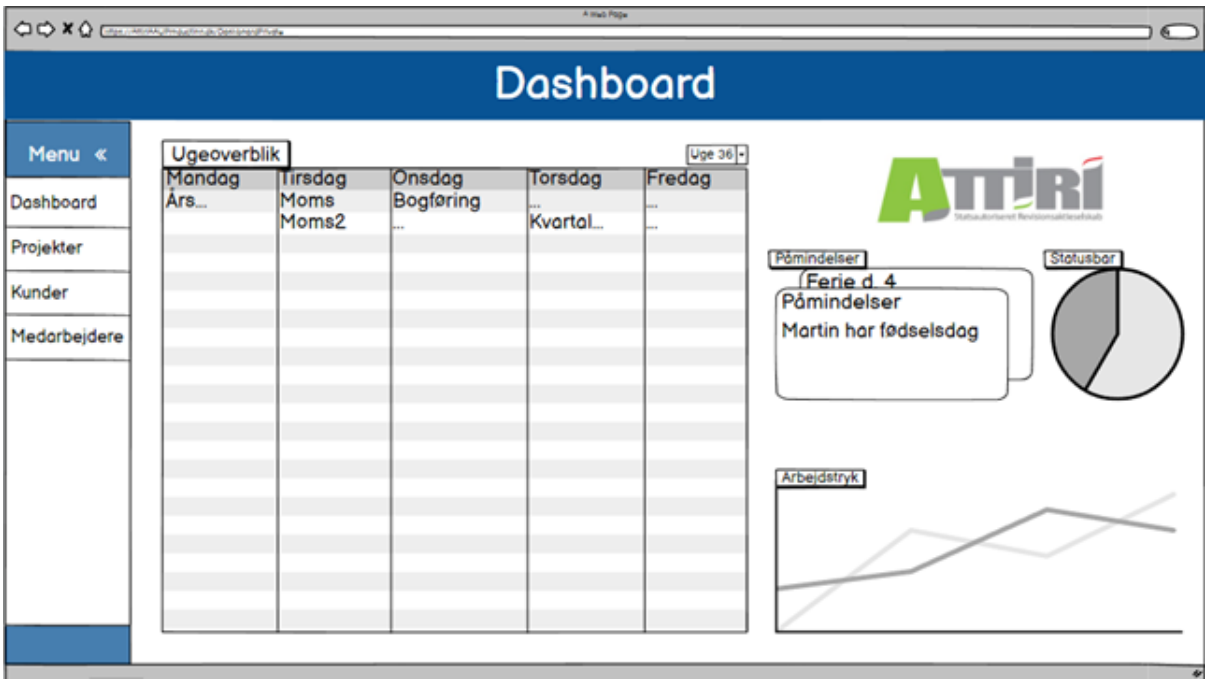


Figure 4.1: A sketch of the dashboard page.

4.1.2 Planning View

The employee view as seen in the Figure 4.2, is only intended for the manager and CEO of the company. This view will be restricted by the login system to any users that it is not intended for, since it displays sensitive data of the employees. On this page an overview is given of the scheduled hours for each employee and serve as a planning tool regarding getting an overview of which employees have hours available and which employees have too much work scheduled in a day. The system features automated alerts, which serve as a way of getting critical reminders and alerts to the attention on the manager and CEO as they plan their worker's days. The workload curve (*Arbejdstryk*) has been carried over from the Dashboard view. The period of which the planning data is displayed, can be changed with the *Uge* dropdown menu, and a different week can be selected. Additionally, the view can be changed to display a monthly period instead of a weekly one, using the *Uge/Måned* toggle button.

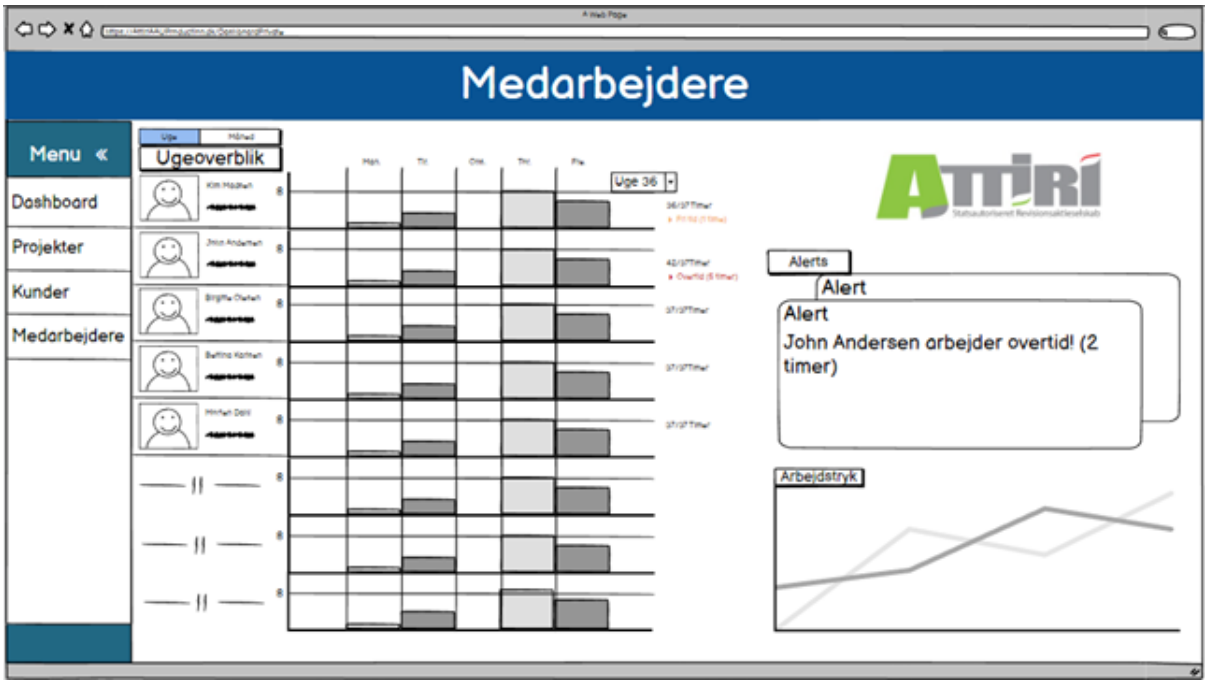


Figure 4.2: A sketch of the manager overview page.

4.1.3 Employee View

The employee view, which can be seen in Figure 4.3 is intended to be viewed by individual employee and displays data and tasks that are relevant for their workday, and to provide an overview of future tasks as well. This page can be accessed by the CEO, manager and the specific employee that the page details. This means that no other employees can access each other's personal view. The page displays a calendar showing the current day of the displayed data, the tasks that are to be completed on that day, personal reminders and a chart showing the scheduled days of the week, quite similar to the *Planning View* from before. Clicking on a chart will redirect you to the task page that you have selected. The calendar can also be navigated to display the scheduled tasks for any other day, by clicking on the desired date. The personal reminders feature a + button, that allows the employee to create custom reminders for themselves to be displayed on their page.

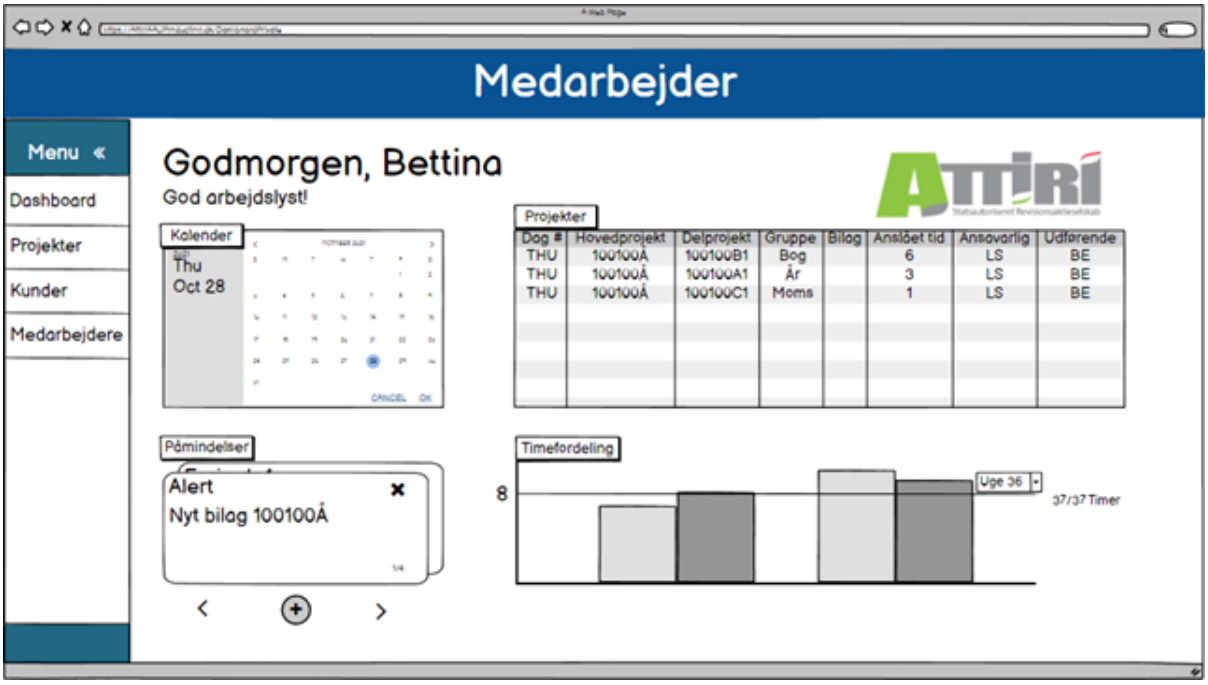


Figure 4.3: An overview of the employees' workload individually and combined

4.1.4 Project View

The project is accessible to all users of the system and displays the project of a customer and all its subtasks that must be completed in order to finish the project. Again, a table like view has been chosen in order to closely resemble the current planning system of the company as they have requested, as seen in Figure 4.4. The information in this view provides and overview of the overall data and displays some overall data of the active projects that the company must work with. The projects can be navigated by clicking the desired project, and the page will navigate to the specific project page, that will display the project and all the subtasks that are appended to it. Additionally, the system automated alerts only relevant to projects are displayed on the right-hand side of the screen and can also be used to navigate around the page with a click, which will redirect the user to the appropriate page of which the alert originates from.

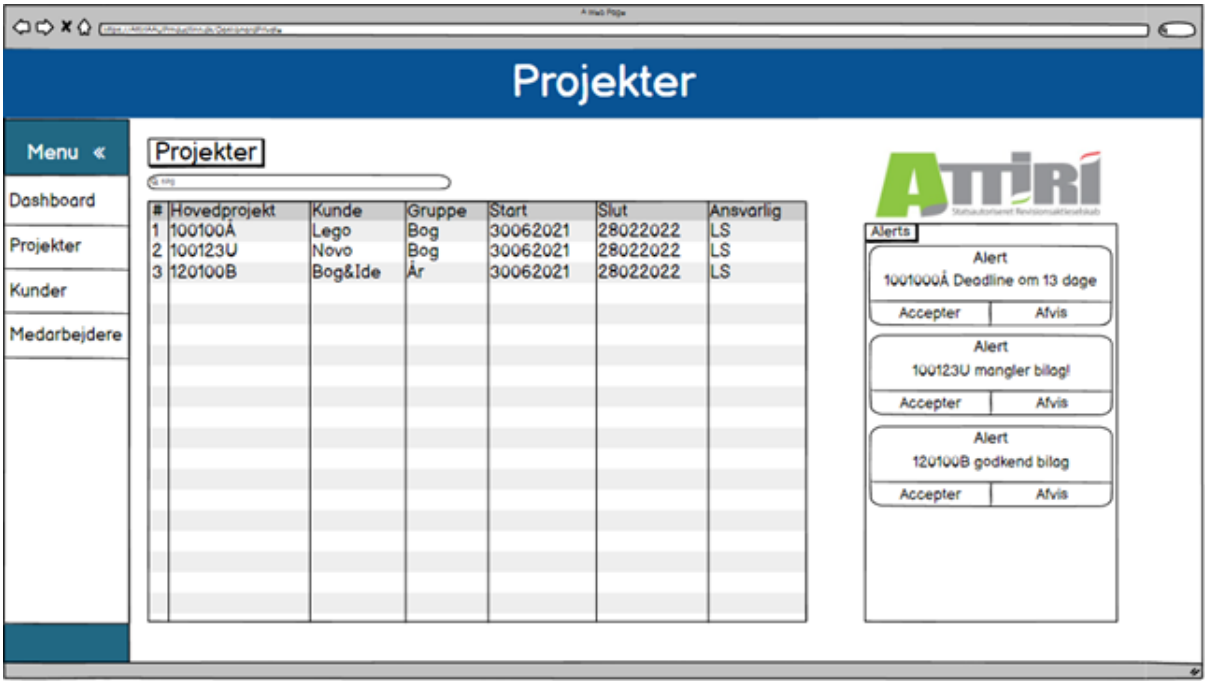


Figure 4.4: A sketch of the project page

4.2 Gestalt Laws

Gestalt Laws are principles that describe how the human eye perceives visual elements. They describe for example, how complex designs and patterns can be broken down into simpler shapes, or how the eyes perceive a selection of shapes as a larger structure [5]. The Gestalt Laws have been considered and applied in the design of the pages, based off theory and feedback from the company of which the solution is intended.

Similarity / Invariance

As mentioned previously in this chapter, the concept of similarity or invariance, has been heavily utilized in the use of tables in the design. The tables in the design are recognizable to the user, and every table serves the purpose of displaying data, as well as being used to navigate around the system. Every time a table is displayed, once the user understands that they can be used to navigate, it carries over the same usability, despite displaying different data depending on the view.

Figure/Ground

The buttons, alerts and reminders, all share some sort of outlined box, that allows them to be more visible to the user. The figure/ground principle details how the eyes can separate shapes from backgrounds. In the case of alerts, buttons and reminders it is important that simple shapes can be used to convey attention or separation in elements, so that they stand out to the user and implicitly show that they can be interacted with in one way or another, even if the background and the element share the same color.

Symmetry and order

In order to create a balanced design that, in this case, is specifically meant to be used to provide an overview at a quick glance, the system needs to follow the principles of symmetry and order in order to not cause confusion when viewed quickly, or require that the user has to analyze the page for an unnecessary amount of time. This is done in part by organizing the elements in an orderly manner, so that the dimensions of different elements line up properly, and are separated appropriately, so that they do not overlap, are skewed or leave too much unused space. An example is the Employee View at Figure 4.3, that neatly displays all the employees in a list and provides a quick overview of each employee. All the employees' scheduled hours in a week are displayed in a graph like design, which promotes the desire to have lines connect, by making sure that every employee has their days maxed out hour-wise, so that the graph is continuous, and no lines are broken. If an employee has too much or too little work in a day, this line is broken and automatically brings attention to itself, as it becomes asymmetrical and unorderly. This serves as a visual queue that there is planning work to be done.

Chapter 5

Implementation

For the implementation of our project, we built the application using the web framework Blazor which is part of the open-source .NET platform. Blazor is a client-side framework but supports a server-side version, allowing Blazor components to be handled server-side on .NET Core while the UI-updates and events are handled using a SignalR connection over the network [4]. As our application requires a lot of updates and events, we found that Blazor is perfect for handling updates without disturbing the workflow of the users. This chapter details which components constitute the program and their integration with one another. How many of these components interact with the Uniconta database using controllers will also be explained.

5.1 Component Life Cycle

A Blazor application is built using .Razor components, which are created with Razor allowing programmers to mix HTML and C#. Razor components have some logic written in C# connected to the UI written in HTML. This means that it is possible to build a UI based on C# functions and binding values in the front end to properties in our models. These razor components can be nested and reused as well. Each page in Blazor is a component, with an "@page" attribute that specifies a URL to access the component.

Each Blazor component has its own life cycle. This life cycle makes us able to decide what order different actions should be executed. All Blazor components inherit from "ComponentBase", which is a set of methods and tasks, defining the different stages in the life cycle.

We mainly use the following tasks:

- OnInitializedAsync();
- OnAfterRenderAsync(bool firstRender);

Since all of these asynchronous methods are virtual, we are able to override them and adapt them to our specific needs.

5.1.1 OnInitializedAsync

The OnInitializedAsync method is called when the component has received its initial parameters from the SetParametersAsync method. We are mainly using the asynchronous version of OnInitialized since we execute asynchronous operations.

5.1.2 OnAfterRenderAsyn

The OnAfterRenderAsync method is called when the component has been rendered. We will be using this method to check if the user has access to this page. This will require a session store, which is accessed via JavaScript. This is the first point in the life cycle where we can perform the "JSInterop" operation (A JavaScript invoked operation).

5.2 Login Layer

To ensure that no users outside of Attiri can access their data, we need to implement a login system. We need this system to be able to differentiate between the types of employees and their different

levels of access. In order to do this, we will mostly rely on the login of Uniconta's login system via the API. This means, if a login is valid in the Uniconta system, it will be valid in our system as well. The login layer in the system must give the employees access if they have valid credentials, but it also must render different elements in the UI depending on the level of access. To do this, we are going to use session storage.

5.2.1 Session Storage

Blazor can generate session storage which contains session details. These session details will contain following information:

- Username
- Password
- Login time
- Access Level
- Employee RowId

It is necessary to store the username and the password in the session store since the API needs the credentials to be able to GET, POST and PUT data. Since the session storage is stored locally in the browser, each different client will by default be able to edit their session storage. This makes a user able to change their access level, once they have logged in to the system through injection. Microsoft has created a solution for this problem. Instead of saving the session details in plain text, we will be using "ProtectedBrowserStorage", which encrypts the data in the browser storage.

Saving data via the "ProtectedBrowserStorage" can be done in two different ways; via the local storage or the session storage. The difference between the two options is; the method of storage, how long the data is stored and how the data can be accessed. With local storage, the data can be accessed by different tabs in which the session storage is generated for each tab, and the data only remains available in a specific tab.

	Reloads Tab	Closing Tab	Closing Browser
Session Storage	Persists	Lost	Lost
Local Storage	Persists	Persists	Persists

Table 5.1: Table showcasing the difference between session storage and local storage.

Based on Table 5.1 above we have chosen to use session storage. This means that the user will have to log in every time they open up the system, due to accessing sensitive information. The security is deemed more important than convenience, since our login is fast and convenient (e.g. no two-factor validation is implemented).

5.2.2 Login Validation

The login will be passed through Uniconta's API. The potential match of the user's login credentials will be stored in the server session storage, along with the following details:

1. Access level
2. Login time

Uniconta creates a log entry every time a login is successful. We utilize this log so that if a log is successfully created the login will be accepted. We are able to do this because the user's login credentials are the same in our proposed solution as in Uniconta. Validation will only occur when they log in and create the session store, and not every time they navigate to a new page.

Every time they open a new page a security check will be made. This is only a validation of the user's access level. The access level in the protected session storage will be compared to the access level of the user stored in the database. This guarantees that no employee can change the set access level and then access any information that is restricted. The client side session store is encrypted to avoid security issues.

5.2.3 Login Layer Implementation

Login

First are we making sure the user has entered some credentials, so we don't try to login without a password or a username. We are doing this by using the DataAnnotation namespace provided by Microsoft. This gives us the ability to tell the user what the errors are with the error messages at line 3 and 5 on Figure 5.2 and we can see the output on figure Figure 5.1.

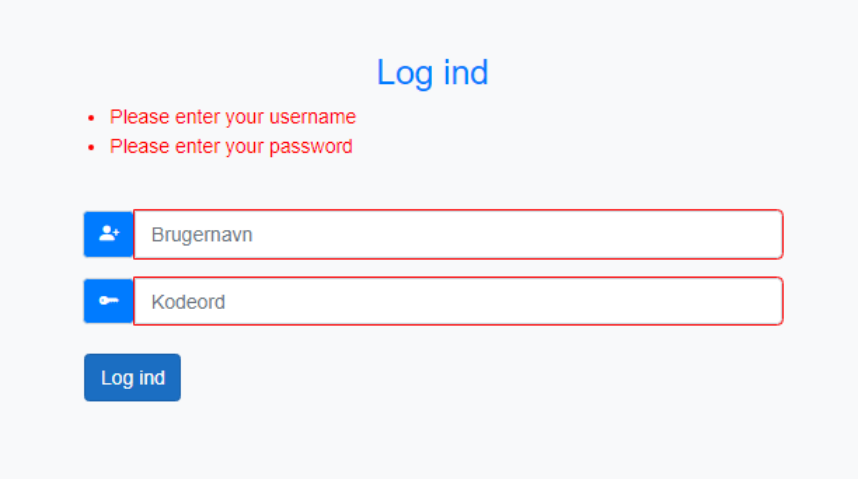


Figure 5.1: login page showing error messages

```
1  public class LoginInfo
2  {
3      [Required(ErrorMessage = "Please enter your username")]
4      public string Username { get; set; }
5      [Required(ErrorMessage = "Please enter your password")]
6      public string Password { get; set; }
7  }
```

Figure 5.2: Frontend Login Validation

When the user has entered a username and password the actual login validation method will start. Validating the login credentials are via the login log as described in subsection 5.2.2. The login is validated via the POST API endpoint, where the user login credentials are needed. Normally (every other time than at the login) the credentials will be fetched through the encrypted session store and the session store is created at a valid login. This means that the login credentials are passed as parameters to the API endpoint. This can be seen at line 4 at Figure 5.3. The ValidAccAsync method takes the protected session as a parameter, on line 1, even though the user's credentials are passed in as well. This is necessary because the controller, which is using the API's GET, POST

and PUT capabilities, uses the protected session storage. The ValidAccAsync returns true based on whether the credentials are valid or not.

```
1  public async Task<bool> ValidAccAsync(ProtectedSessionStorage protectedSession)
2  {
3      LoginLogController controller = new LoginLogController();
4      if (await controller.NewLoginAsync(logininfo.Username,logininfo.Password, protectedSession))
5          return true;
6      else return false;
7  }
```

Figure 5.3: Validation of Account

If the credentials are valid and the system can POST the log to the database, the user's credentials and login time will be set in the session store as seen on line 3 in Figure 5.4. The method takes two parameters. The first is "login" which is the key. We will be using it when we retrieve the data from the session store again. The other is the data that we want to store. In this case, it is the login.SetSessionStorage() method which is returning the credentials and login time as a string. When the session store has been set, the user is redirected to the dashboard page by the NavigationManager, which is a service for redirection.

```
1  public async void SetStorage()
2  {
3      await protectedstorage.SetAsync("login", login.SetSessionStorage());
4      NavMan.NavigateTo("/dashboard");
5  }
```

Figure 5.4: If the login is valid the session store is set

Login Validation

Every time the user navigates to a new page the system checks if the required access level for the page matches the user's access level. This check is seen on line 6 in Figure 5.5. The method takes two arguments, the first is the session store where the access level is stored. The second is the page routing for finding the required access level, which we compute in the OnAfterRenderAsync method as part of the component life cycle described in subsection 5.1.2. If the user does not have the required access the user will be redirected back to the either the login page (in this case the index page) or to the NoAccess page. This can occur if the user tries to route directly to the URL without logging in or if the page requires a higher access level than the user has. We use the NavigationManager for redirection. Beyond the redirection, the user will get a message through the snackbar, informing them that they do not have access to the page. The snackbar is a Javascript function being called at line 17 which serves the purpose of rendering the snackbar.

```
1      private bool Access { get; set; }
2      protected override async Task OnAfterRenderAsync(bool firstRender)
3      {
4          if (firstRender)
5          {
6              Access = await loginValidationService.Access(protectedstorage, "dashboard");
7              StateHasChanged();
8          }
9          if (!Access)
10         {
11             NavMan.NavigateTo("/");
12             Snackbar("You dont have access!", "error");
```

```
13          }
14      }
15      async Task Snackbar(string message, string status)
16      {
17          await JS.InvokeVoidAsync("Snackbar", message, status);
18      }
```

Figure 5.5: Access Validation

The loginValidationService.Access method consist of the two parts. One part of parsing the data from the session store and one for validating the data. It returns true or false based on the outcome of the HaveAccess method in Figure 5.6, which takes two arguments: the session object with all the information stored in the session store, and the page routing. If the AccesslevelCheck and the TimeoutCheck return true on line 3, the user haves access.

```
1  private bool HaveAccess(Session session, string page)
2  {
3      if (AccesslevelCheck(session, page) && TimeoutCheck(session))
4          return true;
5      else return false;
6  }
```

Figure 5.6: Have Access method

The AccesslevelCheck is handled by a switch, on line 3 in Figure 5.7. The switch contains the different pages which require an access level. If the user is trying to access one of these pages the required access level will be compared with the user's access level. The users access level is stored as a string, because it is stored as a string in the session store, which is why it must converted to an integer at line 7 in Figure 5.7. If the page does not specify required access level, the access level check will be successful no matter the user's access level. This is seen on line 10 at the default case.

```
1  private bool AccesslevelCheck(Session session, string page)
2  {
3      switch (page)
4      {
5          case "employees":
6          case "unassigned":
7              if (Convert.ToInt32(session.AccessLevel) == 2)
8                  return true;
9              else return false;
10         default:
11             return true;
12     }
13 }
```

Figure 5.7: Access level Check

The timeout check has been implemented because the system can access sensitive information, and one of the purposes of the system is the dashboard, which is supposed to be running throughout the day. The timeout check is making sure no one can access this information in the case that an employee forgets to logout of the system or so. The session object which is created via the session store is containing the time for the login. If the login occurred more than 8 hours ago, the user will not be granted access. This can be seen on line 1 in Figure 5.8.

```
1  private bool TimeoutCheck(Session session)
2  {
3      if (Convert.ToDateTime(session.LoginTime).AddHours(8) >= DateTime.Now)
```

```
4           return true;
5       else return false;
6   }
```

Figure 5.8: Timeout check

5.3 API

To be able to visualize the relevant data which Attiri has, is it necessary to get the data. All their data is stored in the cloud, which has an API. The cloud solution stores all their data on their clients using the Microsoft developed Open Data Protocol (OData). OData is an application-level protocol for interacting with data via RESTful principles. Since it follows the RESTful principles [9] the data platform is independent, which gives us the ability to get the data, even though we are using a relatively new framework.

Since our solution should be accessible on the web, and Attiri will not have any extra databases neither cloud based nor locally, we need to use the existing database, for the new data we will produce in this system.

5.3.1 Base API

Regardless of which API endpoints we access, some of the functions are the same:

- All the API connections needs a header containing credentials to the system
- OData contains all their data in JSON format – which means everything must be either serialized or deserialized.
- The data stored in OData keeps a lot of redundant information and references to other tables which we are not interested in, and we must post some data which the user do not know anything about. In addition to this the database stores some metadata about the entities. This means we must convert data input to be the OData class type, and vice versa.

Due to the above stated facts, we can create an abstract class, use the protected access modifier, and then use inheritance to make sure only the implementation of the specialized APIs (GET, POST and PUT) will be able to access these methods. By doing this we avoid implementing the same functions repeatedly.

5.3.2 Get API

The access we have to ODatas database is only partly fulfilling the RESTfull principles. We are only able to select a single entity based on the RowId, and can't create filters. This forces us to fetch the full table and then create the filters locally. If our access to the database gives us the ability to filter the entities before fetching them, we would be able to create a better implementation of the Get API.

Since we only want to implement a single version of the Get API it is necessary to make the API returning a generic list of elements. Luckily we are able to do this since all the table classes are available in a NuGet package belonging to the system, and we are able to generate the classes via the system they were created in. We are able to fetch data regardless of the type because we are passing the type as a argument to the method as in line 1 on Figure 5.9. The URL where the method fetches the database is also based on the type, this is done by using the GetDataType method and then concatenate it with the base URL on line 5 of Figure 5.9. Because all the data is stored in the JSON format, it is necessary to parse all the data, this is done by the ResponseToUCDatamodel

method on line 14. This parsing method is only necessary in the GET endpoint, because we need the data and not just a status code telling us if it was a success or not.

```
1  public async Task<List<T>> GetEntitiesAsync<T>()
2  {
3      string entityType = typeof(T).ToString();
4      string entity = GetDataType(entityType);
5      string url = "https://odata.uniconta.com/api/Entities/" + entity;
6      HttpRequestMessage request = CreateRequestMessage(url, HttpMethod.Get);
7      List<T> result = new List<T>();
8      try
9      {
10         HttpResponseMessage response = await Http.SendAsync(request);
11         string responseBody = await response.Content.ReadAsStringAsync();
12         result = ResponseToUCDatamodel<T>(responseBody);
13     }
14     catch (Exception ex)
15     {
16         Console.WriteLine(ex.Message);
17     }
18     return result;
19 }
```

Figure 5.9: Get List of entities

The method of fetching all the entities in the database and fetching a single entity are very much alike. There are only three differences:

- Get SingleEntityAsync only returns a single entity, which is shown on line 1 in Figure 5.10
- Entities are specified in the URL, this is done by a concatenation of the URLs on line 6.
- Even though we are only fetching a single entity we are placing the entity on a list, because the responseBody in the argument is indicating the objects are in a list. Therefor we are using a linkstatement .FirstOrDefault which is returning the first element matching the criteria (we have no criteria and there is only one element on the list). This can be seen on line 10 and 11 in Figure 5.10

```
1      public async Task<T> GetSingleEntityAsync<T>(int rowId)
2      {
3          string entityType = typeof(T).ToString();
4          string entity = GetDataType(entityType);
5          string url = "https://odata.uniconta.com/api/Entities/" + entity;
6          url +=  "/" + rowId;
7          ...
8          List<T> list = JsonConvert.DeserializeObject<List<T>>(responseBody, GetJsonSettings());
9          var result = list.FirstOrDefault();
10     return result;
11 }
```

Figure 5.10: Get single entity

The downside of using these premade classes are the amount of redundant and irrelevant data, we now have to pass around and loop through. Therefore we create a method which constructs a list with a single class modeled by us which only contains the data we need.

5.3.3 Post API

When we construct objects, only some of the data is given by user input. Some of the data e.g. the timestamp, for when the blogpost was created, is set by the system. Most of the classes for the database are found in the NuGet belonging to the system. We are not able to create constructors for the classes because we do not have access to edit these classes. Therefore we have created a new class for each of the tables for which we want to post data to their table. Within this class, we are able to create the properties for the user given input and then create a method which returns the class of the table with the user-given input and the automatic generated data. This can be in Figure 5.11 on line 1, 2, 3, where we have the properties, for which the user is supposed to give input. On line 11 the timestamp is generated automatically in a method which returns an object, which contains all of the correct input and can be posted to the database.

```
1  public string debetorRef { get; set; }
2  public string blogPost { get; set; }
3  public string author {get; private set; }
4  public BlogPostAAU(){
5  }
6  public Blogpostclass ConstructDataModel() {
7          return new Blogpostclass() {
8                  DebetorRef = debetorRef,
9                  Blogpost = blogPost,
10                 Author = author,
11                 TimeStamp = DateTime.Now
12             };
13 }
```

Figure 5.11: AAUConstructor for blogpost

When the data has been converted to the correct type, is it ready to be posted to the database. This happens with the InsertAsync(T DataToInsert) method, as in Figure 5.12. This method is generic, as well as the Get endpoint, which makes us able to insert all kinds of data. Since all the data in ODatas database are being stored as JSON it is necessary to convert our object to JSON format, this is happening on line 11. Because we are inserting data, we are returning a HttpStatuscode to indicate if the data has been posted with success. Based on the statuscode we are able to inform the user, if an error has occured, and what kind of error.

```
1  private async Task<HttpStatusCode> InsertAsync<T>(T DataToInsert)
2  {
3      string url = FindUrl(DataToInsert.GetType().ToString());
4      HandleHttpsRequestError(url);
5      HttpRequestMessage request = CreateRequestMessage(url, HttpMethod.Post);
6      request.Content = new StringContent(JsonConvert.SerializeObject(
7                              DataToInsert, GetJsonSettings()), Encoding.UTF8, "application/json");
8      HttpResponseMessage res = await Http.SendAsync(request);
9      return res.StatusCode;
10 }
```

Figure 5.12: API Post method

5.3.4 Put API

The purpose of the Put API endpoint is to update a single entity in the database. Attiri has a "never-delete-data-policy". This means that they preserve all of their data, even if it isn't in use. As an alternative they use a boolean to specify if the entity is active or not. In this way our PUT endpoint works as our delete endpoint as well. From a technical point of view this is an optimal

way of structuring a database, but Attiri has specifically requested that a delete feature shouldn't be included in the solution.

As previously described, the database is based on types generated by their existing system which contains a lot of metadata. Some of these metadata attributes are useful for us, one of them is the "RowId" attribute. The "RowId" is a unique ID for each entity in each table but is read-only. The value of the attribute is required since the database has the "RowId" as a primary key on all of their tables, and it is the only attribute which is completely unique for each entity. Because of the read-only access modifier, we need to set the value using a different method. The way we have chosen to do this is by fetching the single entity again, and then making the changes directly into the instance of the object. In this way the object's read-only attributes are set by the database, which makes us able to update a single entity after we have made the changes.

As described in subsection 5.3.1, we are converting the data used from the OData type to our locally created type and vice versa. Via our locally created type we are able to fetch the single entity, via the attribute "RowId" and LINQ statements. This is the main use of the method GetSingleEntity described in subsection 5.3.2.

The actual update process of the database is nearly identical to posting data in the database. The difference is the URL and the HTTP method, which is in charge of updating entities. We are modifying the URL by using two methods; "URL.Remove" and "url.Insert" as seen on line 3 and 4 in Figure 5.5. This is necessary since the "FindUrl" method called on line 2 is returning the URL for posting data. The URL for updating is almost identical, with one exception being the routing "Insert", which is instead set to "Update". We can modify the link in this way because we know the URLs are identical regardless of what type of data we want to insert or update.

```
1 private async Task<HttpStatusCode> UpdateAsync<T> (T DataToUpdate)
2 string url = FindUrl(DataToInsert.GetType().ToString());
3
4 url = url.Remove(40, 6);
5 url = url.Insert(40, "Update");
6 ...
```

Figure 5.13: Extract of UpdateAsync method

5.3.5 API Controllers

As OData keeps some redundant data, it can be challenging during development to avoid storing data in one table and trying to read it from another table. We have created six controllers so we only have one place in the source code to find functions for accessing the database. This ensures that we always read and write to the correct table. We have added a controller for a selected number of classes from our model:

1. Alert (associated with the table *AlertsForSubtaskUser*)
2. Blogpost (associated with the table *BlogpostUser*)
3. Customer (associated with the table *DebtorClient*)
4. Employee (associated with the table *EmployeeClient*)
5. Project (associated with the table *ProjectClient*)
6. Subtask (associated with the table *SubtaskUser*)
7. Reminder (associated with the table *ReminderUser*)

Each of these seven classes contain a constructor that takes an object of Uniconta's data model as a parameter. We can use the properties of this object to set the properties of an object from our model. An example of this can be seen in Figure 5.14.

```
1  public Customer(DebtorClient ucCustomer)
2  {
3      CustomerID = ucCustomer.Account;
4      CompanyName = ucCustomer.Name;
5      IsImportant = false;
6      PhoneNumber = ucCustomer.Phone;
7      EmailAddress = ucCustomer.ContactEmail;
8      ContactPersonName = ucCustomer.ContactPerson;
9      IsActive = ucCustomer.Group == "99" ? false : true;
10     Blog = new List<Blogpost>();
11 }
```

Figure 5.14: Constructor from Customer class

OData stores metadata about the entities that fit in a given table. We are only able to read and write to the database if we use objects that are based on this metadata. If we make a get request to the URL *https://odata.uniconta.com/api/entities/DebtorClient* the response will be an array of JSON containing objects of type *DebtorClient*. Through Uniconta we can generate the associacted C# class and serialize a *DebtorClient* object for each object in the JSON array. To use these objects in our application, we instatiate new objects from classes in our model, using the properties of the objects from OData. On line 8 in Figure 5.15 we instantiate a *Customer* object (our model of a customer) for each *DebtorClient* object (Uniconta's model of a customer) in our response.

```
1  public async Task<List<Customer>> GetCustomers()
2  {
3      List<DebtorClient> ucCustomers = await API.GetEntitiesAsync<DebtorClient>();
4      List<Customer> customers = new List<Customer>();
5
6      foreach(DebtorClient ucCustomer in ucCustomers)
7      {
8          customers.Add(new Customer(ucCustomer));
9      }
10
11     return customers;
12 }
```

Figure 5.15: Controller for Customer class

In addition to reading data, some of the controllers are also capable of inserting and updating data. This is required for numerous operations in the application. Examples of these would be creating and assigning subtasks to employees or posting and updating blog posts for customers. To insert an object into a table in OData, the object needs to be based on metadata that OData provides. Therefore, a function is needed for instantiating an object of OData's data model, based on a corresponding object from our data model. For example, an object of type *Subtask* (our model of a subtask) will have to be converted into an object of type *SubtaskUser* (Uniconta's model of a subtask) in order to fit into the table in OData. Controllers that are capable of writing to the database contains constructors for these objects. An example of this can be seen in Figure 5.16. The function *ConstructDatamodel()* is parameterized with an object of type *Subtask*, which contains the data we want to insert. On line 3 we instantiate a *SubtaskUser* object. We use the properties of the given *Subtask* to set the properties of the *SubtaskUser* object.

```
private SubtaskUser ConstructDatamodel(Subtask subtask)
        {
            SubtaskUser ucSubtask = new SubtaskUser();
            ucSubtask.SubtaskDesc = subtask.Description;
            ucSubtask.StartDate = subtask.StartDate;
            ucSubtask.Deadline = subtask.Deadline;
            ucSubtask.ResponsibleEmployee = subtask.ResponsibleID;
            ucSubtask.HoursEstimate = subtask.HoursEstimate;
            ucSubtask.HoursActual = subtask.HoursActual;
            ucSubtask.IsSubtaskDone = subtask.IsDone;
            ucSubtask.ProjectId = subtask.ProjectID;
            ucSubtask.RowId = subtask.SubtaskID;

            return ucSubtask;
        }
```

Figure 5.16: SubtaskUser Constructor in the Subtask controller

5.4 Alert Component

The alert layer is a crucial part of the structure of the problem, as it is scheduled under "must have" in the MoSCoW model at section 2.5. It is a key tool used for planning in the program, and achieves this by allowing users to keep track of deadlines. An Alert can be attached to a Subtask after it is created. The Alert inherits the ID of the Subtask, which serves as its identifier. Since every Subtask ID is unique, every Alert also becomes uniquely identifiable by sharing that same ID as shown on section 5.4 at line 3. The Alert functions by notifying the user responsible for the subtask that a deadline is approaching, depending on the custom set amount of time ahead of the deadline that the user would like to be notified. Once the custom amount of time is hit the alert becomes triggered, and is then displayed for the responsible user. The process of getting the alerts from the database happens in the OnInitializedAync function shown on section 5.4, where we use the the GetAlerts function that is located in the alertController, we can see that at line 3. After getting the alerts from the database we then filter the alert list so we take those where IsActive is true and HasBeenTriggered is false, as shown at line 4 to 5. Now we have an alert list where the alerts are not triggered, then we call the AlertTrigger function on the filtered alert list,as we can see at line 6.

```
protected async override Task OnInitializedAsync()
{
    DBAlertList = await alertController.GetAlerts(protectedSession);
    FilteredDBAlertList = DBAlertList.Where(x => x.IsActive == true && x.HasBeenTriggered ==
    false).ToList();
    AlertTrigger(FilteredDBAlertList);
}
```

Figure 5.17: OnInitializedAync function in alert component

The AlertTrigger function as shown Figure 5.4, it takes a parameter list of the type Alert, it then iterates between each item in list of alert list, as we can see from line 5 to 9 using foreach loop. In the foreach loop we use the DateDifference function shown on section 5.5 to compare the difference in date to the TimeSetting property that the alert has, we then assign the value to the variable DeadlineDateDifference shown at line 5 on Figure 5.4. After that we use the DeadlineDateDifference variable in the if statement to compare it with the TimSetting day for the alert, and if it's less or equal the TimeSetting, the property HasBeenTriggered is changed to true, and it changes in the database aswell using the UpdateAlert function. On Figure 5.18 we can see how the alert looks like when it's triggered.

Figure 5.18: Alerts with the names of the subtasks

```
void AlertTrigger(List<Alert> AlertList)
   {
       foreach (var item in AlertList)
       {
           int DeadlineDateDifference = DateDifference(item.TimeSetting);
           if (DeadlineDateDifference <= item.TimeSetting.Day)
           {
               item.HasBeenTriggered = true;
               UpdateAlert(item);
           }
       }
   }
```

Figure 5.19: AlertTrigger function

The UpdateAlert function as shown on Figure 5.4 takes one parameter with type alert that needs to be updated. The alert get updated using the UpdateAlert method that is located in the alertController.

```
public async Task UpdateAlert(Alert alert)
   {
       await alertController.UpdateAlert(alert, protectedSession);
   }
```

Figure 5.20: UpdateAlert function

5.4.1 Alert Class

The Alert class contains 4 properties as shown in Figure 5.21 in line 3 to 6. The Content which the alert message is going to display. The Subtask ID which it inherits from the Subtask class. TriggerStatus which is a boolean expression that by default is false and switches to true when the TimeSetting is greater than or equal to the to the Subtask deadline.

```
public class Alert
   {
       public int SubtaskID { get; set; }
       public string Content { get; set; }
       public bool HasBeenTriggered { get; set; }
       public DateTime TimeSetting { get; set; }
       public bool IsActive { get; set; }
   }
```

Figure 5.21: The Alert class and its properties

In order to alert the user for when their deadline for a Subtask is approaching, we need a function to calculate the difference between the current date, and the deadline. This is done with the function "DateDifference", as seen in Figure 5.22. This function calculates the time difference by inputting

the Alert TimeSetting parameter which is of type DateTime, and subtracts it from "Today" as seen on line 6 in Figure 5.22. It then returns the amount of days difference as an integer on line 8.

```
1  int DateDifference(DateTime TimeSetting)
2  {
3      var Today = DateTime.Now;
4      var Deadline = TimeSetting;
5
6      var DayDifference = Today - Deadline;
7
8      return DayDifference.Days;
9  }
```

Figure 5.22: DateDifference function for calculating difference in days

5.5 Search Functionality

The existing database on which we are operating holds large amounts of data and being able to find the right data quickly is of importance to Attiri. Therefore, we have implemented a dynamic search functionality for each of the tables, making it possible to quickly search through the data. This chapter explains how the search functionality was implemented.

When displaying tables containing data from their current database every object is added to a list of the specific datatype. Using a simple “foreach loop” the data is then added to HTML elements and presented on the screen. To search through this list, another filtered list is created to hold the elements which meets the search requirements. The search requirements are specified through an input field on the page, with an event being triggered every time a new character is inputted into the field. When the event is triggered, a method goes through the list adding every element matching the search requirements to the filtered list. To ensure that case sensitivity is not affecting the result every character is converted to lower case.

```
1  FilteredList = ActiveCustomerList.Where
2  (x => x.Name.ToLower().Contains(searchWord.ToLower())).ToList();
```

Figure 5.23: Snippet of how the list is filtered

5.6 Charts

As our Moscow diagram in section 2.5 details we must design the front-end in a way it is easily readable and intuitively navigable. We have decided to use graphs according to the figures shown in chapter 4. To create graphs we chose the ChartJS.Blazor API, which is a tool for building graphs, that has been adapted for use in Blazor.

5.6.1 Bar chart

The greatest challenge with rendering the bars is to retrieve the correct information and display it in the right places. To solve this we must make a list containing all the employees. Each employee must have an identifier and an array with 5 indices. An index will contain information on how many hours they will work on each specific day.

```
1  public void EstimatedHoursCalc() {
2          foreach (var person in barchart) {
3                  for (int i = 0; i < 5; i++) {
4                          person.fiveDays[i] = HoursCounter(i, person.RowID);
5                  }
6          }
7  }
8  public double HoursCounter(int dayDifference, int employee) {
9          int selectedWeek = _refresher % ISOWeek.GetWeekOfYear(DateTime.Now);
10         int day = DateTime.Now.Day + dayDifference - DateDifferenceCalc() + (selectedWeek * 7);
11         int monthVar = DateTime.Now.Month;
12         double placeholder = 0;
13         if (day > DateTime.DaysInMonth(DateTime.Now.Year, DateTime.Now.Month)) {
14                 day = day % DateTime.DaysInMonth(DateTime.Now.Year, DateTime.Now.Month);
15                 monthVar += 1;
16         }
17         List<Subtask> filteredList = subtasklist.Where(e =>
18         e.StartDate.Day == day &&
19         e.StartDate.Month == monthVar &&
20         e.ResponsibleID == employee.ToString()).ToList();
21         foreach (var estimatedHour in filteredList) {
22                 placeholder += estimatedHour.HoursEstimate;
23         }
24         return placeholder;
25 }
```

Figure 5.24: Codeblock that gathers and calculates the workhours for each employee

A foreach loop iterating through the number of people in a list can be seen on line 2 in Figure 5.24. For each person there is, a for-loop is run representing a workweek. The workweek being calculated can be seen on line 4. The double array "Fivedays" is assigned the value of the method "HoursCounter" for each index. All of the filtered data being added to the list "filteredList", can be seen on line 16 in Figure 5.24. On line 21 it go through each estimated hour for a specific person. Each hour is then added to a placeholder which is then returned.

A problem we encounter is that a month does not contain exactly 4 weeks, so a month could end abruptly on any day. For example; if we gather the days for a given month, the last bar chart of November 2021 would only contain 2 days because it ends on a Tuesday. To solve this problem we need to also filter on month and calculate if the filter should enter a new month and how many days of the new month it should filter. We also need this if the user wants to pick a new week to visualize incoming estimated work hours

A variable named "selectWeek" can be seen on on line 9 in Figure 5.24. It is assigned to the value of the chosen week modulo the current week. We do this that we know which week is picked. So in the case a week is picked that is in a new month the variable day will become larger than the days in the current month and then set the variable monthVar to be incremented by one, which is the next month. The list is then filtered with the new data and the correct information is gathered.

It is very important that the information displayed through the bar chart is correct but it is also important that the correct bar chart is displayed with the correct user. To easily work around this problem, the employees and bar charts will be rendered in the same order every time.

Another inconvenience we encounter is that every time the bar chart component has been rendered, it will not re-render even if the method "StateHasChanged" is applied. So in case new data is added to the array of working hours, it will not re-render. To solve this problem we have added

a function to reset the graph in form of a button. If the user wants to see next week's schedule, they would have to reset the graph and then pick a new week from a drop down.

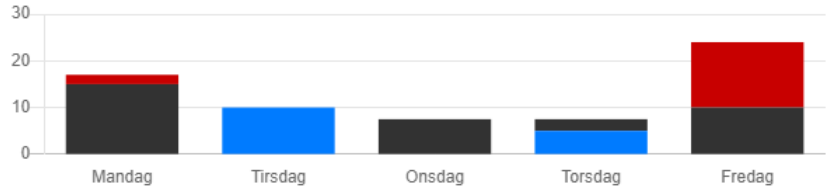


Figure 5.25: Barchart for a single employee

5.6.2 Line chart

The line chart represents how many hours the company has assigned as a whole. The line chart is formed by gathering all estimated work hours for each day for each employee. Each day's total amount of estimated working time is then divided by the amount of employees to get the average. We have done this so the user can easily see where the line drops below the vertical line and then make adjustments

```
1  public double[] MonthCalc() {
2          int daysInMonth = DateTime.DaysInMonth(DateTime.Now.Year, DateTime.Now.Month);
3          double[] placeholder = new double[daysInMonth];
4          for (int i = 0; i < daysInMonth; i++) {
5                  placeholder[i] = AverageWorkMonth(i + 1);
6          }
7          return placeholder;
8  }
9  public double AverageWorkMonth(int days) {
10         List<Subtask> newList = subtaskList.Where(e => e.StartDate.Day == days).ToList();
11         double dayHoursEstimate = 0;
12         foreach (var item in newList) {
13                 dayHoursEstimate += item.HoursEstimate;
14         }
15         return dayHoursEstimate;
16 }
```

Figure 5.26: Codeblock that gathers estimated work hours for each day in month

To be able to calculate and gather the estimated working time for each employee the code block seen in Figure 5.26 is used. The data is kept in the double array method MonthCalc. MonthCalc is used in the dataset that the configuration of the graph uses. A loop that goes through the amount of months in the current month can be seen on line 4 in Figure 5.26. In the loop the method AverageWorkMonth is called and added to an array on the current index. The method AverageWorkMonth can be seen getting all the estimated hours of a given day and then returning the value can be seen on line 10 to 14 in Figure 5.26

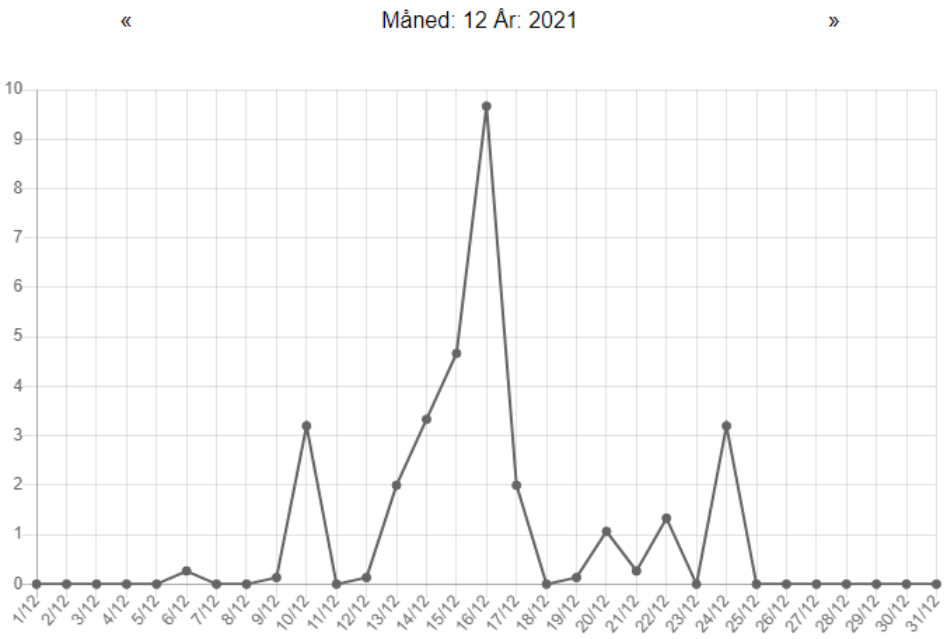


Figure 5.27: Visualization of the company workload as Line Chart

5.7 Subtask Input Component

The ability for Attiri to be able to plan out their workloads relies on seeing the list of projects and dividing them based on the estimated time to complete them. This currently has the flaw that many projects could favorably be divided into smaller subtasks constituting the project as a whole. This is where the subtask input component comes into affect.

This component will be used to divide the project into smaller parts, so that an employee may receive a reminder to e.g. request attachments from a customer containing their financial numbers. An example could be the week after they become available, e.g. 7. of January 2020, when an annual financial report for 2019 has to be finished before 2021, so that Attiri does not end up exceeding a deadline. The project as a whole may take a week but receiving the attachment may take 1-5 days and be a prerequisite to beginning the report. This is central to solving the part of the problem statement at section 2.1 stating "How can we make a tool (...) which makes it manageable to meet deadlines across multiple projects and tasks?"

```
public List<IEmployee> employees = new List<IEmployee>();

public Subtask MySubtask = new Subtask();

public List<Subtask> MySubtasks = new List<Subtask>();

public string Response { get; set; } = "";

public async void submit()
{
    await SubtaskController.InsertSubtask(MySubtask, protectedSession);
}

protected override async Task OnInitializedAsync()
```

```
15  {
16      employees = await EmployeeController.GetEmployees(protectedSession);
17      MySubtasks = await SubtaskController.GetSubtasks(protectedSession);
18  }
19
20  public bool HideLabel { get; set; } = false;
21
22  public void Toggle()
23  {
24      HideLabel = !HideLabel;
25  }
```

Figure 5.28: The C# code for the SubtaskInput component

The Figure 5.28 calls upon the Employee- and SubtaskController at lines 16-17, which are used to receive the list of employees employeed at Attiri and ability to view the existing subtasks. Once the input form is filled out and the button "Submit" is pressed the form calls the submit method at line 9-12 which invokes "InsertSubtask" which translates the class of the type Subtask into a class type compatible with Uniconta's database

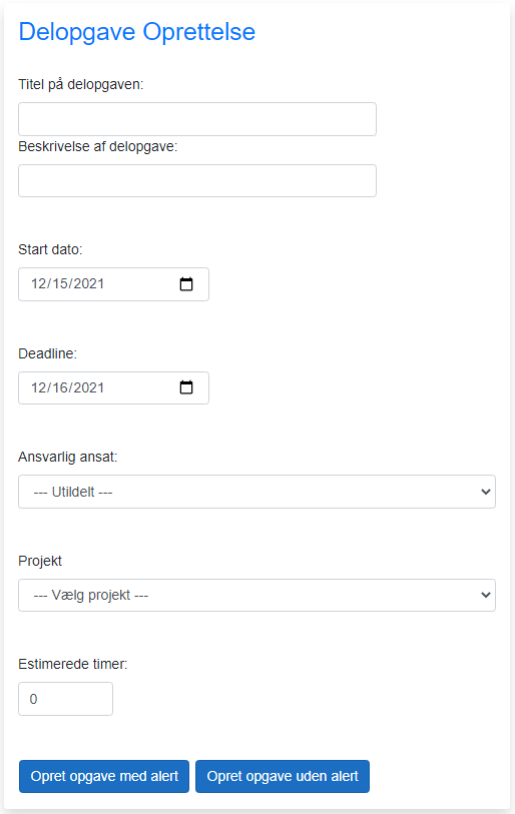


Figure 5.29: Subtask input

5.8 Redistribution of Subtasks

The ability to redistribute a subtask to another employee is one of the systems core functionalities. It is only the managers who have access to redistribute a subtask. This part of the system is split into two parts. The first part is the manager being able to see each employee's workload per week, which is visualized in the bar charts Figure 5.25. The other part is being able to redistribute these

subtasks, which makes the manager able to split out the workload between the employees more even, and at the same time plan the workload, preventing a huge workload just before a deadline.

5.8.1 Manager Overview

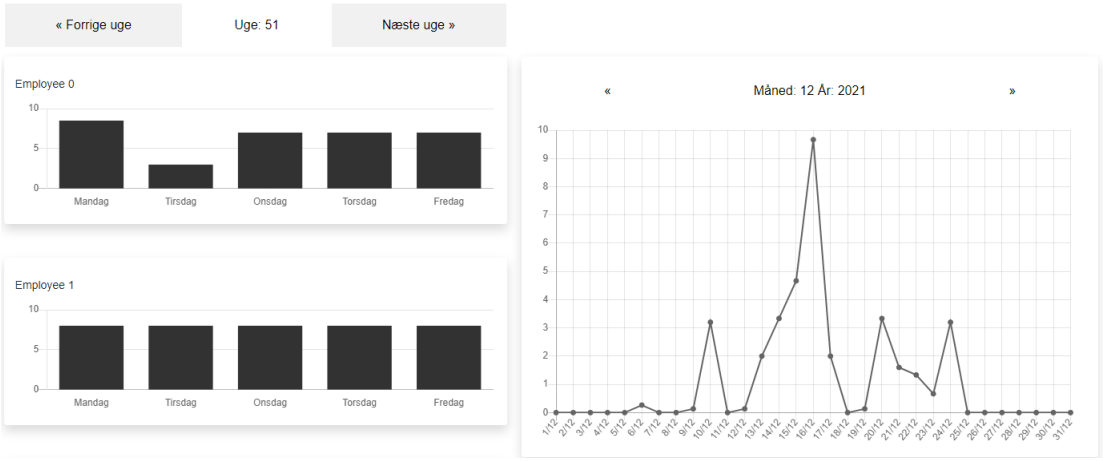


Figure 5.30: Manager overview

The visualization of the employee's workload is done using a bar chart for each employee. To provide an overview of the workload of the entire organization a line chart is used which displays the average amount of planned work hours across all employees. This gives the manager a quick overview of the average workload compared with the specific employee's workload. The logic behind the charts is identical to that of Figure 5.24 and Figure 5.26.

One of our primary focuses has been making this application as intuitive for the user as possible. Figure 5.31 is one of the more explicit examples of this. If the user wants to see more information about something, they can click on it, without there being an explicit button showing this. In this case, is it done by automatic navigating the user to the page where they can redistribute the employees' subtasks, by clicking on the chart displaying the employees' workload. Not only does it remove unnecessary interaction. This automatic redirection is happening in the div on line 8 in Figure 5.31, which has an event listener connected, that invokes the navigation manager, which has been described in previous sections.

```
1  for (int i = 0; i < employees.Count; i++)
2  {
3      IEmployee emp = employees[i];
4          <div class="shadow p-3 mb-5 bg-white rounded" style="margin-top: 10px">
5          <div>
6              <div id="empName" style="margin-top: 10px">@employees[i].Name</div>
7          </div>
8          <div id="singleBar" @onclick="() => Navigate(emp.EmployeeID.ToString())">
9              <BarChart UCData="barchart[i].fiveDays"/>
10         </div>
11     </div>
12 }
```

Figure 5.31: For loop creating each chart for the manager overview

5.8.2 Redistribution of Subtasks

The redistribution page, seen in Figure 5.32, consists of two separate tables. One for showing pending subtasks that are within their deadlines. And another table with pending subtasks which have their deadlines exceeded. This table includes some buttons and a drop-down selection. These buttons give the manager the ability for editing the subtask, reassign to another employee via the drop-down selection or delete the subtask. Once again will there via the snackbar be given a confirmation for the change in the subtask regardless of it has been deleted, edited or reassigned to another employee. Regardless of what action has been performed on the subtask, the subtaskcontroller is able to carry out the operation. The controller has methods for deleting and updating the subtask via the PUT API. Note that when we describe a deletion of some data in our system it will, as described in subsection 5.3.4, only change the property IsActive, from true to false.

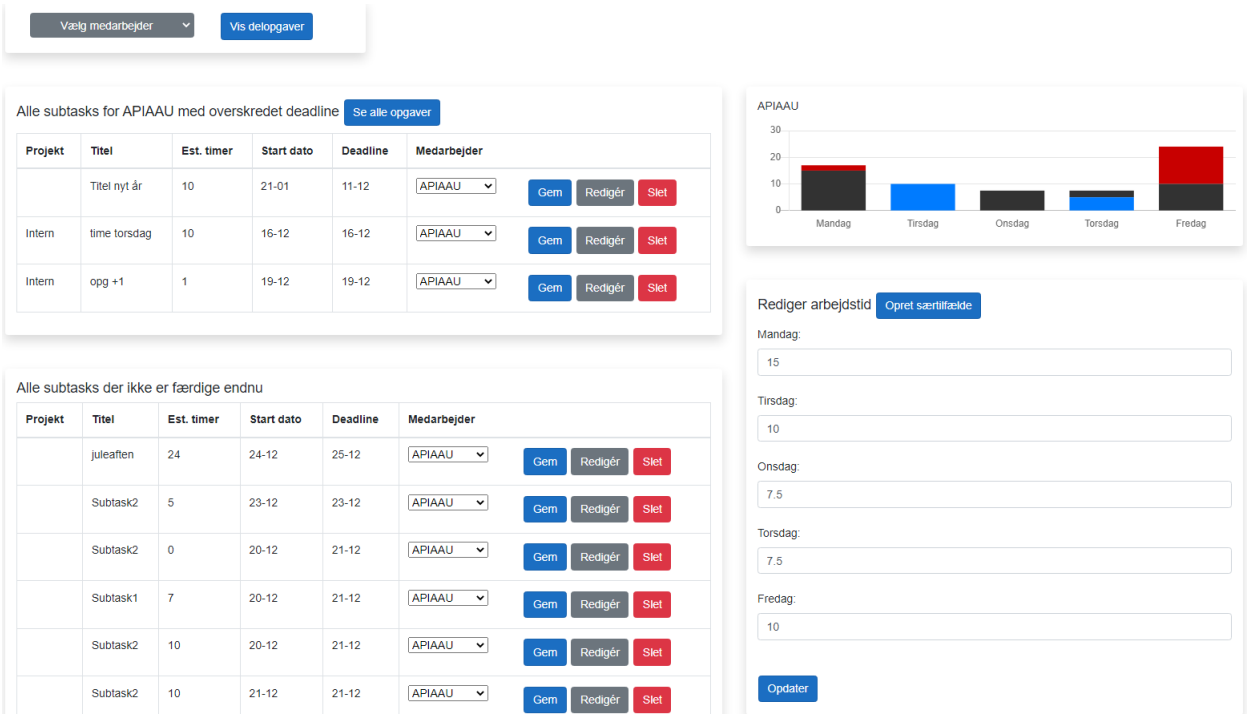


Figure 5.32: Manager redistribution

```
[Parameter]
public string? Id { get; set; }
public IEmployee choosenEmployee { get; set; }
public bool doneLoading { get; set; } = false;
public List<Subtask> allSubtasks = new List<Subtask>();
public List<Subtask> subtasksNotDone = new List<Subtask>();
public List<Subtask> subtasksNotDoneDeadline = new List<Subtask>();
public List<Project> allProjects = new List<Project>();
public List<Project> activeProjects = new List<Project>();
public List<IEmployee> employees = new List<IEmployee>();
public bool hidden { get; set; }
```

```
18
19  protected override async Task OnParametersSetAsync()
20  {
21      donaLoading = false;
22      int id = Convert.ToInt32(Id);
23      allSubtasks = await SubtaskController.GetSubtasks(protectedSession);
24      allSubtasks = allSubtasks.Where(x => x.IsActive == true).ToList();
25      employees = await EmployeeController.GetEmployees(protectedSession);
26
27      choosenEmployee = employees.FirstOrDefault(x => x.EmployeeID == Convert.ToInt32(Id));
28      employees = employees.Where(x => x.IsActive == true).ToList();
29
30      allProjects = await projectcontroller.GetProjects(protectedSession);
31      activeProjects = allProjects.Where(x => x.IsActive == true).ToList();
32
33      allSubtasks = allSubtasks.Where(x => x.ResponsibleID == choosenEmployee.EmployeeID.ToString())
34                                      .ToList();
35      subtasksNotDone = allSubtasks.Where(x => x.IsDone == false
36                                          && x.Deadline.Date >= DateTime.Now.Date).ToList();
37      subtasksNotDoneDeadline = allSubtasks.Where(x => x.IsDone == false
38                                                  && x.Deadline.Date <= DateTime.Now.Date).ToList();
39      doneLoading = true;
40  }
```

Figure 5.33: Fetching and filtering data for redistribution of subtasks

The logic for rendering these tables requires a lot of data, this can be seen on Figure 5.33. This means it is one of our pages with the highest loading time. The page is rendered with a parameter, which is set at line 2, this is set via the URL. Then a lot of different lists is initialized at line 5 to 10. When the parameters have been set, the application can begin filtering the different subtasks. It starts by fetching all the subtasks in one list at line 16. Furthermore, the subtasks which are inactive will be removed. At line 18 all the active employees will be found and rendered as the dropdown selected at the top of Figure 5.32. At last, the employee object will be found via the ID for the employee given in the url. This operation is executed in the employee controller on line 19. All the projects are then being found, for us being able to display the project name and not only the project ID. At the end the subtasks will be filtered into two lists containing the subtasks which are not marked as done with and without an exceeded deadline. These subtasks will then be displayed in the tables in Figure 5.32.

5.9 Customers

As mentioned before in our MoSCoW analysis in section 2.5 an easy overview of customers is a key functionality of the system. The easy overview is achieved by gathering all of the active customers from Uniconta and visualizing them in a structured way. We decided to go with tables according with the figures shown in chapter 4. The table contains the necessary information that the employee might need, such as a company name, contact person, phone number and email address. It also includes a search functionality as mentioned in section 5.5. This makes it easier to find a specific customer.

The process of getting the customers from the Uniconta database is done using the GetCustomers method that is located in the CustomerController file which is seen on line 7 in Figure 5.34. The result is getting the active customers by checking that their IsActive property is true as shown in Figure 5.14 on line 9.

When a customer's name is clicked in the generated table, the user will be redirected to the specific

customer's page. This page contains the important information about the customer. In Figure 5.34 from line 1 to 10 we can see the generated table for the active customers, and from line 4 to 5, it shows what happens when clicking on the CompanyName using HTML Hyperlinks that links to a customer page and passes the CustomerRowID and the CustomerID to it.

```
1   @foreach (var Client in FilteredList)
2         {
3         <tr>
4             <td class=" this"> <a class="nostyle" href="@($"customer/{Client.CustomerRowID}/
5             {Client.CustomerID}")">@Client.CompanyName</a> </td>
6             <td>@Client.ContactPersonName </td>
7             <td>@Client.PhoneNumber </td>
8             <td>@Client.EmailAddress </td>
9         </tr>
10        }
11    List<Customer> customer;
12    IEnumerable<Customer> ActiveCustomer;
13    List<Customer> ActiveCustomerList = new List<Customer>();
14    List<Customer> FilteredList = new List<Customer>();
15  protected override async Task OnInitializedAsync()
16    {
17        customer = await CustomerController.GetCustomers(protectedSession);
18        ActiveCustomer = customer.Where(x => x.IsActive);
19        ActiveCustomerList = ActiveCustomer.ToList();
20        FilteredList = ActiveCustomer.ToList();
21    }
```

Figure 5.34: The progress of getting the active customers

5.10 Blog Post

Blog post is under the "Should have" points of our MoSCoW analysis in section 2.5. The idea of the blog post is to make it easier and faster for the user to write notes and reviews about their customers. Each customer has its own page in the system and for each customer we have a blog post. Each blog post will only show the notes and reviews which are added to the specific costumer.

```
1     <BlogPost customerID="@id" ActiveprojectsList="@FilteredCustomerBlogPosts"></BlogPost>
2
3     [Parameter]
4     public string id { get; set; }
5     [Parameter]
6     public string id2 { get; set; }
7     public Customer Client = new();
8     public List<Blogpost> CustomerBlogPosts;
9     public List<Blogpost> FilteredCustomerBlogPosts;
10    public List<Project> FilteredCustomerProjects;
11    public List<Project> CustomerProjects = new();
12  protected override async Task OnInitializedAsync()
13  {
14      Client = await CustomerController.GetCustomer(id, protectedSession);
15      CustomerBlogPosts = await BlogpostController.GetBlogposts(protectedSession);
16      CustomerProjects = await ProjectController.GetProjects(protectedSession);
17      FilteredCustomerBlogPosts = CustomerBlogPosts.Where(x => x.CustomerID == id
18      && x.IsActive == true).ToList();
```

```
19      FilteredCustomerProjects = CustomerProjects.Where(x => x.CustomerID == id2).ToList();
20  }
```

Figure 5.35: Blog Post filtering implementation

As shown in Figure 5.35 at line 17, the list "FilteredCustomerBlogPosts" will contain blog posts, if its CustomerID matches the ID of the current page. The customer's ID value has been passed to the page using [Parameter] as we can see at line 3 and 4. The blog post contains three functionalities, these are Add, Remove and Edit. When the post is added, it shows who made it and when it's created. The Remove function does not delete the post from the database but instead it changes the value of the isActive property that a BlogPost has from true to false. The next time the page gets updated the post that got "removed" will not show up, as we can see the implementation in Figure 5.35 at line 17 and 18, as the list FilteredCustomerBlogPosts will only contain blog posts if the customerID match the ID that has been passed to the page and the isActive is true.

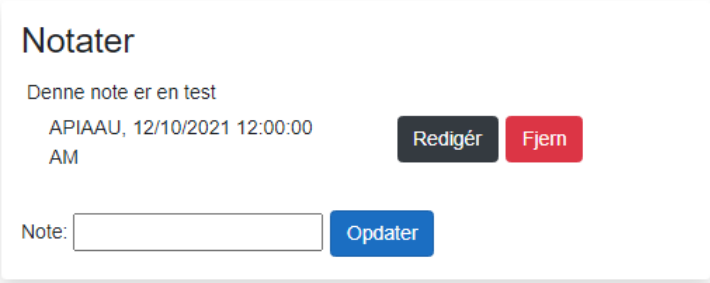


Figure 5.36: BlogPost field in customer page

5.11 Projects

An easy overview of tasks is a requirement placed under the must have section of our MoSCoW analysis at section 2.5. Attiri refers to tasks and assignments as projects.

```
1   <tbody>
2       @foreach (Project project in FilteredList)
3       {
4       <tr>
5           <td class="this"><a class="nostyle" href="@($"project/{project.ProjectRowID}")">
6           @project.ProjectID </a></td>
7           <td>@project.ProjectName</td>
8           <td>@project.TypeOfTask</td>
9           <td>@project.StartDate.ToString("yyyy-MM-dd")</td>
10          <td>@project.Deadline.ToString("yyyy-MM-dd") </td>
11          <td>@project.ResponsibleEmployee</td>
12      </tr>
13      }
14  </tbody>
15  List<Project> projects = new List<Project>();
16  IEnumerable<Project> CurrentProjects;
17  List<Project> CurrentProjectsList = new List<Project>();
18  List<Project> FilteredList = new List<Project>();
19  protected override async Task OnInitializedAsync()
20  {
21      projects = await ProjectController.GetProjects(protectedSession);
22      CurrentProjects = projects.Where(x => x.Deadline >= DateTime.Today);
23      CurrentProjectsList = CurrentProjects.ToList();
```

```
24      FilteredList = CurrentProjectsList.ToList();
25  }
```

Figure 5.37: The visual presentation of projects

We get all active projects from Uniconta by using the GetProjects method that's located in the Project Controller. We can see on line 21 in Figure 5.37 that when the Project Controller is done putting the projects into the projects list, we filter out the outdated projects. On line 22 we compare the deadline of the project with the current date so that the list CurrentProjectsList will only contain the projects which are still active. The search functionality mentioned in section 5.5 is also implemented, where the user will be able to sort by the project ID or the project name.

As shown on Figure 5.37 from line 5 to 11, we collect data relevant to the user such as the project's name, id, type of project, responsible employee, start date and deadline for the project. When clicking on the project name it should direct the user to the project page which can be seen on line 5, using HTML hyperlinks which link to a project page and pass the ProjectRowID that then will be used to get only the project information that matches with the ProjectRowID.
It is able to get only one item instead of a whole list, the method GetProject is used which takes an ID as parameter, which is the ProjectRowID that has been passed. We can see the implementation of it on Figure 5.38 at line 7.

```
1       [Parameter]
2       public string id { get; set; }
3       public Project project = new();
4
5       protected override async Task OnInitializedAsync()
6       {
7           project = await ProjectController.GetProject(id, protectedSession);
8       }
```

Figure 5.38: Initialization of projects

Chapter 6

Testing

This chapter will explain how the program utilizes testing in its many forms. Unit testing, integration testing and usability testing will in this chapter be explored and their results will be expanded upon.

6.1 Unit Test

The core principle of unit testing is to give reassurance to the developers and users that the program does what it is supposed to. Another reason to unit test is to discover hidden bugs or faulty functions in the application. In this section we describe how, why and what we unit test. The functions from the application we will unit test are the controllers, the components and API helper functions.

To unit test our application we use the frameworks bUnit, Moq and NUnit. bUnit is a powerful tool made for Blazor to give developers the option to unit test .razor and .HTML files. Moq is a framework that gives developers the possibility to mock methods, classes, interfaces and much more in C#. NUnit is one of the most common unit testing frameworks in C#. We decided to use NUnit over Xunit and MSunit because the documentation of NUnit is more detailed.

To structure our test we use the arrange-act-assert method also known as AAA. In the arrange part we set up our variables and methods. In the act part we do the necessary computing on the items in the arrange part. Lastly, in assert we write our expectation of the computed items. If our expectation is correct the test is done and successful.

To avoid edge-cases and repetition we make few unit-tests but make them cover the whole program. If the few unit tested functions did not work all other parts of the application would simply not work.

6.1.1 Controllers

The responsibility of each of our controllers is to facilitate reading and writing data of a specific type to the Uniconta database. We have a total of 7 controllers. Each controller is related to an object type from our data model and an object type from Uniconta's data model. Each object from our data model that has a related controller, contains a constructor for instantiating corresponding objects from the Uniconta data model (an example of this could be the Customer class from Figure 5.14). This constructor has the responsibility for translating the property names of an object from Uniconta's data model into properties that makes sense in the context of our data model. We are interested in testing if each of these 7 constructors can properly translate objects from Uniconta's data model into objects of our data model. We do so by setting up a test that resembles an iteraitve logic in the controllers instead of testing the actual controllers. If the test passes, the constructor works in the class and therefore we can assume that it also works when we take it in use in the controllers.

Four of the controllers also have a functionality for instantiating objects of the Uniconta data model, based on objects from our data model (marked with + under POST in Table 6.1). This is needed when we want to store objects in Uniconta's database. To map the property names from objects of our data model to property names of objects in Uniconta's data model, we use a function in each of the four controllers called ConstructDataModel(). We are interested in testing if this function can properly instantiate objects that fit into the database. In the coming subsection, an example of one of the controllers will be tested in its ability to translate objects from one data model to another.

Controller	Our model	Uniconta model	GET	POST
Subtask Controller	Subtask	SubtaskUser	+	+
Project Controller	Project	ProjectClient	+	
Employee Controller	IEmployee	EmployeeClient	+	
Customer Controller	Customer	DebtorClient	+	
Blogpost Controller	Blogpost	BlogpostUser	+	+
Alert Controller	Alert	AlertsForSubtaskUser	+	+
Reminder Controller	Reminder	ReminderUser	+	+

Table 6.1: Controllers and data models

The controllers also use an API that we have created to facilitate the actual reading and writing to the database but the unit tests for this are not a part of the controller tests. The tests of the API are documented in Figure 6.5.

6.1.2 Subtask Controller

In this unit test we want to make a replica of the subtask controller's behavior. This specific controller can get elements from Uniconta but it can also post to Uniconta by using the method ConstructDatamodel. First we arrange our test by instantiating 4 objects of type SubtaskUser, which is a type that originates from Uniconta's data model. We then set the properties of the objects with random data and add the objects to a list. The arrange phase is now done.

```
1  foreach (SubtaskUser ucSubtask in subtaskUsers) {
2          subtasks.Add(new Subtask(ucSubtask));
3  }
4  foreach (Subtask subtask in subtasks) {
5          subtaskUsers2.Add(ConstructDatamodel(subtask));
6  }
7  Assert.IsInstanceOf(typeof(SubtaskUser), subtaskUsers[0]);
8  Assert.IsInstanceOf(typeof(Subtask), subtasks[0]);
9  Assert.AreEqual(4, subtasks.Count);
10 Assert.AreEqual(4, subtaskUsers2.Count);
11 Assert.IsInstanceOf(typeof(SubtaskUser), subtaskUsers2[0]);
```

Figure 6.1: Act and assert part of the subtask controller unit test

In Figure 6.1 line 1 we begin the act phase of our unit test. On line 1 to 3 we mimic the controller's behavior by adding the content of the list SubtaskUsers to a new list called subtasks of type Subtask. On line 4 to 6 we add the content of the list subtaks to a new list called subtaskUsers2 of type SubtaskUser by using the method ContructDatamodel. The act part is now done.

We can now begin to create our expectations of this controller. On line 7 we expect the first element in the list subtaskUser to of type SubtaskUser. On line 8 we expect the first element in the list subtask to be of type Subtask. On line 9 to 10 we expect the lists to be same length as the amount of objects we added to them. On line 11 we expect that the first element of subtaskUsers2 is of type Subtaskuser.

We have now tested the Subtask controller and proven that objects of type SubtaskUser can be used to instantiate objects of type Subtask. We have also proven that we can instantiate objects of type Subtask and use the controller to instantiate objects of type SubtaskUser.

Each controller is tested in the same manner. Some controllers can GET and POST, some can only GET (see Table 6.1). In case a controller only can GET, the method ConstructDatamodel is

excluded and therefore not tested. After running all the unit tests of the controllers we get: *all has passed with zero failures.*. Thus we have proven that the controllers work.

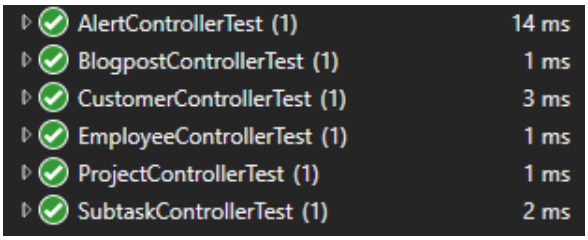


Figure 6.2: Screenshot of successful run of all controller unit test

6.1.3 Components

The controllers have been tested and we have found our implementation secure, therefore we can test our components, since they use the controllers to read and write date from Uniconta. In these unit tests we want to replicate the logic that is expressed in the different components.

Employee workload (chart):

```
public List<BarchartValueModule> barchart = new List<BarchartValueModule>();
List<IEmployee> employees = new List<IEmployee>();
List<Subtask> subtasklist = new List<Subtask>();
//arrange
    //instantiating subtasks here
    //Instantiating employees here
    //Adding the subtasks to a list here
 //act
 GeneratePeople();
 EstimatedHoursCalc();
 //assert
 var dateOfWeek = DateTime.Now.DayOfWeek;
 switch (dateOfWeek) {
   case DayOfWeek.Monday:
     Assert.AreEqual(barchart[0].fiveDays[0], 16);
     break;
   case DayOfWeek.Tuesday:
     Assert.AreEqual(barchart[0].fiveDays[1], 16);
     break;
   case DayOfWeek.Wednesday:
     Assert.AreEqual(barchart[0].fiveDays[2], 16);
     break;
   case DayOfWeek.Thursday:
     Assert.AreEqual(barchart[0].fiveDays[3], 16);
     break;
   case DayOfWeek.Friday:
     Assert.AreEqual(barchart[0].fiveDays[4], 16);
     break;
   default:
     Assert.AreEqual(barchart[0].fiveDays[0], 16);
     break;
 };
```

Figure 6.3: Test: Employee workload (chart)

Figure 6.3 is a unit test of the BarchartLoader component. The component is responsible for creating correct arrays for the barchart compoenent to be displayed. On line 5, we instantiate our subtask, the code have been condensed down to a comment so has the instantiating of employees and adding the subtasks to list on line 6 to 7.

In our act phase we then run our method which reads the list with the added subtasks. The method GeneratePeople on line 9 adds all the employees to the variable barchart. The method EstimatedHoursCalc sets the variable barchart's property FiveDays to the calculated array.

In our assert phase we have a switch statement. The reason we made a switch statement is because we want our test to be able to run succesfully on any given date. Because some of properties use DateTime.Now which returns any current date we need to be able to decide which day the test is run. For example, if this test is run on a Monday, the switch statement would run the following code on line 15. Line 15 asserts that the amount of work the first employee has to do on a Monday is 16 hours, which is also the amount of hours we added to the list, therefore the test is correct and runs without failures.

Unassigned Subtasks component:

```
1  public List<Subtask> AllSubtasks { get; set; } = new List<Subtask>();
2  public List<Subtask> UnassignedSubtasks { get; set; } = new List<Subtask>();
3  private List<Subtask> GetUnassignedSubtasks() {
4          UnassignedSubtasks = AllSubtasks.Where(x => x.IsActive == true).ToList();
5          UnassignedSubtasks = UnassignedSubtasks.Where(x => x.IsActive == true
6          && x.ResponsibleID == "" || x.ResponsibleID == null).ToList();
7          return UnassignedSubtasks;
8  }
9  //arrange
10  //Instantiating subtasks
11  //Adding subtasks the list AllSubtasks
12 //act
13 UnassignedSubtasks = GetUnassignedSubtasks();
14 //assert
15 Assert.AreEqual(1, UnassignedSubtasks.Count);
```

Figure 6.4: Test: Unassigned Subtasks component

Figure 6.4 is a unit test of the component UnassignedSubtask. The component is responsible for showcasing subtasks that are not assigned to anyone. In our arrange phase, the instantiating and adding of objects have been changed to a comment to avoid repetition.

On line 13 in our act phase we assign our list UnassignedSubtasks to be the value of the method GetUnassignedSubtasks from line 3. The method returns a list of subtasks that complies to specifiers on line 4 to 6.

In our assert phase we assert that the amount of subtasks in the list UnassignedSubtasks equals to 1. It equals to 1 because of all the subtasks added to the list AllSubtask, only one subtask complies to the specifiers. Therefore the test is correct and runs without any failures.

6.1.4 API helpers

In this unit test we want to test the methods that help tie the API together. We do this because if an error regarding the API occurs, we would know that the error is generated by the helper methods. The helper methods are:

- GetJsonSettings - Responsible for setting the settings for Json objects.
- ResponseToUCDatamodel - Responsible for giving a response to the Datamodel
- GetDataType - Responsible for getting the type in Odata
- AuthorizationHeader - Responsible for returning a string inorder to log in.

```
1  //arrange
2  string subtaskUserType = typeof(SubtaskUser).ToString();
3  string json = "[{SubtaskDesc: \"1\", RowId: 1}, {SubtaskDesc: \"2\", RowId: 2}]";
4  string authorization = "Basic " + Convert.ToBase64String(Encoding.UTF8
5  .GetBytes(String.Format("{0}:{1}", "a", "a")));
6  string authorizationHeader = await AuthorizationHeader("a", "a", ProtectedStorage);
7  //act
8  List<SubtaskUser> subtaskUsers = ResponseToUCDatamodel<SubtaskUser>(json);
9  //assert
10 Assert.AreEqual(2, subtaskUsers.Count);
11 Assert.IsInstanceOf(typeof(SubtaskUser), subtaskUsers[0]);
12 Assert.AreEqual(1, subtaskUsers[0].RowId);
13 Assert.AreEqual("SubtaskUser", GetDataType(subtaskUserType));
14 Assert.AreEqual(typeof(SubtaskUser).ToString(), "Attiri.MetaModels.SubtaskUser");
15 Assert.AreEqual(authorization, authorizationHeader);
```

Figure 6.5: Test: API helper functions

In the arrange phase in Figure 6.5 line 2 to 6, we create the necessary variables to test all 4 methods. In the act phase line 8, we assign the list subtaskUsers to be the return value of ResponseToUC-Datamodel with the string json from line 3 as a parameter.

In the assert phase on line 10 we create our first assertion. We assert that the amount of objects in the list subtaskUsers is 2. This is correct because the string json included 2 subtasks in JSON-format. On line 11 assert that the type of the object is the type SubtaskUser, this is also correct. On line 12 we assert that the first object in the list has a row-ID which equals to 1. This is true because we added 1 to the first object's row-ID. On line 13 we assert that the type from Odata is the same as SubtaskUser which it is and therefore correct. On line 14 we assert that the type of SubtaskUser rewritten as a string is the same as Attiri.Metamodels.SubtaskUser, which is true. On line 15 we assert that the method authorizationHeader with string a, as a parameter return a login instance which equals to the string authorization on line 4. Since the method returns a string the assertion is correct.

Since all 6 assertions passed the test we can conclude that the test is a success and therefore the API helper methods work as intended

6.2 Integration Testing

The reason for integration testing is that we want to see how well certain components work in conjunction with one another.

This section covers integration tests of our controllers, the API, the class constructors and the logic used in components related to the controllers. The section will be divided into three subsections; *Subtask integration*, *Employee integration* and *Customer integration*. These sections will cover testing of the component logic for our core components. This will be done by using the controllers and the API in conjunction as it is done in our implementation.

6.2.1 Subtask Integration

In this test we want to test the functionality of the subtasks in our application as a whole. This functionality can be accessed in multiple components throughout the application and is used by a manager to assign subtasks to himself or other employees. During the integration test we implicitly test the *Subtask Controller* in conjunction with the API in its ability to POST, PUT and GET subtasks from our database. We also test the logic used to filter out subtasks which have been marked as done, this functionality is for example used on our dashboard to display and color code subtasks that are done.

```
//arrange
List<Subtask> subtasks = new List<Subtask>();
List<Subtask> subtasks2 = new List<Subtask>();
List<Subtask> subtasks3 = new List<Subtask>();
SubtaskController subtaskController = new SubtaskController();
Subtask obj1 = new Subtask() {
    Description = "test 1",
    HoursEstimate = 2,
    StartDate = DateTime.Now,
    Deadline = DateTime.Now.AddDays(2),
    ResponsibleID = "21",
    IsDone = false,
    IsActive = true
};
Subtask obj2 = new Subtask() {
    Description = "test 1",
    HoursEstimate = 2,
    StartDate = DateTime.Now,
    Deadline = DateTime.Now.AddDays(2),
    ResponsibleID = "21",
    IsDone = false,
    IsActive = true
};

//act
subtasks = await subtaskController.GetSubtasks();
await subtaskController.InsertSubtask(obj1);
await subtaskController.InsertSubtask(obj2);
subtasks2 = await subtaskController.GetSubtasks();

Subtask obj3 = new Subtask();
Subtask obj4 = new Subtask();
obj3 = subtasks2[subtasks2.Count - 2];
obj4 = subtasks2[subtasks2.Count - 1];
obj3.StartDate = DateTime.Now.Date;
obj4.StartDate = DateTime.Now.Date;
obj4.IsDone = true;

await subtaskController.UpdateSubtask(obj3);
await subtaskController.UpdateSubtask(obj4);

foreach (Subtask subtask in subtasks2.Where(x => x.StartDate.AddHours(1).Date == DateTime.Now.Date)) {
    if (subtask.IsDone == true) {
        subtasks3.Add(subtask);
    }
}
```

```
47
48  //assert
49  Assert.AreEqual(subtasks2.Count - 2, subtasks.Count);
50  Assert.AreEqual(subtasks3[subtasks3.Count - 1].SubtaskID, obj4.SubtaskID);
```

Figure 6.6: Subtask integration test

In Figure 6.6 the integration test for subtasks can be seen. We start off by arranging the context of the test. We need six objects, three lists of subtasks, the Subtask Controller and 2 subtasks. These are all instantiated on lines 2 - 23. Pay attention to the property "IsDone" on line 12 and 21, we want to see if we can use the PUT capabilities of the Subtask Controller to change one of them so that we can test if the logic used in the subtask components is able to filter out finished subtasks. This will be tested at the end of this test. Firstly, we are interested in seeing if the GET capabilities of the Subtask Controller are implemented correctly through the API. This can be seen on line 26 where we use the controller to get all the subtasks from the database. Then, on line 27 and 28, we test if the controller's PUT capabilities are also implemented correctly. We test this by inserting the two newly created subtasks and then getting all the subtasks from the database and adding them to a list named "subtasks2" (line 29). We assert that "subtask2" has exactly two more elements than the list "subtasks" on line 49.

We are also interested in updating some of the properties in the two lastly added subtasks, so we instantiate two new empty subtasks on line 33 and 34 and set them equal to the two lastly added subtasks. On lines 35 - 40 we update the properties and use the PUT capabilities of the Subtask Controller to update the corresponding subtasks in the database. Note that on line 37 we have set the property "IsDone" to "true", this is interesting because we can now check if the logic from the components related to subtasks are actually able to get the finished subtasks. The logic is displayed on lines 42 - 46. We iterate over all the subtasks in our database, which currently is stored in the list "subtasks2" and add the subtasks that are marked as finished to the list "subtasks3". As the subtask "obj4" had its property "IsDone" marked as "true", we assert that this is in fact the same subtask as the lastly added subtask in the filtered list "subtasks3". This assertion can be seen on line 50.

6.2.2 Employee Integration

In the employee integration test we want to see how well the *Subtask Controller*, the *Employee Controller* and the API works in conjunction with one another. We do this because there are several use cases where employees interact with subtasks throughout the application. We demonstrate this in a specific use case: Get all the subtasks for a specific employee. This functionality is used on a specific employee's personal page but variations of the logic are also used elsewhere in the application. The reasoning behind this integration test is that the relation between an employee and a subtask appears in many forms throughout the application and we therefore need to pick a significant use case to assure us of how well the different components work together.

```
1   //arrange
2   List<Subtask> subtasks = new List<Subtask>();
3   List<Subtask> subtasks2 = new List<Subtask>();
4   List<Subtask> employeeSubtasks = new List<Subtask>();
5
6   SubtaskController subtaskController = new SubtaskController();
7   Subtask obj1 = new Subtask() {
8       ...
9   };
10  Subtask obj2 = new Subtask() {
```

```
11      ...
12  };
13
14  //act
15  subtasks = await subtaskController.GetSubtasks();
16  await subtaskController.InsertSubtask(obj1);
17  await subtaskController.InsertSubtask(obj2);
18  subtasks2 = await subtaskController.GetSubtasks();
19
20  Subtask obj3 = new Subtask();
21  Subtask obj4 = new Subtask();
22  obj3 = subtasks2[subtasks2.Count - 2];
23  obj4 = subtasks2[subtasks2.Count - 1];
24  obj3.Deadline = DateTime.Now.Date;
25  obj4.Deadline = DateTime.Now.Date;
26  obj3.ResponsibleID = "21";
27  obj4.ResponsibleID = "21";
28
29  await subtaskController.UpdateSubtask(obj3);
30  await subtaskController.UpdateSubtask(obj4);
31
32  employeeSubtasks = GetEmployeeSubtasks(21);
33
34  //assert
35  Assert.AreEqual(employeeSubtasks[employeeSubtasks.Count - 1].SubtaskID, obj4.SubtaskID);
```

Figure 6.7: Employee integration test

We start off the integration test for Employee by arranging the context of our test. The test can be seen in Figure 6.7. In this case we need three lists of subtasks, which we declare on lines 2 - 4. On line 6 we instantiate a Subtask Controller, which uses the API to POST, PUT and GET subtasks from the database. On lines 7 - 12 we instantiate 2 subtasks and set their properties appropriately (excluded in the code snippet for readability). We would like to see if we can successfully insert these two subtasks into the database. We also want to see if we can GET these two specific subtasks and update their properties accordingly in the database. On line 15 we use the Subtask Controller to GET all the subtasks in the database, we add these to the list "subtasks". On line 16 - 17, we use the Subtask Controller to insert our two newly created subtasks "obj1" and "obj2". After that, on line 18 we GET all the subtasks from the database and insert them into a list named "subtasks2" to see if the list holds exactly 2 more elements than the list "subtasks".

We also want to test the Subtask Controller's ability to update data in the database. To do this instantiate two new subtasks and assign them to the value of the two lastly added subtasks in "subtasks2". This can be seen on lines 20 - 23. Then, on lines 24 - 27 we update the properties of our two subtasks. They are updated in the database using the PUT functionality in the Subtask Controller on lines 29 - 30. To check if it is true that we have successfully assigned two subtasks to the employee with ID "21", we use logic from the employee's personal page. This logic is the function GetEmployeeSubtasks() which can be seen in Figure 6.8. On line 35 we check if its true that the lastly added active subtask for employee #21 is in fact the same subtask as we updated on line 27.

```
1  List<Subtask> GetEmployeeSubtasks(int loggedInEmployeeId) {
2      IEnumerable<Subtask> specificEmployeeSubtasks =
3        from subtask in allSubtasks
4        where subtask.ResponsibleID == loggedInEmployeeId.ToString()
5        select subtask;
6
```

```
7       return specificEmployeeSubtasks.ToList<Subtask>();
8   }
```

Figure 6.8: Function for getting an employee's subtasks

6.2.3 Customer Integration

On a customer's page, we provide functionality for reviewing and evaluating the customer. We chose to name this functionality "blog". In the customer integration test we want to see how well the *Customer Controller* works in conjunction with the *Blogpost Controller*.

```
1   //arrange
2   Blogpost blogpost = new Blogpost() {
3       Author = "test",
4       CustomerID = "100030",
5       Body = "indhold",
6       TimeStamp = DateTime.Now.Date,
7       IsActive = true
8   };
9
10  CustomerController customerController = new CustomerController();
11  BlogpostController blogpostController = new BlogpostController();
12  Customer customer = await customerController.GetCustomer(1);
13
14  List<Blogpost> customerblogpost2 = new List<Blogpost>();
15  List<Blogpost> filteredcustomerblogpost2 = new List<Blogpost>();
16
17  //act
18  await blogpostController.InsertBlogpost(blogpost);
19  CustomerBlogPosts = await blogpostController.GetBlogposts();
20  FilteredCustomerBlogPosts = CustomerBlogPosts
21                                  .Where(x => x.CustomerID == "1"
22                                  && x.IsActive == true).ToList();
23
24  FilteredCustomerBlogPosts[FilteredCustomerBlogPosts.Count - 1].Body = "Indhold opdateret";
25  await blogpostController
26      .UpdateBlogpost(FilteredCustomerBlogPosts[FilteredCustomerBlogPosts.Count - 1]);
27  customerblogpost2 = await blogpostController.GetBlogposts();
28  filteredcustomerblogpost2 = customerblogpost2
29      .Where(x => x.CustomerID == "1" && x.IsActive == true).ToList();
30
31  //assert
32  Assert.True(FilteredCustomerBlogPosts.Count > 0);
33  Assert.AreEqual(filteredcustomerblogpost2[filteredcustomerblogpost2.Count - 1]
34      .Body, "Indhold opdateret");
```

Figure 6.9: Customer integration test

The test for Customer Integration can be seen in Figure 6.9. The overall purpose of this test is to make sure that we can successfully use the Blogpost Controller to access the API functionalities for POST, PUT and GET blogposts. We also want to prove that the Customer Controller is able to GET customers in the same manner. This is tested in conjunction with logic that mimics the behavior of our customer Components. On lines 2 - 15 We arrange the test context by creating a blogpost, a Customer Controller, a Blogpost Controller, a specific customer and lists to hold all blogposts and the list of blogposts for the specific customer. The first thing we want to do is to add our newly created blogpost to the database. This is done using the Blogpost Controller on line 18. Then, on line

19 we want to get all blogposts from the database and add them to the list "CustomerBlogPosts".

Now we want to use the logic from the customer component to filter out the blogposts that belong to our specific customer, this can be seen on lines 20 - 22. We want to test if we can use the Blogpost Controller's PUT capabilities to update the body of the lastly added blogpost, we do this on lines 24 - 26. Lastly, on lines 27 - 29 we GET all the blogposts and filter out the blogposts belonging to the specific customer. On lines 33 - 34 we check that the lastly added blogpost for a customer now has a body saying "Indhold opdateret".

6.3 Usability Test

Our desire to test the product with a usability test stems from the nature of iterative design [6]. Being able to progressively refine a product, with pointers as to what might be missing, what has missed the mark or potentially what implementation has exceeded the expectation of the client [3].

The ideal usage of this method is applied to large scale project implementations spanning over several months or years. The method is supposed to counteract the regrettable pattern of miscommunication between developers and clients with regards to what the desired product in fact entails. Under normal conditions a developer team would run the prototype of the product through a handful of the intended future users of the product and note down their interactions with the product under observable conditions.

The outcome of a usability test manifests itself in the form of a ranked list of usability problems and knowledge regarding what works well and what works to lesser degrees. This ranked list of usability problems may prioritize a problem of moderate significance occurring several times across all test respondents, above a problem of a more significant nature occurring once with an unknown path for reproducibility, as this is of lesser relevance due to not having a clear path to handle it.

6.3.1 Exploratory testing

We conducted exploratory testing when talking to Attiri, in terms of having them explain their current system and which needs the new system should satisfy. Attiri had many wishes as to what a new system should and said wishes have been laid out in section 2.4. After this meeting we wrote down our understanding of their desires and had another meeting a couple of weeks later to see if any incongruencies had occurred in our understanding of each other.

6.3.2 Validation verification testing

Normally this step would be gone through iteratively, preferably with new participants through each iteration, after other types of tests have been gone through, but due to the format of the semester project there will only be one iteration of verification testing.

The objective of a validation verification is to determine if the usability objectives are fulfilled. These criteria are developed in collaboration between the developer team and the client, as a collaborative effort since the developer team know what's realistic, but the client knows what they want. Our usability objectives are listed under section 2.5.

6.3.3 Choices regarding our verification test

This chapter largely follows the guidelines laid out in [3] .

Immediately upon deciding to perform a usability test some questions will have to be answered such as when, where, who and why.

When

When is meant as a number of iterations. As previously mentioned, only one iteration will be performed due to time constraints and the size of the project. Had the project spanned over e.g. 8 months with a prototype ready at the second month, 4-6 usability tests ranging from exploratory, assessment, validation and follow-up tests would have been performed. In our case, the follow up tests would preferably be performed a few weeks after the release date as an assurance of the product's longevity.

Where

The choice is between a field test or a lab test as they are called in [11] and [12]. A Field test has the advantage of participants using the product in a more genuine manner as they do not feel as if they are "lab rats" contrary to a lab environment in which they feel observed. As recommended by our lecturer Timothy Robert Merritt we will be using our group room as the test environment. The room will be neat and tidy with some refreshment and we will not overcrowd it.

Who

Due to the busy nature of a small accounting firm, we will only be allowed to have one correspondents partake in the usability test. Whom will partake simultaneously in a joint effort. He is a manager with planning responsibilities, which makes him a disproportionate user of the product, as he needs an overview to a greater extend than the average accountant at the firm.

6.3.4 Guidelines for conducting our verification test

Three main guidelines decide how the tasks will be defined:

1. The tasks should be realistic
2. The tasks should be actionable
3. Clues and descriptions of the tasks should be avoided

The three main rules for deciding the tasks can be expanded with these guidelines:

1. Benchmarks/standards are established for evaluating success.
2. An instructor and observer are selected and given according responsibilities
   (a) The instructor is to provide tasks for the subject to perform and have minimal interference. The exception is if the users become unable to solve the task.
   (b) The task of the observer is to note down how the user interacts with the product, noting issues arising and determining their level of interferences with completion of the given task.
3. Important characteristics for the roles.
   (a) Important characteristics from the moderator.
      i. Supportive rather than controlling.

ii. Can handle uncertainty rather than sticking too closely to the test plan. [1].

iii. Establishes good relations rather than appearing better knowing to the test subject.

(b) Important characteristics from the observer.

i. An observer should prioritize making useful notes as to what occurs, rather than attempting to complete the log file during the test.

ii. Stay quiet, you are not here to impact the test, but rather document what occurs.

iii. Write down questions you would like to ask the subject during debriefing.

The subject will be instructed to think-aloud when solving tasks [8]. The way by which we will make sure the subject understands the think-aloud protocol, will be by instructing the subject to count the windows in the room, before a task is assigned. The subject will be told that we as observers do not really care how many windows there are, but rather how the subject comes to their conclusion. If the subject sits quietly and then state their answers they will be told that we as observers have learned nothing as to how they came to their answers [7]. The screen used by the subject should also be recorded as to better be able to finish the log file after the test.

6.3.5 Task list

The tasks have been selected based upon the frequency by which they appear during a regular workday, their importance and the pre-testing presumed probability of causing problems or triggering bugs.

1. Login to the platform
2. Create a subtask assigned to an employee
3. Create a subtask with some incorrect information
4. Correct the subtask to include the correct information
5. Create a reminder to all employees that there is a bowling event next Friday
6. Move the workload from a stressed period to a less stressed period
7. Add a note to a client regarding something you know about them

Description of successful completions of tasks

task number	Success action	Time limit
task no. 1	A maximum of three attempts	5 seconds
task no. 2	All inserted information is correct	30 seconds
task no. 3	The incorrect subtask is created	25 seconds
task no. 4	The incorrect subtask is corrected	35 seconds
task no. 5	The reminder is a public one	15 seconds
task no. 6	The work distribution becomes more even	35 seconds
task no. 7	The note is saved	30 seconds

Table 6.2: Successful task completion descriptors

[1] Think of this as the moderator having read the script, but not following it to a tee

6.3.6 Log File

During the attempted solution of the tasks a logger will attempt to transcribe the issues that the subject encounters, and if unable (which is likely) to write down all difficulties. The log will be completed by watching the video recordings. The log file will be in the format of a table containing timestamps for issues encountered, a transcription of the event, and a short description of the problem. The full log file can be found in the appendix at Table 9.1
After the test, the log file will be made into a problem list where all problems are listed, and the severity of the problems are noted. According to Table 6.3 the issues found in the log file will be sorted into cosmetic, serious and critical problems.

	Delay	Irritation	Expectation vs. actual
Cosmetic	< 1 minute	Low	Small diff.
Serious	Several minutes	Medium	Significant diff.
Critical	Total (user stops)	Strong	Critical diff.

Table 6.3: Problem categorization table

Small, significant and critical difference may seem vague and hard to quantify, but it it simply too hard to make estimates for how much time something should take at the first attempt as a creator of the system. As a result thereof the video will simply be reviewed and estimates of how delayed the subject is will be made.

6.3.7 System usability scale

As the subject has completed the verification test, a System Usability Scale (SUS) test [13] will be handed to the subject. The subject will be asked to fill out a 10 item questionnaire with the answer options ranging from strongly disagree to strongly agree (1-5). A formula for calculating a score will be used, the technical details of the formula will not be explained, but note that a score of 68 is considered average, and thereby anything below 68 is considered below average and vice versa.

6.3.8 Requirements Before The Test

Here the list of items needed before the test will be listed.

- A computer with a screen recording software (such as OBS).
- A consent form.
- A piece of paper with the SUS test ready to be filled out.
- An observer with a text editor for note taking.
- Preferably a neat and tidy room with a few refreshments.

Attached in the appendix there is a script for introducing the test to the subject, the log file, a consent form and the SUS questionnaire.

6.3.9 Summary

The subject was placed in the "lab" (Our group room), signed the consent form, was introduced to the think aloud protocol, after which he was be asked to solve the tasks from the task list found at subsection 6.3.5. During this the observer/logger attempted to write down which issues occurred. The moderator sat idle and only assisted the subject if he was truly stuck. Following this the debriefing took place, here the subject be given the SUS questionnaire and asked about details as to what happened in the test. The questions were asked to clarify as to what went wrong at times, and instances which weren't fully understood by neither the moderator nor the observer/logger.

6.3.10 Result expectations

As can be seen below in subsection 6.3.11 a baseline time for completion of tasks was set, based upon how quickly someone experienced with the system takes to complete the tasks. The baseline is set by a member of the group completing the tasks. It is expected that the test subject will take about 2-3x the amount of time to complete the tasks. This has been decided based upon expectations regarding the test subject's technical skills.

6.3.11 Results of Verification test

Here the test results will be listed such as completion time compared to a baseline for being fast. The issues described here are ranked as cosmetic, serious or critical in accordance to Table 6.3. All problems will be described in the log file and be categorized in the Table 9.2. The subject is expected to complete the tasks considerably slower than the baseline, but a reasonable estimate would be within the margin of 2x-3x the time of the baseline.

Task	Baseline	Subject
Login to the platform	0:05	N/A
Create a subtask assigned to an employee	0:30	2:30/1:25
Create a subtask with incorrect information	0:25	0:55
Correct the incorrect information	0:30	0:30
Create a reminder to all employees	0:20	0:45
Move the workload from a stressed period to a less stressed period	0:35	2:30
Add a note to a customer regarding something you know about them	0:30	0:40

Table 6.4: The times for completions of tasks, see Table 9.1 for more information

The SUS questionnaire totals to 82.5 points which is excellent and showcases that the system is highly usable.

Feedback From Verification Test

The feedback from the subject in the debriefing resulted in the following feedback:

- Projects should be sortable by responsible employee
- Potentially all employees with more than one year of experience should be given admin access as the firm has a lot of faith in its employees.
- The trashcan should be sortable by date of "deletion" (see subsection 5.3.4 for information on how deletion works in the system)
- Alerts appear as required to be set when a subtask is created, this is not reality though.
- The option for editing how many hours each employee has scheduled

All of the aforementioned features have been implemented after the usability test and is described in subsection 6.3.12.

Other than these concrete feature requests the subject was overall satisfied with the system, and even acknowledged that the tasks which took him a long time were because it was his first time using the system. For further info regarding the subjects satisfaction with the system view Table 9.3.

6.3.12 Added functionality from usability test

Most of the functionality that the test subject asked for were minor changes except for the ability to edit how many hours each employee has scheduled. The test subject wanted this change since some of his employees' weekly schedule are different from the common 7.5 hours of daily work. This component is only accessible by managers because we do not want employees to change their schedule without internal communication.

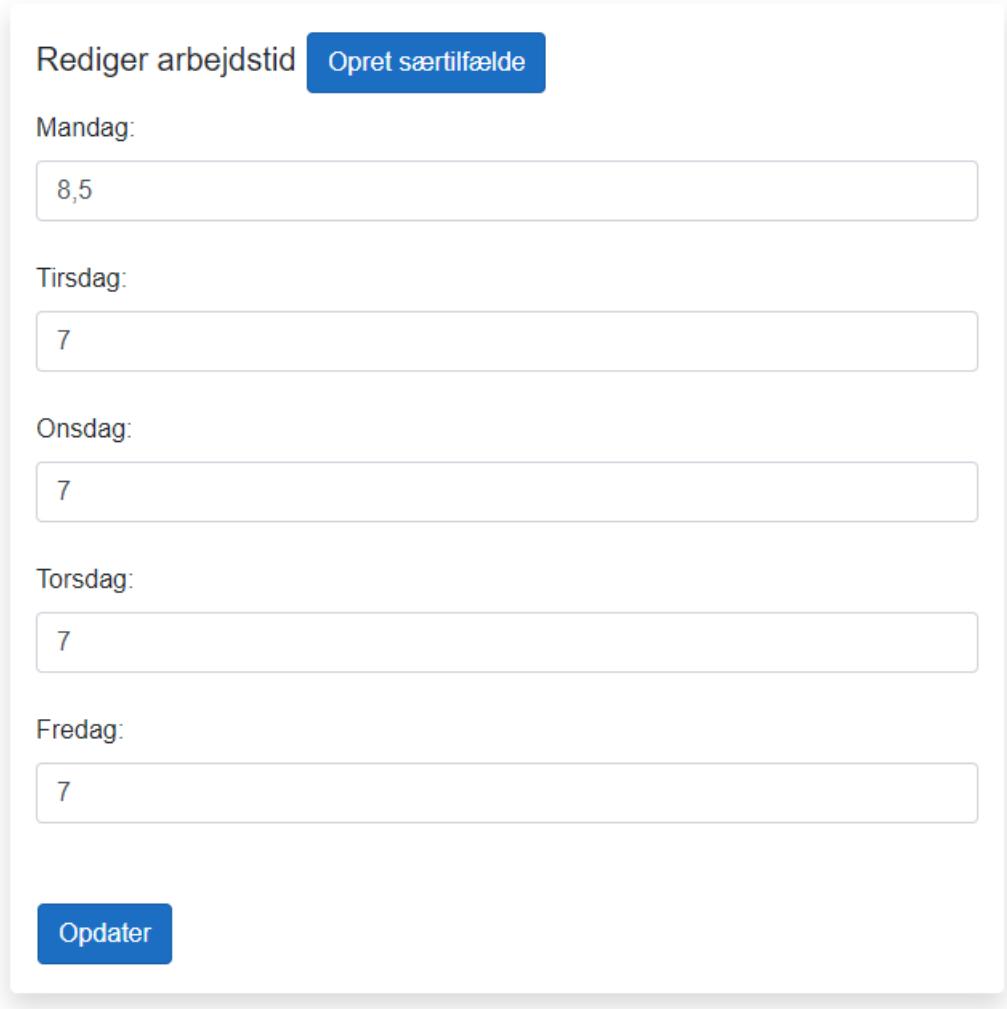


Figure 6.10: Snippet of the wanted component from the usability test

This update changes how the bar charts are displayed. On Figure 6.10, if 3 is inserted in the first row and the button "Opdater" is clicked, this would result in the bar chart displaying everything above 3 hours on a Monday, for that specific employee, as overwork. If the button "Opret særtilfælde" button is clicked, the manager will be redirected to a page where they can create special events that changes an employee's schedule. For example if an employee is fallen sick or is on holiday, the manager can create a period where the employee's daily work schedule is 0 hours. This change is great because it gives the managers of Attiri more flexibility and it makes the application reflect the world better.

Chapter 7

Discussion

This chapter discusses the level to which the requirements laid out in section 2.5 are fulfilled and if the question stated in the problem statement has been answered. In this chapter, we also discuss our approach regarding time planning, database reliability, communication with Attiri and tests.

7.1 Blazor

When choosing the framework for this project we had many considerations regarding the learning curve and the requirements to the application. After meeting with Attiri for the first time we realized that they wanted something that should be accessible from anywhere by all employees. We also found that our project would quickly grow to a considerable size, meaning that we would have to be effective with our time whilst programming. After leaning this we all had an idea that a web application would be the best solution as it would make the program easily accessible. Although we were all agreeing on solving the task with a web application most of us were unsure of how to program a web application in C#, which is the mandatory programming language for this semester project. Therefore, we started researching frameworks in which to build our application and ended up with the framework Blazor, which a group member of ours was already familiar with due to previous web-development experience. After introducing the framework to the rest of the group we agreed that Blazor would be ideal for our solution as it allows for both front-end and back-end developments in C# thereby fulfilling the semester goals. Blazor also allows for the front-end components to be build using HTML and CSS with which we were already familiar from the previous semester. This meant that we could put most of our focus into the application logic without having to learn multiple frameworks in our short time frame. Before settling with Blazor we decided to ask our professor in object oriented programming, for his opinion on the framework. He told us based on experience that server-side Blazor was a good idea for us to use although the community behind Blazor large enough to expect valid answers from Google for all problems we might face. We experienced this several times along the way e.g. when attempting to create a timer for updating data we were thrown exceptions after running the application for a period of time. We were able to resolve these issues on our own after many failed attempts, but at those times an older framework with more available information would have been preferable. In general, we are really pleased that we chose Blazor as our framework as we were able to complete more than the required core functionality of the program although we might have missed out on learning other front-end technologies.

7.2 Testing

Our overall approach to testing revolves around documenting that the core functionality of the application works as expected. We also want to ensure that the functionalities of the application fits its technical context. Our application is object-oriented and web based. When we make GET requests to endpoints on the web, we will not receive actual C# objects that can be used in our application. Instead, we receive a response body containing a string, for example in JSON format. Therefore, a recurring challenge in the application is to serialize and deserialize C# objects into formats that can be used on the web. We deem functionalities related to object oriented web programming essential to test. This reasoning was central when we determined which unit tests to perform. The unit tests that we chose to perform mainly consist of testing the *Controllers* and the helper functions used in the API. These units serve the purpose of translating and parsing C# objects to and from formats used on the web, as well as translating and parsing C# objects from our data model to Uniconta's data model. Had we not conducted these tests, we would not have

any way of knowing nor documenting whether our application is able to do such operations or not.

We also considered the fact that the requirements from our MoSCoW analysis should be reflected in our choice of which components to test. This consideration weighted more than the consideration of object oriented web programming when we selected which parts of our application to include in the integration tests. The tests laid out under integration tests, are tests of components that are central in fulfilling the requirements from the MoSCoW analysis. We selected the three most important elements from our problem domain (*Subtask*, *Employee* and *Customer*) and conducted tests that revolved around ensuring that we can GET, POST and PUT data related to these. We were also interested in testing whether the logic used in the related components was working correctly or not.

As our application accesses sensitive data, test reproducibility has been an issue that we have had to face. Due to a signed NDA, we are not allowed to store any data locally, including data used for testing. We could create a local database with dummy data that has properties and entries that mimics the real data, but this would force us to make significant changes to our API, our *Controllers* and our *MetaModels* (C# classes for the database generated by Uniconta). On this basis, we chose not to go with this approach, as significant changes to the code would make the tests of our functions redundant. Instead, we came up with the idea that we could copy the real database and store the copy live on OData, so that our API, our *Controllers* and our *MetaModels* would access live data and therefore be tested exactly as they are implemented. This approach works very well in terms of testing our actual implementation, which is the purpose of testing. However, it is not a very good approach when it comes to scalability and reproducibility, as storing data is not free and the copy will be deleted when the testing is over. Despite being cost inefficient, we preferred this approach in this specific case, as Attiri's database is not very large. It should also be mentioned that the copy of the database easily could be removed again and therefore not take up any space. Regarding test reproducibility, we need to add login credentials to the test project to access live data. This comes at the expense that we are the only ones able to reproduce the tests. Nevertheless, with this approach, testing of our core functionalities has been conducted in a way that is as close to the actual implementation as possible and therefore the approach has been deemed superior.

When it comes to usability testing a few issues arose. The first problem (noted as P1 under Table 9.2) was interesting as it highlighted a potential compatibility issue with the way logging into the platform worked. The conclusion from troubleshooting was that since none of the accounts which Attiri uses were listed under the section of Uniconta, where our account was listed to receive access to a "test and play" environment, within which nothing could be broken. This was identified as a communication error, as that was how the log in system was supposed to function. Though this happening seemed negative it actually helped the group gain a more universal understanding of the system. Had it not occurred confusion may have arose later where one member explaining the issue might have been harder as other members would not have experienced it first hand.

Usability tests are great for getting new insight regarding the user interface. A great example of this would be P2 from Table 9.2 at which we were so accustomed to the navigation bar and what each page contained, that we didn't realize that the label did not reflect the content on the page. Had the test subject not run into this issue, the system might have been delivered with a label having an illogical name. Had that happened, some employees could have been confused upon their first interaction with the system.

During our usability tests, sample size was an issue since only one user could participate, rather than a more desirable amount such as 3 or 5. All potential findings could have been questioned on this basis, as issues found by a single subject might not be representative or even reproducible.

If the participant had lacking technical skills and verbal skills, the usability test would have concluded with much less feedback. The usability test tries to remedy this by utilizing the "think aloud protocol" as mentioned in subsection 6.3.4, but even this can not help lacking verbal skills. It can be added that reminding the subject to utilize the protocol may create a hostile environment. We were supposed to have 2 people go through the usability test, but due to untimely illness one person could participate. The person who did participate, had to go through it one week later than planned, also due to illness. Because of this, no large suggestions for functionality stemming from the usability test were possible to implement, due to the approaching deadline. These delays and the exclusion of one of the subjects stood to potentially make the results of the test almost redundant. Fortunately, the test subject managed to recover from their illness in time and provided plenty of useful data.

7.3 Odata's database failing

The application uses Odata's database and therefore can experience downtime whenever the database is down. Such an issue was first experienced on the 13th of December when Odata's database was down for 4 hours and attempts to access the site would result in a 404 network error. However, Uniconta still works even if the database is down. So in future cases where the database experiences issues, Attiri would have to use Uniconta in the mean time. It is expected that such an issue is very rare. To solve the issue we could make an external database, but because it is outside our MoSCoW analysis and our time constraints we are unable to do so.

7.4 Time Planning

Time planning was organized through Azure Devops' team board. The board can be seen on Figure 9.1, where we created tasks under the section of 'to do' which people could attach themselves to and move the task to 'doing' and when done to 'done'. The system worked quite well as people could pick out tasks which fit their competences. The critique could be raised that it was too unstructured and loose, but any more rigidity and structure to the way we scheduled our time would result in too much time spent having meetings and going through bureaucratic procedures. As such we're quite satisfied with the way we choose to organize our time.

Periodically short team goals (usually for Friday, as most commonly supervisor meetings were on Mondays) would be agreed upon and worked towards. Long term goals for dates upon which certain things should be done were also set up. One such goal could be as following: "On this date the criterion analysis should be deemed done, and sent to our supervisor for review." This, plus the usage of the board mentioned above covers our methods of planning the project.

7.5 Lack of Important Features

While deciding the most important features of the program, as well as the criteria of functions set forth by Attiri, some features were deemed too advanced to implement, due to the time constraint of the project. This meant that through our design process, some features that objectively improved the functionality of the program had to be left out. One such feature was the "Subtask Rollover" feature. The "Subtask Rollover" feature was meant to be in charge of spreading a subtask that exceeded the hourly work limit of an employee across several days, in case a subtask required more time to be completed than available work hours in a day. This means, that in the final version of the program, a subtask assigned to an employee that exceeds the available work hours will automatically result in a warning for the manager. This can give the manager an imprecise overview of the employees and their allocated time, and since there is no way to roll subtasks across several

days, this issue will remain unresolved.

How could we have avoided this problem and implemented this feature as requested by Attiri? The lack of the aforementioned feature comes down to a question of priority and time management. Going into the development phase of the project, we knew that we wanted to secure the core functionality of the program first. If time allowed for it, we could then move on to other features that would improve the functionality of the program. We defined core functionality as the features that ensured that the program was usable in the most basic state. That included things like communicating with the database and visualizing data.

We knew that the "Subtask Rollover" was a feature that was important for the functionality of the program however during the development process we came across several other ideas that took precedent. Thus, this seemingly important feature got delayed due to new ideas and features taking priority. Once we reached a comfortable stage in development, we concluded that the feature was too difficult to implement in the short time that we had left before our deadline. Instead, the remaining time was used to correspond with Attiri, fixing eventual features and implementing any last-minute changes Attiri might have had to our program.

A solution to this problem could have been to prioritize our features differently or set aside the necessary time to develop the "Subtask Rollover" feature. This could have resulted in other features not making it into the final version of the problem, which could have posed other problems. We rarely looked back on our MoSCoW analysis during our development phase and developed most of our features as we saw fit through our daily meetings. Perhaps we could have included a priority list in our daily meetings to constantly keep track of which features were the most important to implement. As we look back on the project, some features and implementations perhaps shouldn't have taken priority over the "Subtask Rollover" feature. Because of being deemed more important at the time, or less difficult to implement, those features were implemented first. In the end, we didn't have enough time to implement a seemingly important functionality of the program.

7.6 Dialogue with Attiri

Throughout our collaboration with Attiri, several challenges surfaced that required communication to overcome. From things like properly integrating our access into their system through creating several logins, to keeping Attiri updated with our progress, communication has been key to ensure that we were staying on target. It has also been important to continuously balance the expectations that Attiri had throughout the development process. Looking back at the development process and the communication with Attiri, some factors could have arguably been done better. While we didn't run into any issues with the way we communicated with Attiri, we recognize that some areas could be improved.

During our correspondence with Attiri, most of the communicative work was left to a single group member, who was in charge of emailing back and forth with our contact at Attiri. While this was not inherently an issue, it could be argued that the rest of the group didn't get to experience communicating and formulating any issues that we ran into, so that Attiri could help us find a solution. Additionally, this could lead to some information being forgotten when communicating whatever Attiri had stated in emails and phone calls back to the group. The result of this could be a large difference in understanding among the group members, as one group member was in charge of communication, while the others at times could be left out of those communications. However, this can also lead to several benefits, like having continuity between conversations and more easily following up on emails and phone calls, since only one person had to retain all of the information exchanged between the group and Attiri. Additionally, one person got more experience formulating our issues and questions to the contact person at Attiri, and perhaps a clearer understanding

between the two parties was gained from that approach. Our approach was effective for our development process, since the one group member in charge of communicating between the group and Attiri, was effective in relaying information back to the group. Additionally, this approach also meant that Attiri always knew who to contact, and always had a contact person to rely on.

Another case of importance when it comes to communication has been periodically contacting Attiri, in regards to questions and issues that we might run into during the development process. During the latter stages of developing the program, we did not communicate much with Attiri and prioritized our time with development and integration. Falling out of contact with the client you are supposed to develop a solution for can have consequences like going down a development path that is undesirable for Attiri. It can also cause Attiri to lose their overview of what is going on in the development process and lose track of how far we have gotten with our solution and what to expect once it is delivered. We were far into our development process before we demoed the software for Attiri and by that time our result was relatively locked in, due to time constraints. This means that if Attiri had been dissatisfied with our solution or result and if it required larger changes, we would not have been able to implement them in time. That would, in the worst case, result in us having spent half a year on development, and handing over a piece of software that ultimately would not be used in the way we foresaw. This potential problem could maybe have been resolved by doing smaller, partial usability tests of the program. Perhaps demoing the individual components could have ensured that we could not end up in a situation where Attiri would be dissatisfied with our solution. In our case, the 2 demos that we performed for Attiri went well, and they seemed interested in using the solution, as well as already having some suggestions for future development.

Chapter 8

Conclusion

In this report, we have established our collaboration with the accounting firm: Attiri defined a problem and proposed a solution; to create a planning tool that utilizes data from their existing database, with the capability of visualizing data and assigning employees a daily schedule. We set out to develop a solution that could achieve the problem statement of section 2.1.

"How can we make a tool that provides the person in charge of planning with an overview over individual employees' workload, and makes it manageable to meet deadlines across multiple projects and tasks?"

Attiri's previous planning system was outdated and relied heavily on meetings and ad-hoc assignments that made the managers of the firm lose oversight. Due to also working with a large number of customers, Attiri also had an issue of meeting their deadlines for their customers, which resulted in fines that negatively impacted the firm. By increasing the overview provided to the employees in charge of managing company time and tasks, and making it easy to assign employees assignments and tasks, the group sought to provide Attiri with a better tool to manage their time and meet their deadlines.

During the development of the solution, we have managed to make a functioning program following our MoSCoW analysis in section 2.5. The program is able to interact with their current database (Uniconta) and visualize and display relevant data in several graphs and charts. Additionally, we have reworked their planning system with permission from Attiri, in an attempt to increase the company workflow and overview. In our proposed solution, we have introduced the context of a "subtask" which breaks down larger projects into smaller daily tasks, that can then be assigned to an employee. The managers in charge of delegating tasks in the firm, can with our solution create subtasks, and delegate the task to an employee. This makes it possible to schedule each employee's day and provides them with a complete overview of how to utilize their work hours. The program communicates with the database in real-time, which means that any and all changes to the database through our program are immediate, and the company can react accordingly to any changes that might be made.

All of this culminates to not only solving the problem statement, but also satisfying descriptors of the objectives of the semester project such as but not limited to [1].

- Analyze and model requirements in the object-oriented paradigm
- Design, program and test an application in the object-oriented paradigm
- Understand and utilize concepts and facilities in both object-oriented analysis, design and programming and, on this basis, construct an application of high, internal and external quality
- Develop a running application that solves user problem
- Carry out systematic user interface evaluation
- Gain skills in balancing multiple learning goals

We concur that the proposed solution set out to achieve its core goal, which is to provide Attiri with a planning tool that visualizes their data and provides them with the means to assign daily tasks in a reactive system. During our development process, we have chosen to prioritize program stability and core functionality, as opposed to neat design and other requested features. We have managed to achieve a stable program with the core functionality in place, we admittedly haven't been able to introduce other key features as requested by Attiri, for example, the "Subtask Rollover"

feature section 7.5. This is due to factors of time, complexity and prioritization.

This leads to the conclusion that the program is usable, and that the most important features were implemented successfully and remain in working order. We believe that the program can be used fully in its current state, and provides Attiri with a better alternative to their current planning methods.

Chapter 9

Appendix

9.1 Appendix

Script

Before anything here's a consent form for you, it's quite standard, and contain things such as "we can use the information from this test, and if you suddenly regret partaking in it you can withdraw your consent."
But let's begin, to start off with we'd like to tell you that we will be using the think aloud protocol, what this means is that we'd like you to say what you think during this test. So just to get started if you could figure out how many windows are in this room and think aloud whilst doing it we can get started.

Now that that's out of the way let's get started. If you now could log into the system that would be nice.

Great, Now create a subtask and have it assigned to an employee.

This continues until all subtask have been completed.

great, now we'll talk about anything that you would like to talk about, things that confused your or things you really liked.

Here's the SUS scale test try and fill it out, and if you have any questions just ask.

Not that we're done feel free to ask if you have any questions, but if not, have a great day.

CONSENT FORM

Date:_____
Project title: _____

I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.

2. **Use of my data in the study**

- I understand that data which can identify me will not be shared beyond the research team (students and academic supervisors) of this study.
- I agree that anonymized and processed copies of my data (i.e. data that do not contain any personally, identifiable information) may be used for the publications and reports describing the project and its results.
- I give permission for my words to be quoted for the purposes described above, only after my identity has been protected by using a pseudonym.

3. **Reuse of my data**

- I give permission for the data (only after anonymization) that I provide to be reused for the sole purposes of future research, publications, presentations and learning.
- I understand and agree that this may involve depositing all my data in an authorized data repository. Only anonymized data will be available to other researchers (e.g. journal reviewer) at a reasonable request.

4. **Security of my data**

I understand that safeguards will be put in place to protect my identity and my data during the duration of the study, and after if my data is stored for future use.The safeguards include storing all my raw data in a secure encrypted system with controlled access provided by Aalborg University.

5. **Copyright**

I give permission for data gathered during this project to be used, copied, excerpted, annotated, displayed and distributed for the purposes described above.

6. **Signatures (sign as appropriate)**

Name of participant (IN CAPITALS)	Signature	Date

Log file

Timestamp	Description	Problem
00:19	The subject is asked to login to the platform and asks which login he should use, he is told to use the one he thinks would work. He becomes confused as to what his login is, as he normally uses autofill to login, the login doesn't work and he finds his phone to see his login and attempts several times without success. Eventually we log him in using our test account.	P1: The problem was later identified as all logins (except ours) did not have access to the test environment and has been fixed.
03:00	Upon being asked to create a subtask the subject begins to look at dashboard and the list of customers, after attempting to locate where subtasks are created for 150 seconds he is told where to do it. Confusion happened due to the page where subtasks are created not having a relevant name, in the the test it was named "Unassigned" in English, rather than "new/unassigned subtask" in Danish. Once shown where to do it, it went well.	P2: Naming convention was was confusing, and as the user was navigating the system for his first time did not know where components were located.
07:55	The subject (unnecessarily) attempt to create an alert for the subtask and struggles with choosing an appropriate date as the field defaults to 00/00/0001, he expresses his desire for it to default to the day before the deadline of the subtask	P3: The input field for the time of alerts defaults to 00/00/0001 rather than the day before the deadline which causes confusion.
10:00	Subject attempts to locate a subtask through the employee page, whilst logged into an employee account which the subtask is not assigned to, he then navigates to the manager overview and locates the subtask and edits it's information	P4: Subject does not know where the components are located and the naming of pages in the navigation bar does not make it obvious.
12:00	Subject navigates to the wrong page but quickly navigate to the right page and identifies the reminder component, an error with the reminder component (the save button is not visible unless the user scrolls within the component, which no scroll wheel displays the option of) prevents him from saving the reminder though, this was fixed shortly after the test.	P5: The Subject briefly navigates to the wrong page and then locates the correct page, but the task becomes complicated as the "save" button is not visible upon first glance.
13:30-16:00	The subject once again does not know where exactly no navigate but eventually finds it	P6: The subject spends a lot of time figuring out where to navigate and ponders if the program will throw exceptions if he moves the start date of a project past its deadline.

Table 9.1: Log File appendix

The Problem List

ID	Problem description	Category
P1	The login does not work, the cause was identified and solved, but this issues stood to render the program unusable. A backup login was planned to avoid it making the usability test impossible.	Serious
P2	The user finds the naming of the items of the navigation bar to not explain where he needs to navigate, and as such just goes thorough what he finds to be good guesses, eventually he requires assistance and when helped quickly solves the task.	Serious
P3	The user perceives creating an alert as a requirement to creating a subtask and decides to do so, through this he expresses that the date field for alerts should not default to 00/00/0001 but rather something useful.	Cosmetic
P4	The user did not know where the component was to be found and as such spent some time finding it, this caused a minor delay	Cosmetic
P5	The user spent a bit of time finding the component and upon finding the component had the be shown the save button as it was hidden from sight.	Cosmetic
P6	The user did not know where the component was to be found and as such spent some time finding it. This problem should be categorized as serious due to the time of the delay, but considering that the user was using the system for his first time it would seem fair to consider it a cosmetic problem	Cosmetic

Table 9.2: The Problem List

SUS Questionnaire

		Disagree	**Partly disagree**	**Neither / nor**	**Partly agree**	**Agree**
	Question	1	2	3	4	5
1	I think that I would like to use this system frequently.					X
2	I found the system unnecessarily complex.		X			
3	I think the system was easy to use.				X	
4	I think that I would need the support of a technical person to be able to use this system.	X				
5	I found the various functions in this system were well integrated.				X	
6	I think there were too many inconsistency in this system.			X		
7	I would imagine that most people would learn to use this system very quickly.					X
8	I found the system very cumbersome to use.	X				
9	I felt very confident using the system.				X	
10	I need to learn a lot of things before I could get going with this system.		X			

Table 9.3: SUS questionnaire

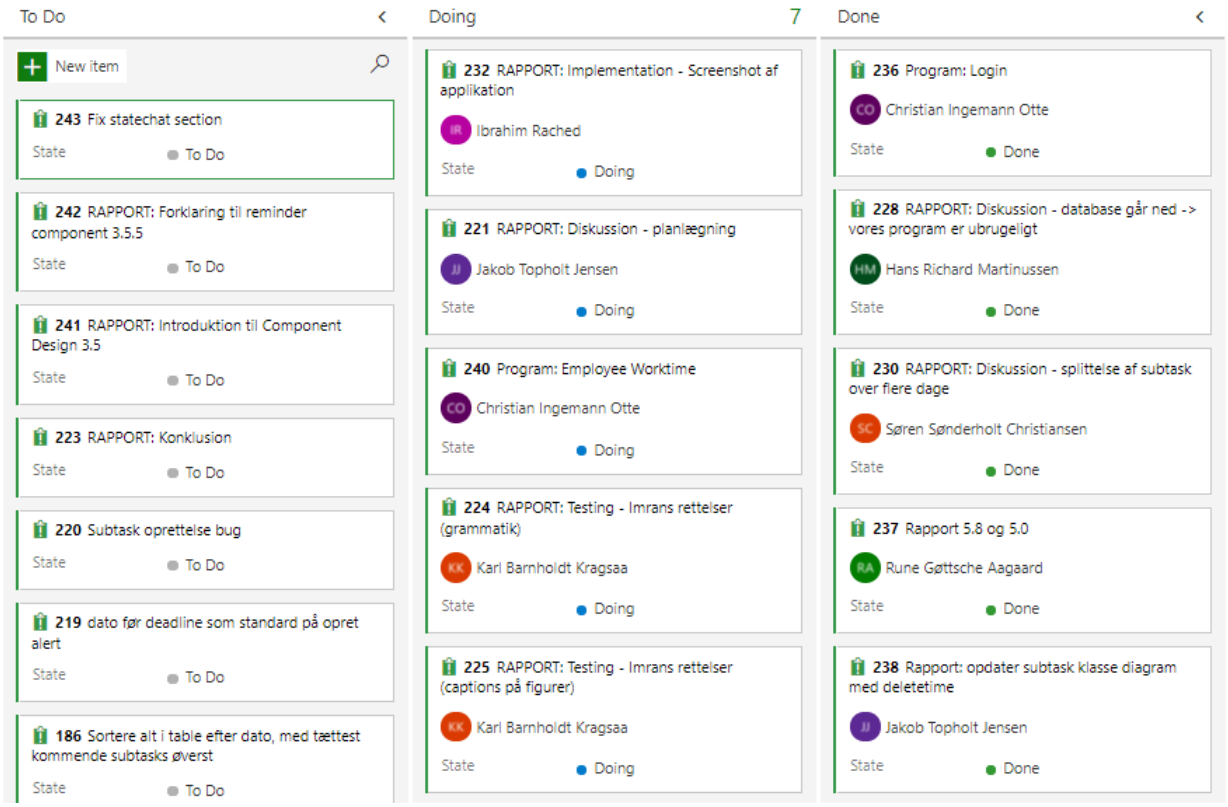


Figure 9.1: Devops assignment board

Bibliography

[1] AAU. *A Well-structured Application (2021/2022)*. https://moduler.aau.dk/course/2021-2022/DSNDATFB310. (Accessed on 12/07/2021). 2019.

[2] David Benyon. *Designing User Experience*. 2019.

[3] Rubin Chisnell. *Handbook of Usability Testing*. John Wiley And Sons Ltd.

[4] patrick Fletcher. *Introduction to SignalR | Microsoft Docs*. https://docs.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr. (Accessed on 12/16/2021). Sept. 2020.

[5] Interaction Design Foundation.

[6] Interaction Design Foundation. *Design iteration brings powerful results. So, do it again designer! | Interaction Design Foundation (IxDF)*. https://www.interaction-design.org/literature/article/design-iteration-brings-powerful-results-so-do-it-again-designer. (Accessed on 12/01/2021). Feb. 2021.

[7] Mike Hughes. *Talking Out Loud Is Not the Same as Thinking Aloud :: UXmatters*. https://www.uxmatters.com/mt/archives/2012/03/talking-out-loud-is-not-the-same-as-thinking-aloud.php. (Accessed on 11/30/2021). Mar. 2012.

[8] Jakob Nielsen. *Thinking Aloud: The #1 Usability Tool*. https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/. (Accessed on 12/02/2021). Jan. 2021.

[9] restfulapi.net. *What is REST*. https://restfulapi.net/. (Accessed on 11/20/2021). Oct. 2021.

[10] Jan Stage. *OBJECT ORIENTED ANALYSIS DESIGN 2. ed.* 2018.

[11] TECED. *Field Usability Testing - TecEd*. https://www.teced.com/services/usability-testing-and-evaluation/field-usability-testing/. (Accessed on 12/02/2021). 2020.

[12] TECED. *Laboratory Usability Testing - TecEd*. https://www.teced.com/services/usability-testing-and-evaluation/lab-usability-testing/. (Accessed on 12/02/2021). 2020.

[13] Usability. *System Usability Scale (SUS) | Usability.gov*. https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. (Accessed on 12/01/2021). Unknown.