

Christian Oxås Nilsen
Ine Sofie Zile Løseth
Jakob Holkestad Molnes
Kacper Lukasz Nowicki

Årets Nyhetsjeger

A quiz system with an accessible game interface
and an AI powered dashboard

Bacheloroppgave i Dataingeniør
Veileder: Anniken Susanne Th. Karlsen
Mai 2024



NTNU

Kunnskap for en bedre verden

Christian Oxås Nilsen
Ine Sofie Zile Løseth
Jakob Holkestad Molnes
Kacper Lukasz Nowicki

Årets Nyhetsjeger

A quiz system with an accessible game interface and
an AI powered dashboard

Bacheloroppgave i Dataingeniør
Veileder: Anniken Susanne Th. Karlsen
Mai 2024

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for IKT og realfag



Kunnskap for en bedre verden

0.1 Abstract

Sunnmørsposten (SMP) has set a goal to increase the interest for news among those under 40 years old. They expect that games are a way to engage readers and increase readership. Traditionally, many paper newspapers include games such as crossword and Sudoku. SMP sees that nowadays news can be read on websites, mobile applications, social media, or other platforms. They want to stay on top of the trends so they can engage a wider audience.

Sunnmørsposten wanted a quiz system where the questions were based on their articles and players could compete on leaderboards, to win real-life prizes and become the "News Hunter of the Year."

With this goal in mind, the project aimed to create a fun and engaging user interface for players, and an administration panel where quizzes could be created and managed. The team also suggested to create questions with the help of artificial intelligence to streamline the process, and *Sunnmørsposten* was very receptive to the idea.

We have delivered a Web based quiz application that can be integrated into *Sunnmørsposten*'s website and mobile app. The GitHub repository includes instructions on how to setup the server and run the application. The project fulfills all the requirements from *Sunnmørsposten*. Quizzes are easy to create with AI, the users can play quizzes, and the players can see themselves and others on the leaderboards. Additionally, administrators can manage leaderboards and users.

To achieve this result, we depended on a variety of technologies such as: Go for application logic, Templ for templating, HTMX for swapping HTML elements, ChatGPT for generating questions, PostgreSQL for storing data, GitHub for version control, and Docker for containerization. We adopted an Agile methodology to conduct our bachelor's thesis. The team had regular meetings with the stakeholder, to ensure that the project did not deviate from their expectations and requirements.

0.2 Sammendrag

Sunnmørsposten (SMP) har satt som mål å øke interessen for nyheter blant de under 40 år. De forventer at spill er en måte å engasjere lesere og øke lesertall. Tradisjonelt så har mange papiraviser inkludert spill som kryssord og Sudoku. SMP ser at nå til dags kan nyheter leses på nettsider, apper, sosiale media og andre plattformer. De ønsker å holde seg oppdatert på trender slik at de kan engasjere flere lesere.

Sunnmørsposten ønsket et quiz system med spørsmål basert på deres artikler og spillerne kan konkurrere om å havne på topplistene, for å vinne premier og bli "Årets Nyhetsjeger."

Målet med prosjektet var å lage et gøy og engasjerende brukergrensesnitt for spillere, og et administrasjonspanel for å lage og administrere quizer. Teamet foreslo også at spørsmål kunne bli lagd med kunstig intelligens for å effektivisere prosessen, og *Sunnmørsposten* var veldig åpne for idéen.

Vi har levert en Web basert quiz applikasjon som kan integreres med Sunnmørspostens nettside og mobilapp. Vi inkluderer instruksjoner for hvordan man kan sette opp serveren og kjøre applikasjonen på GitHub. Prosjektet oppfylder alle kravene fra Sunnmørsposten. Quizer er enkle å lage med KI, brukerne kan spille quizer, og spillerne kan se seg selv og andre på topplistene. I tillegg kan administratorer administrere topplistene og brukerne.

For å oppnå dette resultatet var vi avhengige an en rekke teknologier. Vi brukte Go for applikasjonslogikk, Templ for templating, HTMX for å bytte ut HTML elementer, ChatGPT for å generere spørsmål, PostgreSQL for lagring av data, GitHub for versjonskontroll, og Docker for containerisering. Vi tok i bruk en Agile metodikk for gjennomføre bacheloroppgaven. Teamet hadde regelmessige møter med produkteieren, for å forsikre oss om at prosjektet holdt seg til forventningene og kravene.

PREFACE

This thesis is part of our education as computer science students at NTNU. We would like to thank Sunnmørsposten for making this project possible, and a special thanks to Liv-Jorunn Håker, Dag-Arne Alnes, and Hanna Relling Berg for a good, collaborative process.

Thanks to all testers from Sunnmørsposten and fellow students who participated in user testing and gave valuable feedback.

Our sincere appreciation to our supervisor Anniken Susanne Th. Karlsen, who supported and encouraged us throughout the whole process, giving us feedback and guidance when we needed it.

Our heartfelt thanks to Arne Styve, our study program manager, for good advice and guidance.

CONTENTS

Abstract	i
0.1 Abstract	i
0.2 Sammendrag	ii
Preface	iii
Contents	iv
List of Figures	ix
List of Tables	xi
List of Code	xii
Glossary	xiii
Acronyms	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Project description	1
1.3 Stakeholders	1
1.3.1 History of Digital News	1
2 Theory	3
2.1 Games	3
2.1.1 Gamification	3
2.2 Design and User Experience	5
2.2.1 Wireframes	5
2.2.2 Accessibility	6
2.2.3 WCAG Standard	6
2.3 Software Architecture	6
2.3.1 Monolith	6
2.3.2 Model-View-Controller	7
2.4 Software Development	7
2.4.1 Version Control	7
2.4.2 Agile	8

2.4.3	DevOps	9
2.4.4	Linters and Formatters	9
2.4.5	Cohesion and Coupling	9
2.4.6	Refactoring	10
2.4.7	Code Reviews	10
2.4.8	Documentation	10
2.4.9	Software Testing	10
2.4.10	Conceptual Frameworks for Software Testing	12
2.5	Web Development	13
2.5.1	HTTP and Network Communication	13
2.5.2	HTML, CSS and JS	14
2.5.3	Web Frameworks	14
2.5.4	CSS Frameworks	15
2.5.5	Server-Side Rendering	15
2.5.6	REST API	15
2.6	Programming Paradigms	15
2.6.1	Imperative Programming	16
2.6.2	Declarative Programming	16
2.6.3	Object-Oriented Programming	16
2.6.4	Functional Programming	16
2.6.5	Multi-paradigm Programming	16
2.7	Database Management System	16
2.7.1	Relational Databases	17
2.7.2	SQL	17
2.7.3	Database Normalization	17
2.7.4	Seeding	17
2.7.5	Schema Migrations	18
2.8	Security	18
2.8.1	Cryptography	18
2.8.2	Authentication	18
2.8.3	Common Vulnerabilities	18
2.9	Virtualization	19
2.10	Artificial Intelligence	19
3	Methods	21
3.1	Team and Project	21
3.1.1	Project Methodology	21
3.1.2	Jira	21
3.1.3	Confluence	21
3.1.4	Discord	22
3.2	Design and Prototype	22
3.2.1	Models and Diagrams	22
3.2.2	Figma	22
3.3	Environment and Development	22
3.3.1	Git	22
3.3.2	GitHub	23
3.3.3	Code Editor	23
3.3.4	SSH	23

3.3.5	WSL	23
3.3.6	Docker	24
3.3.7	Adminer	24
3.3.8	Secret Management	24
3.4	Solutions	24
3.4.1	Go	24
3.4.2	Echo	24
3.4.3	Air	24
3.4.4	Templ	25
3.4.5	HTMX	25
3.4.6	JavaScript	25
3.4.7	Tailwind CSS	25
3.4.8	GNU Make	25
3.4.9	Shell Scripts	25
3.4.10	PostgreSQL	25
3.4.11	MinIO	26
3.4.12	Caddy	26
3.4.13	AI	26
3.5	Quality Assurance	26
3.5.1	Google Lighthouse	26
3.5.2	Code Reviews	27
3.5.3	Sonarlint	27
3.5.4	Unit Tests	27
3.5.5	Integration Tests	27
3.5.6	End-to-end Tests	27
3.5.7	Usability Testing	28
4	Results	29
4.1	Administrative Process	29
4.1.1	Meetings	29
4.1.2	Project Management	30
4.1.3	Time Management	30
4.1.4	CI/CD	30
4.2	Features	31
4.2.1	Account Creation	31
4.2.2	Usernames	31
4.2.3	Navigation Menu	31
4.2.4	Play Quizzes	34
4.2.5	Point System	39
4.2.6	Guest Users	40
4.2.7	Public Leaderboard	40
4.2.8	Completed Quizzes	41
4.2.9	Profile	41
4.2.10	Manage Labels	41
4.2.11	Manage Quizzes	41
4.2.12	Other Admin Features	44
4.2.13	Interaction Feedback	48
4.2.14	Error Handling	48

4.3	Engineering Results	49
4.3.1	Architecture	49
4.3.2	Database	57
4.3.3	Domain Model	59
4.3.4	Prototyping	60
4.3.5	User Interface	62
4.3.6	HTTP Routes	63
4.3.7	Object Store	63
4.3.8	Security Measures	63
4.4	Theoretical Results	64
4.4.1	Gamification	64
4.5	Quality Assurance	64
4.5.1	Documentation	64
4.5.2	Code Quality	64
4.5.3	Unit Testing	65
4.5.4	Integration Testing	66
4.5.5	End-to-end Testing	68
4.5.6	Usability Testing	69
5	Discussion	71
5.1	Theoretical Discussion	71
5.1.1	Technology	71
5.1.2	Development Process	72
5.1.3	Gamification	72
5.1.4	Anti-cheating Measures	73
5.1.5	Leaderboard	73
5.1.6	Design	74
5.1.7	Security Measures	74
5.2	Engineering Discussion	75
5.2.1	Inline Frame	75
5.2.2	Usernames	75
5.2.3	Quiz Creation	76
5.2.4	Play Quiz	77
5.2.5	Guest Mode	77
5.2.6	Point System	78
5.2.7	Limitations of HTMX	80
5.2.8	Code Quality	81
5.3	Reflection	81
5.4	Future Work	81
5.4.1	Gamification	81
5.4.2	Notifications	82
5.4.3	User Authentication and Verification	82
5.4.4	Exclusive Leaderboards	82
5.4.5	Bucket	83
6	Conclusions	85
	References	87

Appendices:	97
A GitHub repository	98
B Diagrams	99
C User Stories	102

LIST OF FIGURES

1.3.1	Percentage of the Norwegian population that uses the medium on an average day [6].	2
2.1.1	A division-based leaderboard in <i>Duolingo</i>	4
2.1.2	A <i>Bitmoji</i> avatar in <i>Snapchat</i>	4
2.4.1	The test pyramid. Source: <i>Test Pyramid</i> by Martin Fowler. . . .	12
2.4.2	The test trophy. Source: <i>The Testing Trophy and Testing Classifications</i> by Kent C. Dodds.	13
3.1.1	Scrum framework by Dr Ian Mitchell. Published under CC0 1.0 .	21
4.1.1	Successful runs of unit and integration test jobs on GitHub actions	31
4.2.1	The navigation menu on a larger screen.	32
4.2.2	The closed navigation menu on a smaller screen.	32
4.2.3	The opened navigation menu on a smaller screen.	33
4.2.4	The admin navigation menu on desktop.	33
4.2.5	The admin navigation menu on mobile.	34
4.2.6	A quiz card for "Påske Quiz!"	35
4.2.7	A quiz card for "Påske Quiz!" when the "play" button is hovered.	36
4.2.8	An answered question that was correct, where 100% of users chose the first alternative.	37
4.2.9	The result of the quiz, points-wise. Three screenshots at different times, side-by-side. The circle is filled with a wave animation, gradually going from the bottom to halfway up, since the player got half of the possible points.	37
4.2.10	The quiz summary page, showing the results of the quiz.	38
4.2.11	The list of articles that the questions were based on.	38
4.2.12	Final iteration of the point system.	39
4.2.13	The public leaderboard, with six users opted-in for competition.	40
4.2.14	Edit page for labels.	41
4.2.15	The page for editing a quiz. Top part (1/2).	42
4.2.16	The page for editing a quiz. Bottom part (2/2).	43
4.2.17	The modal window for editing a question.	44
4.2.18	The username management page. The adjective table (left) and noun table (right) displays all combinations for usernames. . . .	45
4.2.19	Searching for a word applies to both tables. The tables display any words that contain the searched text.	45

4.2.20	Clicking on the row displays an input for editing the word and a button to delete it.	46
4.2.21	When a word is changed or deleted, an undo button will show up. The change is not saved immediately.	46
4.2.22	A word that is marked for deletion. The change is not saved immediately.	47
4.2.23	To save the changes, press the "Lagre" button. To reset all changes, press the "Nullstill" button.	47
4.2.24	Below each table, there are buttons to change the page to see a different selection of words.	47
4.2.25	To add a new word to either list, write it in the input field above the table and click on the (+) button.	48
4.2.26	Error page displayed when a non-admin user tries accessing the admin dashboard.	49
4.3.1	High level architecture overview.	50
4.3.2	Database schema.	58
4.3.3	Domain model from the perspective of a Nyhetsjeger user.	60
4.3.4	Domain model from the perspective of an administrator.	60
4.3.5	Early wireframes for the application.	61
4.3.6	Early wireframes for the dashboard.	61
4.3.7	High-fidelity designs for parts of the application.	62
4.5.1	A <i>SonarLint</i> rule shown in <i>Visual Studio Code</i>	65
4.5.2	Bruno test results, from Bruno CLI	68
4.5.3	Bruno test suite	69
5.1.1	"quiz2" is active from some time in January, but it ends in February.	74
5.2.1	Second iteration of the point system. The range of points rewarded is limited to three constants.	78
5.2.2	Third iteration of the point system. The value decrease linearly until it reaches the minimum (20% of the maximum points) at the question's time limit.	79
B.0.1	Use case diagram of the Nyhetsjeger portal.	99
B.0.2	Use case diagram of the player.	100
B.0.3	Use case diagram of the administration.	101

LIST OF TABLES

4.3.1 "user_question_points" SQL view. The last column is "points_awarded". 59

4.3.2 "user_quizzes" SQL view. The last column is "answered_within_active_time". 59

LIST OF CODE

3.1	Caddy example config for a reverse proxy	26
4.1	.env file	53
4.2	Docker file	54
4.3	Docker Compose file	57
4.4	Setup and tear down functions for the base integration test suite.	67
4.5	Setup and tear down functions for integration tests.	67
4.6	Users module integration test suite setup.	67
5.1	Example of how "hx-target-error" can be set up in a button. . . .	80
5.2	Example of setting the response header in Go's Echo framework .	80

GLOSSARY

- architecture** The internal structure of an application and how it is modularized. 6, 7, 22, 49
- bug** An unintended error in the code that causes it to behave unexpectedly or not as intended. 10, 80
- cache** A temporary storage for storing frequently accessed data, reducing the time needed to retrieve it from its original source. 15
- client** A device or application that requests and receives services or resources from a server. 13–15, 25, 68
- code smell** A characteristic of code that can indicate a problem. Violation of standard coding practices. Not necessarily a bug, but can lead to one. 9, 10
- compilation** Code compilation is the process of turning human-written code into code that is executable by the computer. 24, 30
- CRUD operation** Create, read, update, delete. The four essential operations for data persistence. 17
- framework** A set of tools that provides a foundation for application development. It offers reusable code, libraries, and conventions to streamline development processes. 14, 15, 25
- frontend** The visual part of the product. The things the user sees and interacts with. 14, 63
- server** A program or device that provides services or resources to clients over a network. 8, 13–15, 26, 27, 68
- standard library** A library available by default in a programming language. 63
- URL** Short for Uniform Resource Locator. It is an address that references a resource on the Web. 41, 63, 70, 71, 76
- UUID** Short for Universally Unique Identifier. It consists of 128 bits. 57, 63

ACRONYMS

- AI** Artificial Intelligence. 1, 19, 29, 72, 75, 76
- API** Application Programming Interface. 7, 15, 24–27, 29, 40, 49, 52, 63, 68, 72, 80, 82
- CI/CD** Continuous integration / Continuous delivery. 9, 10, 23, 30, 53
- CLI** Command-line interface. x, 27, 53, 68
- CSR** Client-side Rendering. 15
- CSS** Cascading Style Sheets. 14, 15, 25, 53, 54, 63, 71
- CSV** Comma-separated values. 75
- CVCS** Centralized Version Control System. 8
- DB** Database. 1
- DBMS** Database Management System. 16, 18, 25
- DTO** Data Transfer Object. 7
- DVCS** Distributed Version Control System. 7, 8
- EU** European Union. 6
- GUI** Graphical User Interface. 27
- HTML** HyperText Markup Language. 14, 15, 25, 26, 49, 62, 63, 68, 80
- HTMX** Hypertext markup extensions. 25, 63, 71, 80, 81
- HTTP** Hypertext Transfer Protocol. 13, 14, 24, 25, 48–50, 63, 68, 75, 80
- HTTPS** Hypertext Transfer Protocol Secure. 14, 26, 50, 63
- IDE** Integrated Development Environment. 9, 23
- JS** JavaScript. 14, 15, 25, 52, 68, 77, 80, 81

JSON JavaScript Object Notation. 29, 68

LLM Large Language Model. 19, 26

LSP Language Server Protocol. 23

MVC Model-View-Controller. 7, 49

NTNU Norwegian University of Science and Technology. 21, 23, 85

OOP Object-oriented programming. 16

PR Pull Request. 23, 27

REST Representational State Transfer. 11, 15

SDK Software Development Kit. 63, 72

SDLC Software Development Life Cycle. 9, 10

SMP Sunnmøreposten. 1, 29–31, 41, 70, 72–77, 81, 82, 85

SQL Structured Query Language. 17, 18

SSB Statistisk sentralbyrå. 2

SSL Secure Sockets Layer. 14

SSR Server-side Rendering. 15

TCP/IP Transmission Control Protocol/Internet Protocol. 13

TLS Transport Layer Security. 14, 50

UI User Interface. 5, 7

UML Unified Modeling Language. 22

VSCode Visual Studio Code. 23

W3C World Wide Web Consortium. 6

WCAG Web Content Accessibility Guidelines. 6, 62, 74

WSL Windows Subsystem for Linux. 23

INTRODUCTION

1.1 Motivation

We found the project interesting, as it provided a challenge regarding Artificial Intelligence (AI), Databases, graphic design and user experience. An additional incentive for choosing this project was the liberty to select technologies of our choice, and the loose, open-ended project requirements, giving the team room for creativity.

1.2 Project description

The project delivers a Web based quiz system. It includes a quiz side for the users, where they can play quizzes and compete in leaderboards. For the quiz administrators, there is a dashboard designed to optimize quiz management. It features an intuitive user interface and suggests questions utilizing AI, streamlining the quiz creation process. Administrators can view the leaderboards and get user contact information, in order to reward the top players. Additionally, the system can be integrated into other Web based solutions.

1.3 Stakeholders

Sunnmøreposten is a media house founded in 1882 by Olaf Sandvig and Hans Tybring von Zernichow [1]. In 1997, the online newspaper "smp.no" was started. Today, *Sunnmøreposten* hosts podcasts, games and news through their website, along with selling physical and digital newspapers. *Sunnmøreposten* is the largest media house in Møre og Romsdal with 73 400 unique daily users [2].

1.3.1 History of Digital News

Mass media is a term describing media with widespread exposure, reaching the vast public [3]. The term has evolved over time, capturing the invention and rise of technologies. Initially it captured printed media, but throughout the 20th century technologies such as radio, television and the Internet were included.

In 1980, *The Columbus Dispatch* was the first to publish an online version of their newspaper [4]. Since then, many newspapers have followed suit. In Norway, the first three major newspapers to go digital was *Panorama*, *Brønnøysunds Avis*, and *Dagbladet* in March of 1995 [5].

As the world becomes digitized, consumers turn to digital news media [6]. Statistics from *Statistisk sentralbyrå* show this trend among the Norwegian population in recent years, as illustrated in figure 1.3.1.

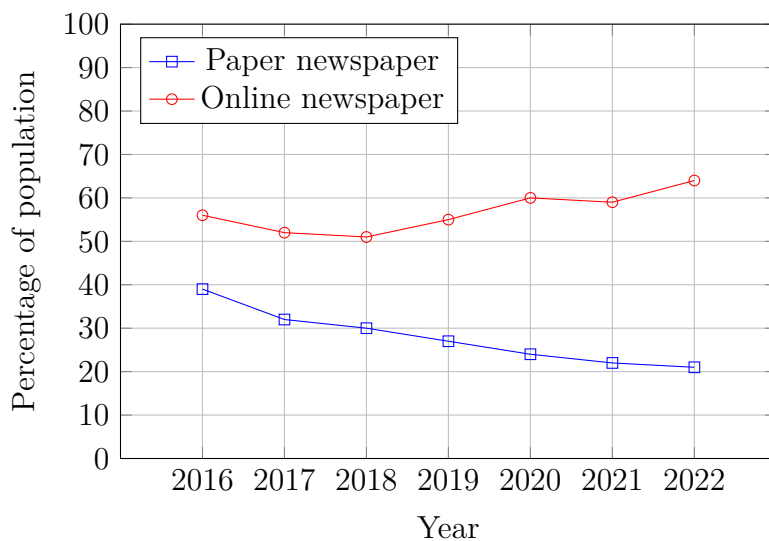


Figure 1.3.1: Percentage of the Norwegian population that uses the medium on an average day [6].

2.1 Games

With digital newspapers it is possible to have other games than in traditional newspapers. If a game becomes popular, it can drive traffic to their website and attract new readers [7]. An example of this is the viral game *Wordle*.

Wordle was developed by Josh Wardle and became open to the public in October 2021 [8]. It was then bought by *The New York Times* in January 2022 and became available for everyone to play on their website, free and without needing to sign in. In between this period, the game had about 300 000 daily users [9]. After the acquisition, that number grew to the millions.

The New York Times also offers a variety of other games that poses new challenges daily. By using this formula of presenting daily challenges, readers can include these games as part of their daily routine and it has a habit-forming effect [10].

There are also games from traditional newspapers that can be used in digital newspapers. One such game is the quiz. According to the *Oxford English Dictionary*, a quiz can be defined as "a series of questions asked of competing individuals or teams, and often divided into rounds" [11]. With digital media, it is possible to apply more gamification techniques than in traditional newspapers.

2.1.1 Gamification

Gamification is the act of adding game elements to non-game contexts, with the purpose of enhancing the experience [12]. These enhancements should make an activity more fun to do, and encourage certain behaviours.

One common system in games is the point system [13]. There exist various types of point systems that are used for different scenarios based on the context of the scenario. A few examples would be: reputation, experience, and redeemable points. The points are rewarded based on their context and can give a sense of progression, reward, or challenge to earn the highest amount of points.

Badges are a reward for completing non-trivial challenges [14]. Each badge is rewarded after finishing a specific non-trivial amount of work or special effort. Badges can be seen as prestigious to receive, therefore they act as incentives.

Leaderboards rank players according to a score. The way the score is calculated depends on the system used [15]. It can measure the performance of players and create a competitive spirit of wanting to beat the ones on top or maintaining a top score. While it seems leaderboards increase the competitiveness of the top players, they may be a demotivator for players on the low-end of the leaderboards.

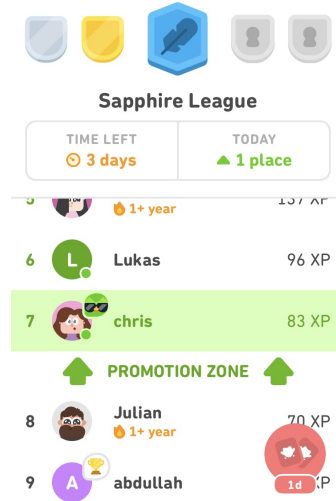


Figure 2.1.1: A division-based leaderboard in *Duolingo*

Avatars are the personification of the player in the digital space [16]. These can range from simple icons to fully animated models. Since avatars are used to convey the players identity, allowing for customization and unlocks may incentivize players to play until they unlock a certain customization.



Figure 2.1.2: A *Bitmoji* avatar in *Snapchat*

Streaks are incentives where a player gets rewards for doing an action repetitively [17]. This may be daily log-in rewards, or getting a bonus for answering several questions correct in a row, like in the online game *Kahoot* [18]. Streaks may be used to create habits by making a player log in every day, or else they will lose out on rewards with significant value. It can also be used to reward answering

questions accurately instead of just quickly, in cases where haste affects the score. Implementing streaks in the future market may be tough, as new rules and regulations come into place. For example, China banned certain mechanics that could be potentially predatory in December 2023, such as daily log-in rewards [19].

2.2 Design and User Experience

There are many concepts that are involved in graphic design such as color theory, composition, hierarchy, balance, scale, and contrast [20].

In the book *The Design of Everyday Things*, Don Norman discusses several design principles that are important in general, but which can also be applied to the design of digital products [21]. His six principles are visibility, feedback, constraints, mapping, consistency, and affordance.

When it comes to designing User Interfaces for the Web, Jakob Nielsen's ten heuristics are well-known and a good rule of thumb [22]. They are:

1. Visibility of System Status
2. Match Between the System and the Real World
3. User Control and Freedom
4. Consistency and Standards
5. Error Prevention
6. Recognition Rather than Recall
7. Flexibility and Efficiency of Use
8. Aesthetic and Minimalist Design
9. Help Users Recognize, Diagnose, and Recover from Errors
10. Help and Documentation

2.2.1 Wireframes

Wireframes are simple sketches of an application that focuses on content layout and user experience. "The wireframe usually lacks typographic style, color, or graphics, since the main focus lies in functionality, behavior, and priority of content" [23].

They are useful for designers to showcase how the User Interface will look and function to product owners, users, and other stakeholders, before the application is made.

2.2.2 Accessibility

It is always important to design applications so that they are accessible to everyone regardless of disabilities and other factors, but especially if it has a large and diverse user base.

It is estimated that up to 1 in 12 men and 1 in 200 women have congenital red–green color blindness [24]. The choice of colors is therefore important to consider, so that the application functions the same regardless of color blindness. In addition to using color to impart information, it is possible to use symbols to convey the same meaning.

”As of 2015, there were 940 million people with some degree of vision loss. 246 million had low vision and 39 million were blind” [25]. A significant percentage of the population has some form of visual impairment. Using a color scheme with high contrast and bright colors can make it easier to see the details of the application. Larger text size and easily-readable fonts are also important. Other features, such as ARIA labels [26], can help assistive technologies like screen readers by adding extra information to the webpage.

The webpage elements, such as buttons and links, must be easy to click and interact with for those with mobility issues.

Animations should not be excessive for those with vestibular motion disorders. Some devices have accessibility features for those who prefer reduced motion, but it is also up to the website to implement it [27].

2.2.3 WCAG Standard

Web Content Accessibility Guidelines (WCAG) is a set of guidelines created by W3C [28]. Its purpose is to make Web content more accessible. Norway follows the EU’s Web Accessibility directive, which requires websites and mobile applications of public sectors to conform to WCAG 2.1 Level AA.

WCAG 2.1 contains three levels; A with 30 success criteria, AA with 20 success criteria, and AAA with 28 success criteria [29]. According to *Tilsynet for universell utforming av ikt*, Norway’s public sector has to conform to 48 requirements and the private sector to 35 [30].

These requirements include, amongst others; must have alternative text for non-text content, the option to display content in either portrait or landscape mode, and titles and headings must be descriptive of its content [30].

2.3 Software Architecture

2.3.1 Monolith

Monolith is a type of software project architecture where all parts of the code is run together, as opposed to a microservice architecture, where every service is run separately [31]. Both approaches have pros and cons, but for this projects a monolith fits better.

While monoliths can be harder to update as the project grows because of the interlinked code, a small code change might lead to a large effort to refactor the code base, for smaller projects where the scope is known, and the boundaries for

what is needed is understood, monoliths tend to work better [31]. Microservices can be a burden to keep up with the lack of larger infrastructure needed to support them.

2.3.2 Model-View-Controller

Model-View-Controller (MVC) is a software architecture pattern in which the application is divided into three layers (or components), namely the model, the view and the controller [32]. The layers are isolated and designed to fulfill their responsibilities.

The model layer is responsible to implement the business logic of the application [32]. It defines the application's data, handles the processing of data, and sets the business rules. It can also be responsible for the persistence of data, for example by communicating with a database. This layer is not concerned about the presentation of application data to the user [32]. It does not implement any interactions with them. Instead it provides an API for other application components, allowing for data retrieval and manipulation.

The view layer is responsible for the presentation of model data and all user interactions [32]. It implements all UI logic. This layer does not communicate directly with the model. Instead, it directs all user input to the controller for processing. It may be worth mentioning that the view can perform some initial input validation.

The controller brings the two previous layers together [32]. It is responsible for processing user input, it interacts with the model, and determines how to respond to user requests. All views presented to the user are provided by the controller, while being updated with data coming from the model.

To further decouple the model and view layers, a Data Transfer Object (DTO) can be used [33]. It is essentially an object carrying data [34] [33]. It "packs" parameters or return values, allowing to reduce number of calls or values returned.

By using DTOs, highly specialized views can be created without impacting the underlying models [33]. For instance, in a quiz application; a special DTO called "UserAnswer" can hold a "Question" object as well as "ChosenAlternative." This way, a feedback view can get all the data required in a single data structure.

2.4 Software Development

2.4.1 Version Control

"Version control is the practice of tracking and managing changes to software code" [35].

Version control is paramount for development, as it keeps track of changes in source files. It allows developers to view and revert to previous versions of the project. This is also beneficial when multiple persons contribute to the same project on different machines, to keep the files synchronized [36].

A Distributed Version Control System (DVCS) is a type of version control in which the source code and history is mirrored on each users computer. The approach is similar to a peer-to-peer network [37].

One such system is Git, which is open-sourced [38]. It was originally created by Linus Torvalds in 2005, because none of the existing solutions met his needs. Git is now the most popular version control system.

When introducing changes, Git requires a commit message. These messages can be standardized using a specification such as Conventional Commits [39]. It aims to standardize commit messages to be human and machine readable. This allows for easy human reviews and enables automated tools to generate change logs and apply semantic version tags for software releases. Following this specification, commit messages should start with standardised keywords like "fix:", or "refactor:".

A Centralized Version Control System (CVCS) keeps the source code and version history on a master server. Each user will need to commit their changes to this central repository. The main difference between a CVCS and a DVCS is how they approach storing the repository. CVCS goes for a similar approach to client-server, unlike DVCS which is similar to peer-to-peer [37].

2.4.2 Agile

Agile software development are methods that follow certain principles in the 2001 Manifesto for Agile Software Development that popularized agile software development [40]. The 12 principles can be summarised as follows [40]:

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Deliver working software frequently (weeks rather than months).
4. Close, daily cooperation between business people and developers.
5. Projects are built around motivated individuals, who should be trusted.
6. Face-to-face conversation is the best form of communication (co-location).
7. Working software is the primary measure of progress.
8. Sustainable development, able to maintain a constant pace.
9. Continuous attention to technical excellence and good design.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. Best architectures, requirements, and designs emerge from self-organizing teams.
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly.

From these principles, agile frameworks such as Scrum and Kanban was developed to assist developers in their workflow [40].

2.4.3 DevOps

DevOps is the concept of bringing both development and operations together into the same team [41]. Traditionally, separate teams would manage each area, one team would be responsible for both automating building, testing and the automatic provisioning of the infrastructure needed to run the product.

Another way to say that, is "Continuous Integration" and "Continuous Delivery", or shortened to CI/CD [41]:

- **Continuous Integration** is the practise of frequently merging code changes from developers into a central repository. As changes are merged, automated builds and tests are run to detect potential issues early in the development cycle.
- **Continuous Delivery** is the automated process of delivering software changes to production-like environments (such as testing). This means code changes that pass CI testing are automatically packaged and prepared for production deployment.

Software code quality refers to the code being among other factors maintainable, reliable, clear, secure and fit for purpose [42]. Higher quality code is easier to work with, build upon, diagnose issues and debug.

There are many ways to improve code quality: introducing processes into the Software Development Life Cycle (SDLC), use of automated tools, and being mindful of the best practices.

2.4.4 Linters and Formatters

A simple way to improve code quality is to use a linter; an automated tool that often comes as an IDE plugin [43]. Linters are designed to point out code smells, common issues and vulnerabilities. They often provide the developer with explanation as to why something is a code smell, the severity level and they may even suggest solutions.

Code style is personal and can be a debatable topic [44]. Consistent code style increases the overall code quality. It makes the code more readable and maintainable. Style can be enforced by an automated formatter tool. It styles the source code automatically, according to predefined standards and/or preferences.

2.4.5 Cohesion and Coupling

Cohesion and coupling are two terms often used in software design. These are important concepts to keep in mind when aiming to develop high quality software.

Cohesion refers to the level to which the components inside a class or module belong together [45]. For instance: in a Web shop application, a module with functions for product management, such as "setPrice" or "setName", should not implement functions for printing product information to the user. Such module would have low degree of cohesion, because it would have more responsibilities than managing the products.

Generally, well-designed software will have high degree of cohesion [45]. This means the code would be divided into modules/classes with well-defined responsibilities and they would stick to said responsibilities. This makes the code easier to understand and maintain.

Coupling on the other hand refers to the degree to which different modules depend on each other, and how much they "know" about each other's implementation [45]. In practice, it has a significant impact when introducing changes. Modules with a high degree of coupling will require changes to be made in all modules involved. By contrast, loosely coupled modules make it possible to change the internals of a module without the need to modify the modules depending on it.

Ideally, developers strive for low or loose coupling in their modules. This enhances the code quality by significantly increasing code maintainability.

2.4.6 Refactoring

Refactoring is the practice of changing source code without modifying external behaviour of the module [46]. It can involve restructuring code, renaming variables and functions or methods. In many cases, refactoring can resolve code smell [47]. Refactoring often results in simplified code, reduced duplicate code, and use of best practices.

2.4.7 Code Reviews

Code review is a process in which software developers review each others work [48]. These reviews can be incorporated into the SDLC, becoming a part of the day to day work. Code reviews can decrease the chance of bugs making it into production, thus enhance security and additionally improve overall code quality. It is especially valuable if developers review code in the area of their expertise. Additionally, such practice facilitates for easy sharing of knowledge.

2.4.8 Documentation

Documentation of code is an essential part of development. It allows for maintainability and lets other developers understand what pieces of code do, and how it can be used without needing to analyze the code itself. Documentation syntax and standardization is dependent on the programming language. Inline documentation can be used to explain the code in more details, which is helpful for understanding complex parts of a function.

2.4.9 Software Testing

Software testing is a term used to describe various processes used to evaluate the software product's fitness for purpose, robustness, and performance [49]. Many testing strategies can be automated, thereby incorporated into the SDLC via CI/CD. Testing can help improve the software and prevent bugs from making it into released versions. It can also alert about regressions in the code, meaning changes won't break previously working code. For example, by changing the output of a function to something unexpected.

There are many strategies for software testing. The strategies especially relevant for this thesis are discussed further in this section.

Unit Testing is the process of testing units of code, which are the smallest functional pieces of the program [50]. Unit testing allows the developers to test parts of the code in isolation, pinpointing locations of problems.

Unit tests use different strategies to evaluate different test cases. Examples are logic checks, boundary checks, or error handling [50].

Further, unit tests can be divided into positive and negative tests. Positive test cases are cases where the tester expects a unit to succeed. Conversely, negative test cases are cases where the tester expects the unit to fail, by for instance returning or throwing an error or exception.

Unit tests must ideally be fast [51]. They should not send requests to external services, communicate with the database, or interact with the file system. These operations simply take too much time and they test more than just the unit. However, it is still possible to unit test code that depends on these operations, in such cases test doubles (mocks/stubs) may be used. Test doubles "pretend" to be the external dependencies by acting as the real things. They produce expected outputs without actually using the network, file system, or other time consuming operations.

Integration testing is used to verify that different components of the software product work together as expected [49]. Such testing can evaluate whether bigger parts of the product, for example the main REST API service and the database, cooperate as expected.

Functional testing aims to evaluate functionality of a software product based on the functional requirements [52]. It is usually done by providing the program with inputs reflecting different use scenarios, and comparing outputs to expected data [49]. This type of testing is often referred to as "black-box testing", implying that the implementation details are not relevant [52].

Usability testing, is often referred to as user testing. Its purpose is to evaluate the user experience [53]. It can identify flaws in the design, but additionally it can lead to ideas for improvement or new features.

The process of usability testing involves two parties: the test user, called the participant, and the test facilitator [53]. The facilitator provides the participant with tasks to perform on a wireframe or the application. The facilitator will then observe the participant, noting anything they get stuck on, or if they provide feedback. By nature, usability tests cannot be automated.

Exploratory testing is another form of manual testing. In this testing strategy the tester is free to try things out and attempt to break things in the application [51] [54]. They are encouraged to test different scenarios, and find problems that slipped through the existing automated tests.

2.4.10 Conceptual Frameworks for Software Testing

There are ideas and metaphors around the way programmers structure their test suites and which types of tests are prioritized. These can act as guidelines for how much effort should go into the different testing strategies.

The test pyramid is a concept most developers are familiar with. The idea is that unit tests are at the bottom and form the base of the pyramid, integration/service tests are in the middle part, and end-to-end tests or manual tests are the tip [51]. The size of the pyramid's chunk in a given group reflects the amount of tests/efforts dedicated to the given test group. The placement in the pyramid reflects the test speed as well as the cost of running the tests. The higher the test is, the slower it is to run and the more expensive it is [55]. Here, cost refers to the computational resources required, but it can also refer to the actual monetary cost (for example, the cost associated with manual testing).

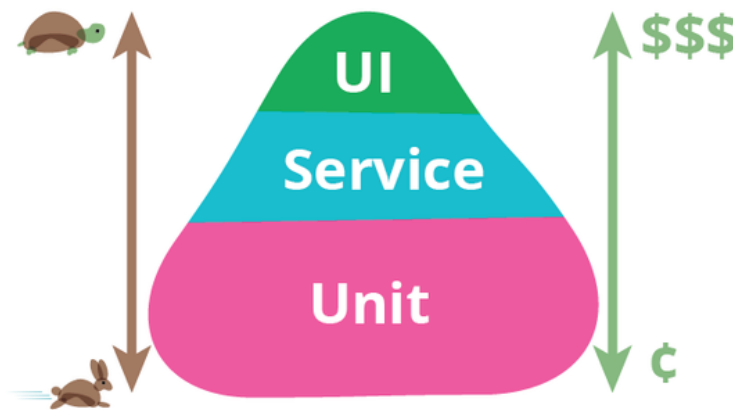


Figure 2.4.1: The test pyramid. Source: *Test Pyramid* by Martin Fowler.

Recently, a new way of thinking has been emerging; the test trophy, also called the test honeycomb [56]. The idea behind the test trophy is that static tests (such as the compiler or linting) are included as the base, with the top part being similarly structured to the test pyramid. However, the part representing the integration tests is now significantly larger in area, implying more efforts towards these types of tests.

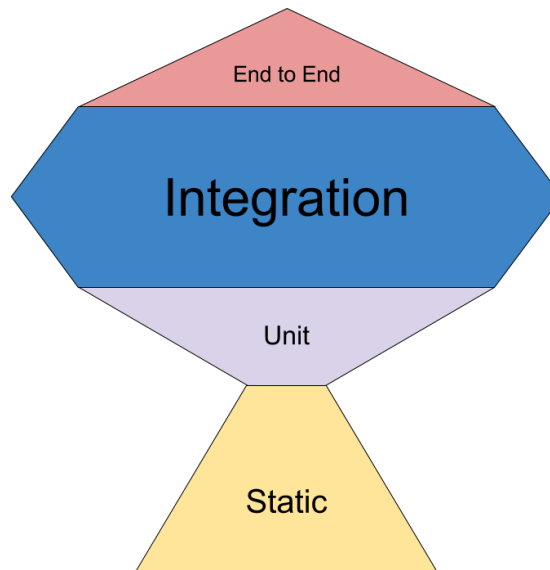


Figure 2.4.2: The test trophy. Source: *The Testing Trophy and Testing Classifications* by Kent C. Dodds.

The test trophy suggests the developers should focus more on the integration testing [56]. The differences between the pyramid and the trophy are speculated to come from blurry definitions of unit and integration testing. Tim Bray points out in his article *Testing in the Twenties* [57], in which he criticises the test trophy, that unit testing is incredibly important in lower level infrastructure, but "it's possible that some of my findings are less true once you get out of the infrastructure space."

Many integration tests can be replaced by unit tests using test doubles, thus making them faster and cheaper to run. However, this is a trade-off. During mocking, real-world cases are not tested [58] [59]. The tests lose credibility and reliability, because the mocks can obscure real-world interactions.

2.5 Web Development

2.5.1 HTTP and Network Communication

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents [60]. The HTTP protocol is the basis of a data exchange done on the Web.

Clients and servers communicate with *requests* from the client and *response* from the server [61]. These requests are GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE and PATCH [62]. HTTP usually relies on a Transmission Control Protocol/Internet Protocol (TCP/IP) connection to transfer information, but any reliable transport protocol will work [63].

Another feature of the HTTP protocol is its response codes [64]. They are informational completion responses that can tell if the request was successful, if more information is needed, or there was an error. These codes are organized in every hundred, starting from 100 up to and including 500, with each iteration being a class of codes for a specific situation.

A problem with HTTP is regarding its security, as it is by itself only able to

send the content in plain-text. Hypertext Transfer Protocol Secure (HTTPS) was created to solve this issue.

HTTPS was created in 1994 by Netscape as "HTTP over SSL" [65]. Over time Secure Sockets Layer (SSL) has turned into what is now Transport Layer Security (TLS). HTTPS allows for privacy and security by using certification, authentication and end-to-end encryption to secure the connection between the server and the client.

2.5.2 HTML, CSS and JS

These three are the pillars of Web development, which regards the frontend of the Web application. "HTML is the most basic building block for the Web" [66]. It allows for structuring HTML elements on the page, such as images, paragraphs, and buttons. CSS describes the webpage's appearance, such as font size and background color. JavaScript (JS) describes its behavior, such as changing background color when clicking on a button.

HTML documents are usually delivered using HTTP from a webserver or by e-mail. They can be viewed in Web browsers and graphical e-mail clients, amongst others [67].

"Each web browser uses a layout engine to render web pages, and support for CSS functionality is not consistent between them. The adoption of new functionality in CSS can be hindered by a lack of support in major browsers" [68].

This means that Web developers need to evaluate how to achieve the desired results while ensuring an optimal experience for the majority of users.

One specific HTML element is the inline frame (iframe). "The <iframe> HTML element represents a nested browsing context, embedding another HTML page into the current one" [69]. For example, it could be used when it is not effective or possible to have the content be part of the top HTML page itself.

"99% of websites use JavaScript on the client side for webpage behavior" [70]. JS enable webpages to be completely responsive and dynamic. However, some users elect to disable JS for various reasons, such as network performance issues, avoiding ads, and improving privacy. Therefore, Web developers must also consider how a webpage functions without JS.

2.5.3 Web Frameworks

One limitation of vanilla HTML is that it is not a programming language. Therefore, it's widely popular within Web development to utilize frameworks that can dynamically render HTML. Frameworks also make it easier for developers to focus on application-specific features instead of low-level details, by abstracting the implementation.

"The use of frameworks enhances code organization, scalability, and maintainability" [71]. Many frameworks promotes code reuse and templating, reducing the amount of necessary code.

Some of the popular Web frameworks of 2024 are: React, Django, Ruby on Rails, Spring Boot, and Laravel [71]. Certain frameworks also allow for writing Web applications with other programming languages entirely, without needing to write HTML, CSS, or JS.

2.5.4 CSS Frameworks

CSS out of the box makes it possible to apply styles directly on the HTML element using the "style" property, or by applying a class and writing CSS for the class. However, in a large project this becomes a lot of code to keep up with, and the lack of organization can also cause accidental code duplication.

CSS frameworks, such as Bootstrap and Tailwind CSS, aid in writing more efficient and organised code. Some frameworks, such as Bootstrap, are component based [72]. This means they provide ready-made components such as buttons, alerts, and navigation bars. Other frameworks, like Tailwind CSS, are utility-first based [73]. They provide a set of low-level utility classes that can be used to apply specific styling.

2.5.5 Server-Side Rendering

Server-side Rendering (SSR) means to render HTML on the webserver, as opposed to Client-side Rendering (CSR) which renders HTML in the browser [74]. SSR relies less on JS being enabled than CSR in order to display the page. A SSR webpage can still rely on JS on the client-side for interactivity after the initial page load.

The most significant benefit of SSR is the page load speed. "By offloading some of the rendering tasks to the server, you can reduce the amount of work the user's browser needs to do, resulting in faster initial load times and a smoother user experience" [75].

There are also drawbacks to SSR. A webpage still relies on JS on the client-side to be interactive. Some frameworks can send the JS to the client only when needed in order to improve performance [74]. However, these technologies are notoriously complex to program efficiently.

2.5.6 REST API

REST API, also known as RESTful API or REST, is an architectural style designed for the Web [76]. It contains a set of rules for how network-based applications can share data [77]. The intention is to decouple the client and the server.

A benefit is that it is simple and standardized [76]. The developer does not have to worry about how to format the data or the request. Other benefits is that it is stateless, has great scalability, and allows for caching.

2.6 Programming Paradigms

Programming paradigms are a collective of high-level principles and models which programming languages build upon [78]. The highest level of paradigms are imperative programming and declarative programming. These then consist of other programming paradigms like object-oriented programming and functional programming. Often programming languages cannot be purely defined in a paradigm and are then what is called a multi-paradigm programming language [79].

2.6.1 Imperative Programming

The imperative programming paradigm focuses on describing how to get to a result by running commands which alter the state of the program [78]. It is important then to understand that since the computer interprets commands, the order of these commands matter. An example of imperative programming would be object-oriented programming.

2.6.2 Declarative Programming

The declarative programming paradigm is informally described as, "what is to be computed, but not necessarily how it is to be computed. Equivalently, in the terminology of Kowalski's equation: $\text{algorithm} = \text{logic} + \text{control}$, it involves stating the logic of an algorithm, but not necessarily the control" [80]. An example of declarative programming would be SQL.

2.6.3 Object-Oriented Programming

Object-oriented programming (OOP) is an imperative programming paradigm [78]. In OOP languages, a digital model represents an object in the real world with properties and behaviors. This model consists of multiple objects in the chosen limited domain to model. Further, the object can be categorized into classes [81].

In recent years, quite a few OOP languages have started including concepts from functional programming, often called lambda functions [81], making these languages multi-paradigm languages.

2.6.4 Functional Programming

Functional programming is a subsection of the declarative programming paradigm in which the program itself consists of gluing together functions [82]. A programming function works quite similarly to a mathematical function, in which it usually takes in an input and returns a value. Functional programming works by *declaring* what you want by *gluing* functions together.

2.6.5 Multi-paradigm Programming

Multi-paradigm programming language is a programming language that implements two or more paradigms [79]. The purpose of a language implementing features from multiple paradigms is so that the programmer has the ability to choose the best tool for the job. One paradigm might be more efficient than another paradigm for specific problems.

2.7 Database Management System

A Database Management System (DBMS) is a system for managing the database, as well as allow users and applications to interact with the database [83]. It

includes the database itself, user management, as well as the means to manipulate the data and the database's structure.

2.7.1 Relational Databases

A relational database uses tables to organize the data [84] [85]. Tables represent data entities, which are logical groupings of attributes. This way, related data is grouped together and the attributes are the table's columns. Each instance of the entity type is a row in the table, often referred to as a record.

Relational databases use keys to both uniquely identify rows within a table, and to express relations between different tables (entity types) [84] [85]. A primary key is simply a unique entity identifier, which can be either a single property (column) in a table, or it can be constructed from multiple columns, becoming a composite key [84]. A foreign key is used to "link" entities together by referring to a different entity type from a table. Foreign keys can be used to express one-to-one, one-to-many and many-to-many relationships between entities.

2.7.2 SQL

SQL is a flexible, generally declarative language that provides all the basic CRUD operations for data and database manipulation [86] [87]. It can be used to structure more complex queries. It can combine data from multiple tables (join), select data conditionally, aggregate it, sort it, and group it. When it comes to the database structuring, it can be used to define constraints, which increases data integrity and validity.

SQL can be used to implement parts of the application's business logic, leveraging its flexibility, and efficiency [88]. This way the database system is no longer just a data storage. This approach can impact the maintainability and testability of the code, but it is still viable in cases where all data comes from SQL sources and developers are familiar with SQL.

2.7.3 Database Normalization

Generally, relational databases need to be normalized, in order to reduce data redundancy and increase data integrity [89]. However, in some specific cases, it may be beneficial to denormalize the database, in effort to increase performance [90].

A good example of it is *Instagram* denormalizing the "like" counters to reduce resources needed to retrieve this data [91]. A simple "select" statement for a single value is vastly more efficient than a "select" statement that has to count many rows. This greatly reduced the load on *Instagram*'s servers, which was previously an issue when popular celebrities published new posts.

2.7.4 Seeding

Database seeding is the process of adding dummy or initial data to the database [92]. It is common to populate the database using SQL scripts in the development environment, or when initially setting up the application.

2.7.5 Schema Migrations

Database schema can be migrated from one state to another using change based migrations [93]. The migration files are SQL scripts containing statements to create new structures or modify existing ones. The scripts need to be executed in order, and the initial database state needs to be known.

2.8 Security

Several considerations must be made when developing applications for the Web. Availability over the Internet opens the application up to possible threats, such as: potentially leaking sensitive information, malicious actors misleading the end-users, or system downtime.

There are multiple technologies and techniques that are commonly used to reduce the risk associated with Web applications.

2.8.1 Cryptography

To protect users' private data when using our app, cryptography becomes an important concept. It involves the use of asymmetric encryption algorithms where a pair of keys, one public and one private, are generated [94]. The public key is freely distributed, while the private key is kept secret. Messages encrypted with the public key can only be decrypted with the corresponding private key, ensuring confidentiality and authenticity.

2.8.2 Authentication

Authentication is a fundamental part of ensuring the security of Web applications [95]. It verifies the identity of users or systems attempting to access resources or services. Common authentication mechanisms include passwords, biometrics, and cryptographic tokens.

In the context of Web applications, authentication often involves verifying user credentials, such as usernames and passwords, before granting access to protected resources, such as uploading resources [95].

2.8.3 Common Vulnerabilities

One common vulnerability is SQL injection, which is the process of making an SQL database execute arbitrary code via unsanitized user input [96]. This can both return data that it is not supposed to, or even ruin a database by forcing it to drop tables.

One way to counter SQL injection is by using prepared statements. It is SQL that is prepared in advance, usually with placeholder values [97]. Later, when the statement is used, the parameter values is passed to the statement and the SQL is run against the DBMS.

Another vulnerability is cross-site scripting, also known as XSS [98]. It is the method of injecting client side scripts into websites and making them run on other peoples computers. This essentially allows outside actors to force actions on other

computers. Some ways to prevent XSS include: sanitizing input, encode output data, and restricting which resources can be loaded [99].

2.9 Virtualization

A virtual machine is a virtualization or emulation of a computer system [100]. It gives virtual versions of the CPU, memory, storage and sometimes the GPU. Sometimes it does that by simply passing through the physical components to the virtual machine, or by specifically allocating it a smaller section of the component.

Containerization offers a lighter-weight alternative to virtualization [101]. Instead of emulating an entire machine, containers package an application along with its essential dependencies into a portable, self-contained unit. This container can then run consistently across different systems and environments.

Key characteristics of containers [101]:

- **Operating System Sharing:** Containers share the host machine’s operating system kernel. This makes them lightweight and efficient because they don’t need to include a full operating system.
- **Faster Startup Times:** Containers eliminate the overhead of booting an entire operating system, resulting in significantly faster startup speeds compared to virtual machines.
- **Enhanced Portability:** The consistent packaging approach of containers ensures they can easily move between environments (development, test, production) without compatibility headaches.

2.10 Artificial Intelligence

More and more media outlets are starting to embrace AI as one of the tools at their disposal [102]. It allows them to summarize articles, generate images, or help writing articles. The main AI tool used in this project is a Large Language Model (LLM).

An LLM is a type of neural network, specifically a transformer model [103]. It processes input data in a sequential manner, capturing contextual dependencies and patterns within the data. These models are trained on vast amounts of text data generated by users of the Web and published literary material, enabling them to generate coherent and contextually relevant responses to given prompts or queries [103]. A notable example of such a model is the LLM, ChatGPT.

3.1 Team and Project

3.1.1 Project Methodology

We decided on following the agile development method, as this is the one we have received training in, and we saw that it fit the project. This also contributed greatly to the decision of using a SCRUM-like approach. This entails using sprints, regular with the stakeholder, 5-minutes-maximum stand-up meetings, sprint reviews, and sprint retrospectives. A benefit was that NTNU provided Jira and Confluence, which are tools to assist with agile development.

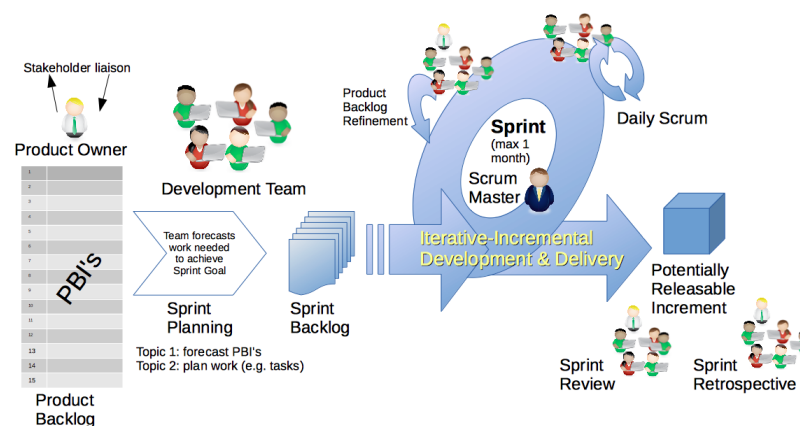


Figure 3.1.1: Scrum framework by Dr Ian Mitchell. Published under CC0 1.0

3.1.2 Jira

We used Jira as an issue tracker and planner. We used it actively to manage sprints, track tasks and their progress. Additionally, we logged our work time.

3.1.3 Confluence

We chose to use Confluence as a wiki for the project, effectively using it as a notebook. We actively used it to collaborate on documentation in real time, such

as meeting notes or sprint retrospectives. It integrates nicely with Jira.

3.1.4 Discord

We used Discord as a communication platform. It allowed us to easily discuss and notify each other of important matters. We shared resources for developing, and discussed meeting times. Since most of the team already had experience with the tool, it was the best choice.

3.2 Design and Prototype

Before starting to make the application itself, we wanted to plan how it would look.

3.2.1 Models and Diagrams

”Models and diagrams are representations of a real-world application. Models provide an abstract view of the system, while different diagrams provide concrete representations of the system” [104].

The team was provided with project specifications by the product owner. A way to get an easy overview of the requirements was by using making UML models and diagrams. We created these using *LucidChart*, *Excalidraw*, and drawing on paper.

Use-case diagrams and user stories were made to display how different types of users would use the system and the functionality that needed to be implemented to fulfill these requirements.

We also used diagrams to make simple sketches of the system architecture in order to plan ahead and compare different solutions. We made a database schema to view different possible solutions and if the logic seemed sound.

3.2.2 Figma

We decided to use Figma as the wireframing tool for our project. We needed a tool that had good real-time collaboration and at no cost to us [105]. An additional benefit was being able to use the same tool later on in the development process for making high-fidelity designs. The team also had experience with the tool from previous projects, making the design process quicker.

3.3 Environment and Development


3.3.1 Git


For version control, we used Git. It allowed us to work in parallel, utilizing branching.

For structuring of the commits themselves and their messages, we used Conventional Commits. Additionally, we frequently used emojis in the commit messages, to give an instant idea of what kind of changes were committed. Use of emojis

was made easy and encouraged by a Conventional Commits extension we used for VSCode.

Example commit messages using Conventional Commits and emojis:

”fix:  typo quizzes to quizzes”

- ”fix:” signifying the commit is a fix
- ”” - pencil emoji, together with the ”fix:” prefix indicating ”fix typo”
- and the commit message itself, describing the change

3.3.2 GitHub

We used a remote Git repository on GitHub to manage our source code centrally and synchronize the changes. We used Pull Requests (PRs) to exercise peer code reviews, further described in section 3.5.2. Additionally we utilized GitHub actions to automatically run CI/CD pipelines on specific events.

3.3.3 Code Editor

We primarily used Visual Studio Code as our code editor, as we are familiar with it and it is highly extendable. By the use of plugins relevant for the technologies used, it is made into an Integrated Development Environment (IDE). Additionally it connects directly to WSL, allowing us to work in a Linux environment.

Additionally, one of the team members used Neovim as the code editor of choice. Paired with an Language Server Protocol (LSP) and other extensions, this terminal based text editor can be as effective as VSCode, while staying less resource intensive.

3.3.4 SSH

We used SSH to securely connect to and execute commands on the server assigned to us by NTNU. Both manually, in order to setup and tweak the production environment, and automatically as part of the CI/CD.

3.3.5 WSL

As we had a variety of operating systems on our workstations (one person with Arch Linux and three with Windows 11), the Windows users utilized Windows Subsystem for Linux (WSL) in order to unify our development experience. This vastly simplified choice of utilities and development of shell scripts, as we only had to make sure these worked on Linux. Developing in a Linux environment also meant we would be working in an environment most similar to the application’s production environment. As an added benefit, WSL integrates seamlessly with VSCode.

3.3.6 Docker

In order to simplify the deployment and have a consistent production environment for our application, we used Docker [106]. We used it to build an image and run the application as a lightweight container. It was an easy choice to make, since we had earlier experience with it,

Additionally, we used Docker compose to specify and manage any services needed for the application or the environment. Docker made it easy to run services and utilities that otherwise would require installation on the host.

3.3.7 Adminer

We used Adminer to inspect and manipulate the database in the development environment. We were familiar with the tool from earlier projects and had experience with troubleshooting the database with it. It is simply added to the Docker compose file, becoming available in the wanted environment.

3.3.8 Secret Management

To avoid hard-coding secrets and configurations, we used environmental variables. This way the application could get them dynamically from the environment as needed. We used a ".env" file to store the values persistently. This file is excluded from version control.

3.4 Solutions

3.4.1 Go

As the main programming language for the project, we chose Go. We chose it specifically for its simplicity, static typing, errors as return values (instead of exception try/catch) and incredibly fast compilation. It is performant, garbage collected and comes with a rich standard library [107]. A module from the standard library worth mentioning in this section is "context". It provides a "context.Context" type commonly used in Go APIs to send cancellation signals, create deadlines and store data as key-value pairs in a request context [108].

3.4.2 Echo

We chose to use Echo [109] as a Web framework for its advanced route matching and flexibility. It supports centralized HTTP error handling, which we leveraged to establish common error formats. Additionally, its API is close to Go standard library's HTTP module.

3.4.3 Air

In development we used Air [110] to automatically rebuild and restart the application as soon as changes in source files were saved. This allowed us to stay productive and focus on the programming.

3.4.4 Templ

We used Templ [111] as the templating language. It is built for Go and allows for the use of Go concepts in its template files. Templ files are used to generate Go code, making their APIs work directly in our Go modules. Templ is built with security in mind, and as a preventive measure against JavaScript injections it limits the use of variables and expressions in certain HTML attributes [112].

3.4.5 HTMX

We used Hypertext markup extensions (HTMX) [113] (which is a JavaScript library) to easily send HTTP requests from the browser. We utilized its HTML element targeting and swapping capabilities to simply and effectively replace parts of the web page with HTML elements returned by the server. This way we avoid unnecessary full website reloads.

3.4.6 JavaScript

For this project, JavaScript is used as a scripting language for the Web browser. We use it to implement client sided interactions. On top of the default JS, we include some additional libraries. These are *SortableJS* [114] for simple implementation of draggable elements, and *Odometer* [115] to animate number transitions.

3.4.7 Tailwind CSS

For styling we used Tailwind CSS. It is a powerful and flexible CSS framework, developed with mobile-first in mind, making it perfect for our needs [116]. Additionally, it makes it easy to create responsive designs. With Tailwind, all styling information is attached directly to the elements. This simplified making changes and encouraged us to create reusable components.

3.4.8 GNU Make

We use GNU Make [117] as a utility to simplify execution of commands and shell scripts frequently used in the development environment. These are specified in the "makefile." It allowed us to specify targets like "live-reload" or "build", whilst hiding all complexity around the actual commands required to achieve these goals.

3.4.9 Shell Scripts

We use shell scripts to automate tasks and simplify more complex commands, which are not practical to put in the makefile. Bash is used as the scripting language in all the shell scripts in the project.

3.4.10 PostgreSQL

We chose PostgreSQL as the DBMS for the application, since we were familiar with it and knew its capabilities. It is performant, robust and feature rich, making it

flexible and a good choice for any project [118]. We specify PostgreSQL as one of the services in the Docker compose.

3.4.11 MinIO

We used MinIO as a storage solutions for images. It is compatible with S3, which is the most common API for object storage [119]. This way we do not rely on specific implementation, and the object store service can be easily replaced. We utilized Docker compose to run this service.

3.4.12 Caddy

In production environment we use Caddy [120] Web server as a reverse proxy. It sets up HTTPS certificates automatically [120] [121], meaning we do not have to manage them manually. Additionally its configuration is trivial (see code 3.1), and it is included as a service in the Docker Compose file.

```
1  example.com {  
2      reverse_proxy: 192.168.0.49  
3  }
```

Code 3.1: Caddy example config for a reverse proxy

3.4.13 AI

GitHub Copilot [122] was used as an "AI pair programmer". It was mainly used as a smart auto-complete, to generate: repetitive code, parts of code documentation and suggest potential solutions.

ChatGPT API was utilized to generate question suggestions for quizzes. As it is an LLM [123], it is capable of analyzing provided articles and producing relevant questions.

3.5 Quality Assurance

Ensuring that a project has a high quality is vital for customer and user satisfaction, as well as maintainability of the project.

3.5.1 Google Lighthouse

The team used Lighthouse to measure performance, accessibility and use of best practices in our application [124]. This tool is built into the developer tools in *Google Chrome*, making it easily accessible and simple to run. We used it actively to measure performance and make sure the HTML elements used are semantic, and they provide enough information for technologies such as screen readers.

3.5.2 Code Reviews

Our team used code review practice in combination with approval process for PRs onto the development branch. This way all changes and features would be peer reviewed and quality assured before making it onto our stable development branch. From there, code would soon be merged into the main branch, making it into production.

3.5.3 Sonarlint

SonarLint is a linter we used to enforce rules regarding code quality. It helped us to uphold highly maintainable code by analyzing it and pointing out some potential problems.

3.5.4 Unit Tests

For unit testing, we used Go's standard library unit testing module and tooling, because they come directly with the language. The test are placed in the same modules as the tested code, this way test functions have access to the tested module's private members. The test files are placed beside normal module files, and the filenames end with `”_test.go”`, following Go standards [125].

3.5.5 Integration Tests

In order to test the applications integration with external services, such as the database we could either use mocks or real services. There are advantages and disadvantages associated with both approaches, discussed earlier in section 2.4.9. We decided to avoid using mocks and aim for integration testing most closely resembling real interactions.

To implement integration tests, we utilized *Testcontainers*, which is a Go package [126]. It allows us to manage containerized services in a testing environment. Tests can separately set up resources needed and clean-up when they are done. Tests are isolated and repeatable.

In order to further simplify the development of integration tests, we utilize *Testify* package for Go. It provides a simplified testing API and makes it easy to create test suites, with setup and tear down functions [127].

3.5.6 End-to-end Tests

We utilized Bruno [128] for end-to-end API testing. We use Bruno CLI to automatically run the tests without need for a GUI.

This testing approach requires the application and all necessary services to be running in the test environment, making it relatively slow (in comparison to unit and integration testing).

We do have shell scripts to simplify the running of these tests. Additionally we have a separate, simple Go Web server with a few utility endpoints. They are used by the Bruno client to assign user sessions with different roles.

3.5.7 Usability Testing

For usability testing we had a manual process where a team member (or members) would give a task to a potential user. The test facilitator(s) would then observe the user and note anything of interest, such as: user unintentionally clicking a button multiple times, user being clearly confused or any user comments.

4.1 Administrative Process

The administrative process describes the results regarding collecting information during the different phases, how the development group structured itself, and how we managed our work hours.

4.1.1 Meetings

During the project, the team had regular meetings with the stakeholders every other week. In these meetings, we discussed the project requirements and our ideas regarding solutions. Meeting notes were written in Confluence.

Since the project description had no requirements for which technologies to use, we provided a list of our suggestions and the reasons we wanted to use them. We also asked if it would be preferable if we used the same technologies as they already used, but they were happy to let us choose and approved our suggestions.

In the information collection phase, we created user stories (see them in appendix C) based on the project description and additional clarification given during the early meetings. Some ideas stayed throughout the project, while other ideas were scrapped as the requirements changed. Some examples of this are discussed in section 4.4.1.

We got insight into how we could integrate our solution with their existing website and mobile application. Some limitations were introduced for security reasons. We could not be given access to their login API, so users needed different accounts for playing quizzes than reading articles. That creates a higher barrier of entry, since it's one more step to do. It was also a challenge figuring out how to get protected data to our server. Sunnmøreposten's articles are not public, as some of them are behind a paywall. In the end, we did not get access to their API and we were sent a data dump of articles as JSON files to use for AI generation of questions.

In the design phase, we presented wireframes for the application, for both the player and the administrator pages. After we got these approved, we created high-fidelity designs of the application. Sunnmøreposten was particularly helpful in this process, as their Head of Digital Development also has a background in graphic design and was able to give very helpful and constructive feedback.

During the development phase, we performed usability tests with different demographic groups at Sunnmøreposten. We tested on users with more and less technological experience of different ages. This process is further elaborated in section 4.5.6.

The team also had meetings with the advisor every other week. She helped the group stay on track and gave us some valuable advice on making a use-case diagram (see appendix B), database schema (see section 4.3.2), and domain model (see section 4.3.3).

4.1.2 Project Management

The team used agile methodology, with sprints spanning two weeks each. In each sprint, we had meetings with the advisor and project stakeholder.

At the beginning of each sprint, we planned ahead which features and issues we wanted completed and how polished they needed to be. In earlier sprints, the goal was to create prototypes, and later on the goals were to create finished features.

Sprints and issues were tracked in Jira. We connected our issues to user stories, and each issue could also have sub-tasks. This was helpful if multiple people would work on the same task, or if it was a large and complex issue.

Features were made on different branches in GitHub. Once a feature was functional, a pull request would be opened to merge it into the "dev" branch. When each iteration of the application was completed, "dev" was merged into "main", which was then deployed to production.

We held stand-up meetings every day before lunch to keep each other updated on the tasks that were done.

4.1.3 Time Management

The project requirements and expectations were finished within the given time, but some wished-for features were not prioritized due to the time constraint.

4.1.4 CI/CD

For Continuous integration / Continuous delivery we used GitHub actions. On every push or pull request to the "dev" branch, the "Run unit tests" and "Run integration tests" jobs would run to check if the code compiled, and verified that all the tests passed. On the "main" branch, the same tests would run, in addition to a Docker workflow, that published a new Docker image and then pushed it to the production machine. It did that by using SSH to tunnel into the production machine and then ran an update script.

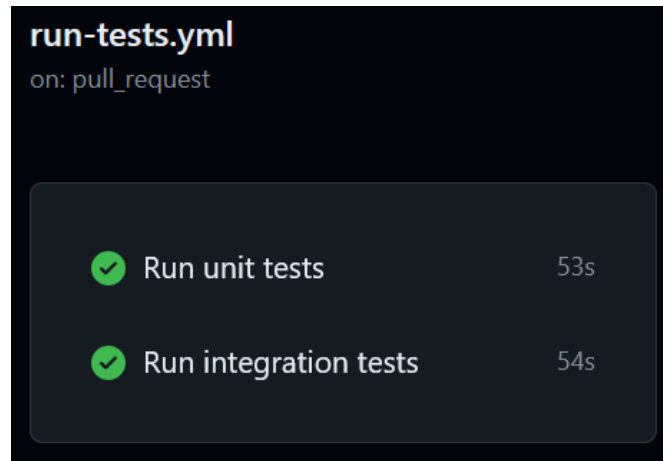


Figure 4.1.1: Successful runs of unit and integration test jobs on GitHub actions

4.2 Features

The features of the finished project include all the original project requirements. Other features have been included based on requests from the stakeholder and our own suggestions that the stakeholder agreed with.

4.2.1 Account Creation

A user can create an account for playing Nyhetsjeger by authenticating using a *Google* account. After logging in for the first time, it is possible to choose a username and whether the user wants to opt-in to leaderboards. If they opt-in, their username will be visible on leaderboards and administrators can view their contact information to distribute prizes.

4.2.2 Usernames

The usernames are comprised of a list of adjectives and nouns, where each possible combination is a username. With SMP having 73 400 unique daily readers [2], we figured that 90 000 unique combinations was sufficient in the beginning. This could be achieved using 300 adjectives and 300 nouns.

While the initial list of usernames was generated by the development team, it is possible for administrators to modify the usernames in the admin dashboard page. For further detail regarding username administration, see section 4.2.12.

4.2.3 Navigation Menu

In order to navigate between pages in the application, we use a navigation menu at the top of the page. The main menu is minimized on smaller screens, and can be expanded with a toggle button. On larger screens, the whole menu is always displayed.



Figure 4.2.1: The navigation menu on a larger screen.



Figure 4.2.2: The closed navigation menu on a smaller screen.



Figure 4.2.3: The opened navigation menu on a smaller screen.

On the admin panel, the menu is on the left side on larger screens, and on the top on smaller screens.

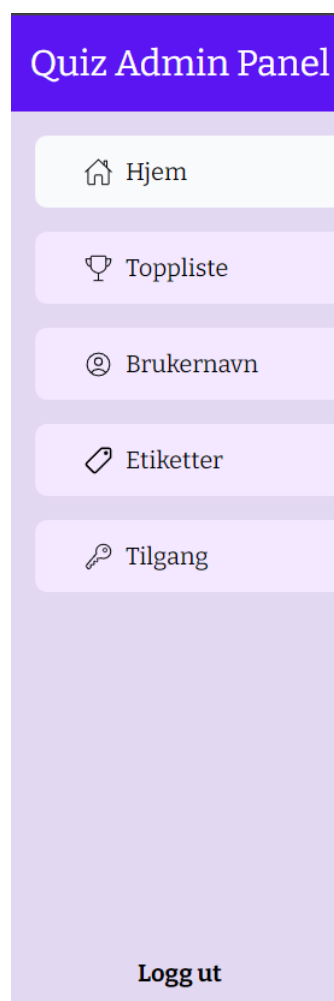


Figure 4.2.4: The admin navigation menu on desktop.

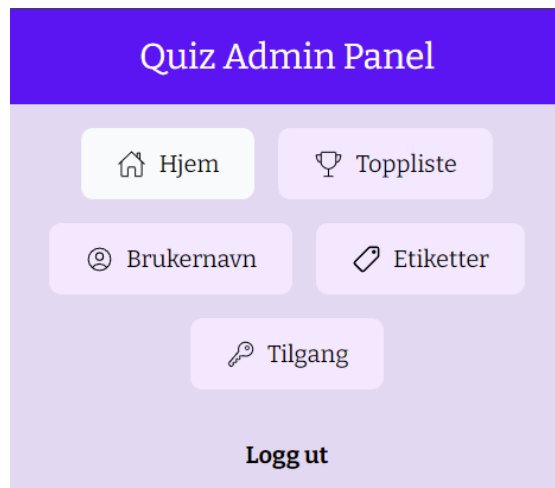


Figure 4.2.5: The admin navigation menu on mobile.

4.2.4 Play Quizzes

A quiz can be played by answering the questions that it contains. Points are given for correct answers and no points are given for an incorrect answer. The culmination of points are displayed throughout the quiz. After all the questions are answered, a summary page is shown to the player.

Each quiz preview is represented as a "card." These cards include information about the quiz, such as title, main image, active end (when the quiz will stop being active), number of questions, possible earned points, and which leaderboards the quiz applies to.



Figure 4.2.6: A quiz card for "Påske Quiz!"

When the card is hovered, a gradient shadow is displayed behind the card. When the "play" button is hovered or focused, the gradient outline on the card and the gradient color on the button changes color.



Figure 4.2.7: A quiz card for "Påske Quiz!" when the "play" button is hovered.

Each quiz contains at least one question, ideally more. A question has a time limit that counts down to zero. The question can still be answered after the time limit, but the player will be rewarded less points. The point calculation is discussed further in section 4.2.5. Each question has two to four answer alternatives. Multiple alternatives can be correct. A progress bar is displayed, showing which question the player is currently on, out of the total amount of questions.

After answering the question, the user gets feedback via the alternative buttons. A check mark or cross is displayed if it is correct or incorrect. Additionally, the distribution of user answers are also displayed, so users know if their answer was popular or not.

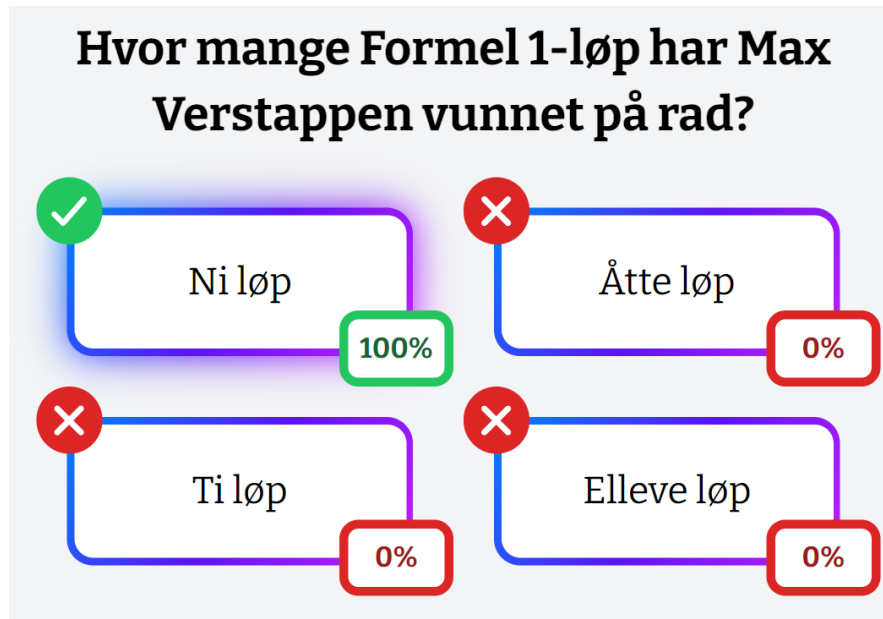


Figure 4.2.8: An answered question that was correct, where 100% of users chose the first alternative.

After answering all the questions, the player will then see the summary page, where the quiz results are displayed. The points earned from the questions are displayed, out of the total amount that could be earned. The player can see all the questions and their selected answer, and how many of the points they earned for it. A check mark is shown if the player's selected answer was correct, and a cross if it was incorrect.



Figure 4.2.9: The result of the quiz, points-wise. Three screenshots at different times, side-by-side. The circle is filled with a wave animation, gradually going from the bottom to halfway up, since the player got half of the possible points.



Figure 4.2.10: The quiz summary page, showing the results of the quiz.

If there were any questions based on articles, an additional page linking to these external articles is displayed.

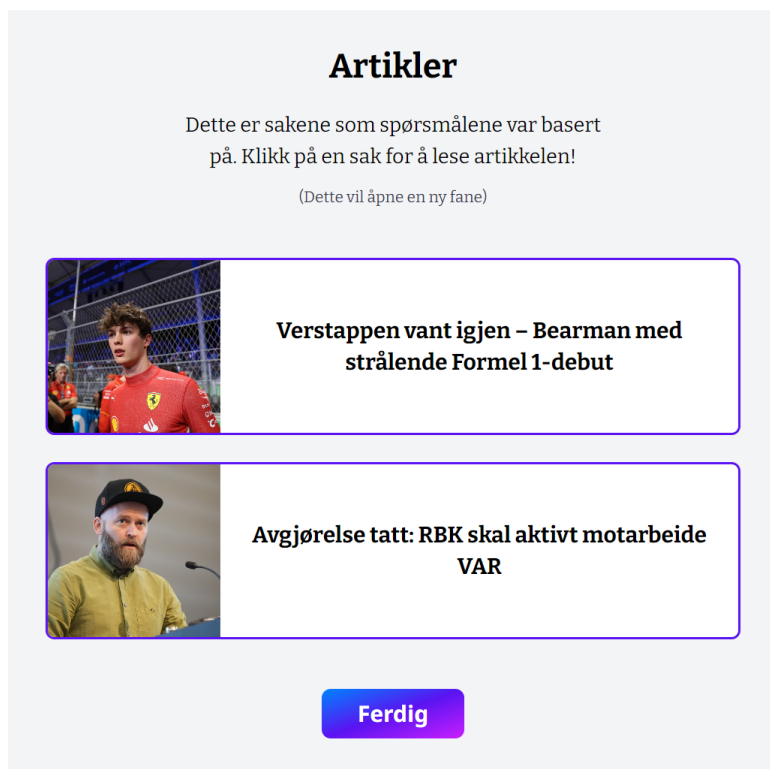


Figure 4.2.11: The list of articles that the questions were based on.

4.2.5 Point System

Players are rewarded with points for correctly answering questions in quizzes. These points are used to assess the player’s performance and place them in rankings. Throughout the project we iterated upon the point system and adjusted it to the stakeholder’s desires. This process is described in more detail in section 5.2.6.

In the final implementation of the point system, the points reward is reduced linearly with time used to answer a question. There is however a grace period, so it is still possible to achieve full score if the user answers within that time window. The reward is reduced until it reaches the minimum number of points at the question’s time limit. This is illustrated in figure 4.2.12, the question configuration is $points_{max} = 100$, $time_{limit} = 30$. The $time_{grace} = 3$ and $points_{min} = \frac{1}{5}points_{max}$. All $time$ values are in seconds.

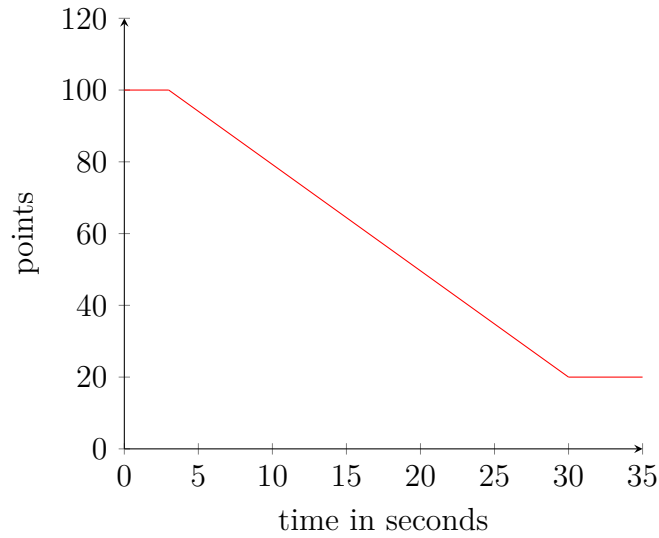


Figure 4.2.12: Final iteration of the point system.

The sloped part of the line in figure 4.2.12 can be expressed with the function below, with domain $[time_{grace}, time_{limit}]$. Any answer with t (seconds spent on answering) less than $time_{grace}$ or greater than $time_{limit}$ is rewarded with a constant number of points equal to $points_{max}$ or $points_{min}$, respectively.

$$f(t) = \frac{time_{grace} \cdot points_{min} - points_{min} \cdot t + points_{max} \cdot t - points_{max} \cdot time_{limit}}{time_{grace} - time_{limit}}$$

The function describes a line intersecting two known points, $(time_{grace}, points_{max})$ and $(time_{limit}, points_{min})$. Slope formula $m = \frac{y_2 - y_1}{x_2 - x_1}$ and a line through point $y - y_1 = m(x - x_1)$ were used to derive the final function.

This function for point calculation is implemented as a PostgreSQL function, becoming a part of the database schema. The function is "calculate_points_awarded(start_time, end_time, time_limit, max_points)". It takes in the specified parameters and returns the number of points awarded, rounded to the nearest integer.

This function is used directly in "SELECT" queries and SQL views. This is explained in more detail in section 4.3.2.

4.2.6 Guest Users

Users can play a quiz in guest mode, but points do not count towards the rankings. In guest mode, no data is stored in the database. Instead, all intermediate data needed to provide question feedback and the quiz summary is stored in the browser's local storage. The guest quiz uses public API endpoints. These endpoints expect additional data to be included with the requests and can render views identical to the ones used in the normal mode.

4.2.7 Public Leaderboard

The public leaderboards, as opposed to the admin leaderboards which are discussed further in section 4.2.12, are available for all users to see. It displays the top 10 players in a defined time period, plus the current user's position on the leaderboard. Only players that are opted-in for competitions will be displayed. Only quizzes that were played during the active time period are given points.

Each quiz can have multiple "labels" and each "label" can contain multiple quizzes. Examples for such labels are "January 2024", "All Year 2024", or "Easter 2024." These labels are used to display different leaderboards, where only the quizzes for that label contribute to the ranking.



The image shows a screenshot of a public leaderboard titled "Toppliste". The table has three columns: "Plass" (Rank), "Navn" (Name), and "Poeng" (Points). The data is as follows:

Plass	Navn	Poeng
1	nødvendig dobsonteleoskop	476
2	naturleg elefant	159
3	grøn alligator	96
4	bleik sølv	88
5	mirakuløs fiolin	86
6	frisk fennek	0

Figure 4.2.13: The public leaderboard, with six users opted-in for competition.

4.2.8 Completed Quizzes

A player can see their completed quizzes on a separate page. The quizzes are displayed similarly to unplayed quizzes, except the button says "results" instead of "play." Completed quizzes cannot be replayed, but the user can revisit the summary page containing their results.

4.2.9 Profile

A player can see their account information on the profile page. Here they can get a new random username, opt-in or opt-out of competitions, log out, or delete their account.

4.2.10 Manage Labels

During the discussion with Sunnmøreposten, we landed on the solution to categorize quizzes by label, which functions as a leaderboard. Admins can manage labels on a separate page, where they can add and remove labels, and select active status.

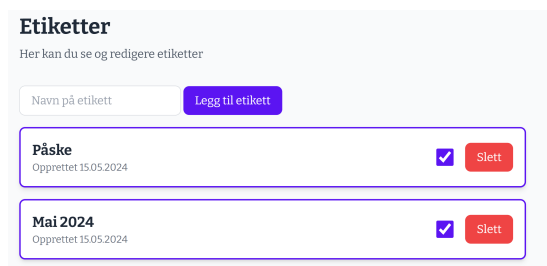


Figure 4.2.14: Edit page for labels.

Once a label is active, it can be assigned to a quiz on the quiz edit page. It will show up for users as a visible leaderboard.

4.2.11 Manage Quizzes

While not an explicit requirement in the project description, being able to manage quizzes was an important feature. Quiz administrators needed a page where they could create, edit, and delete quizzes.

When creating a new quiz, some default information is filled in automatically. It will have the title "Quiz: Uke 13", if it is week 13. Every quiz has the same default thumbnail image, and the active start date is set from the creation of the quiz to a week in the future.

Admins can add or remove links to articles. When selecting a thumbnail image, it can be done either with the URL, file upload, or suggestions from the articles.

When creating a question, an article from the list can be selected. The admin can choose to generate a question based on the article using *ChatGPT*, or write it themselves. After the question is generated with *ChatGPT*, it can still be edited. When generating questions, we had to be very specific in the prompt about how we wanted the response to be formatted, to ensure it could be parsed correctly.

Each question has a title, two to four alternatives, an optional image, points, and a time limit. The administrator has the option to change the points and time limit from the default to a range of predefined values. This is to ensure a general consistency between quizzes and take away any decision paralysis. The question alternatives can be randomly shuffled, to avoid any bias in arrangement. In the list of questions, the questions can be arranged by dragging to a position and drop it. This was implemented with the help of the library `sortable.js`.

Each quiz can be added to one or multiple "labels", which provides information on which leaderboard the quiz counts toward. A quiz is not published by default, and has to be manually published. It needs at least one question before it can be published. A published quiz will not show up for users until the active time has started. Publishing or deleting a quiz opens up a confirmation window first.

Rediger Quiz

Trykk utenfor et tekstfelt for å lagre endringen automatisk. (Dette gjelder ikke for bilder).

Velg tittel

Påske Quiz 2024!

Velg etiketter

Påske ×

Mai 2024

Velg artikler (valgfritt) ⓘ

Link **Legg til +**

1. Blir filmstjerne i år, også **×**
2. Forfang: – Folk tenker: «For noen bortskjemte drittunger» **×**
3. ID-porten til Altinn var nede – problemet synes å være løst **×**




Lag spørsmål ⓘ

1. Hva er formålet med webkameraet i Lundeura på Runde?
2. Hvilket dyr gemmer egg i påsken?
3. Hvilken tjeneste hadde innloggingsproblemer tidligere i dag?


Legg til nytt spørsmål +


Figure 4.2.15: The page for editing a quiz. Top part (1/2).


Velg forsidebilde

Med URL  Med Fil  Fra artikler 

Velg URL



Velg aktiv periode 

Fra 


Til 

Figure 4.2.16: The page for editing a quiz. Bottom part (2/2).

Rediger spørsmål ✕

Velg artikkel (valgfritt)

Avgjørelse tatt: RBK skal aktivt motarbeide VAR

Generer spørsmål med KI

Informasjon

Velg spørsmål

Hva stemte Rosenborg-medlemmene for under årsmøtet i 2023?

Fyll inn svaralternativer (mellom 2 til 4) **Velg riktig svar**

Å støtte bruk av VAR	<input type="checkbox"/>
Å avskaffe VAR	<input checked="" type="checkbox"/>
Å forbedre VAR	<input type="checkbox"/>
1816	<input type="checkbox"/>

Generer tilfeldig rekkefølge

Ekstra innstillinger

Velg bilde (valgfritt) (i)

Med URL
Med Fil
Fra artikler

Velg URL Last opp

Ingen bilde valgt enda.

Velg poeng
Velg tidsbegrensning

Avbryt
Slett
Lagre

Figure 4.2.17: The modal window for editing a question.

4.2.12 Other Admin Features

Administrators have access to specific leaderboards, where they can view any "label" (whether active or not), such as "May 2022". It shows all users that are opted-in to competitions. By clicking on the username, they can view that person's account information, such as username, e-mail, and scores across different leaderboards.

Administrators can also manage usernames by adding, editing, or removing words that can be combined to make a username. Each username is unique and

consists of an adjective and a noun. They are displayed in two separate tables, with a select amount of words at a time. They can be browsed using the arrows below the table. It is also possible to search for words, which applies to both tables. Any changes must be saved explicitly and any unsaved changes can be reset. If a deleted word has also been edited, the undo button will first affect deletion, then the edit.

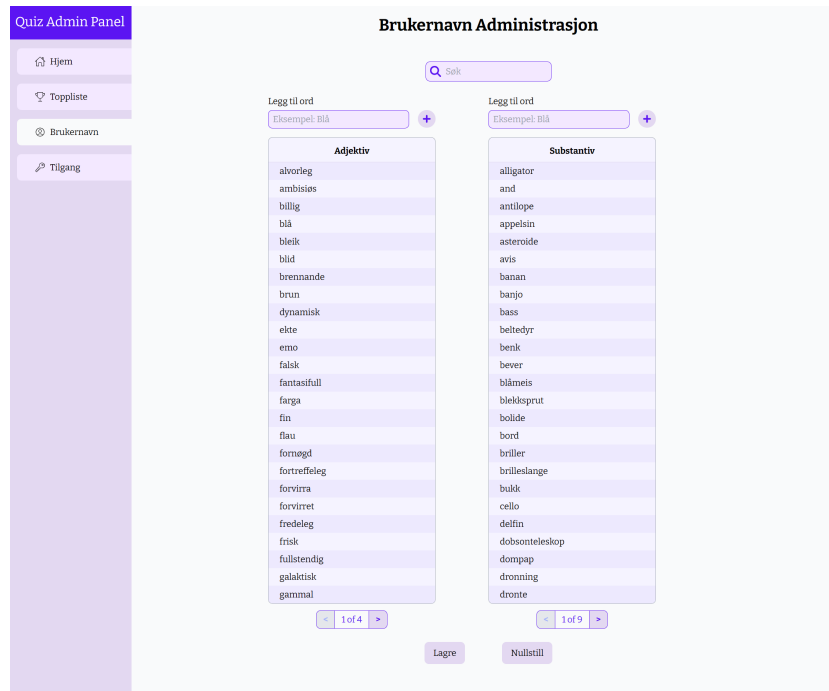


Figure 4.2.18: The username management page. The adjective table (left) and noun table (right) displays all combinations for usernames.

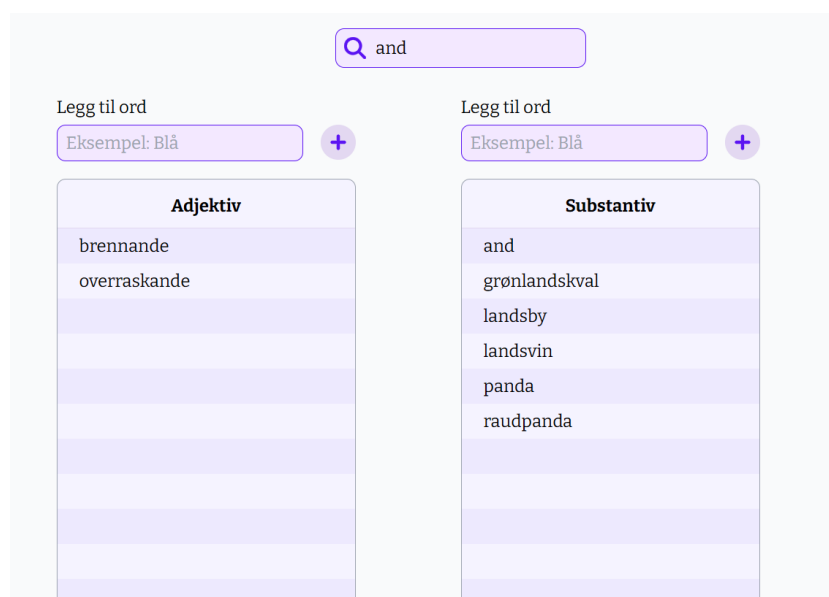


Figure 4.2.19: Searching for a word applies to both tables. The tables display any words that contain the searched text.



Figure 4.2.20: Clicking on the row displays an input for editing the word and a button to delete it.

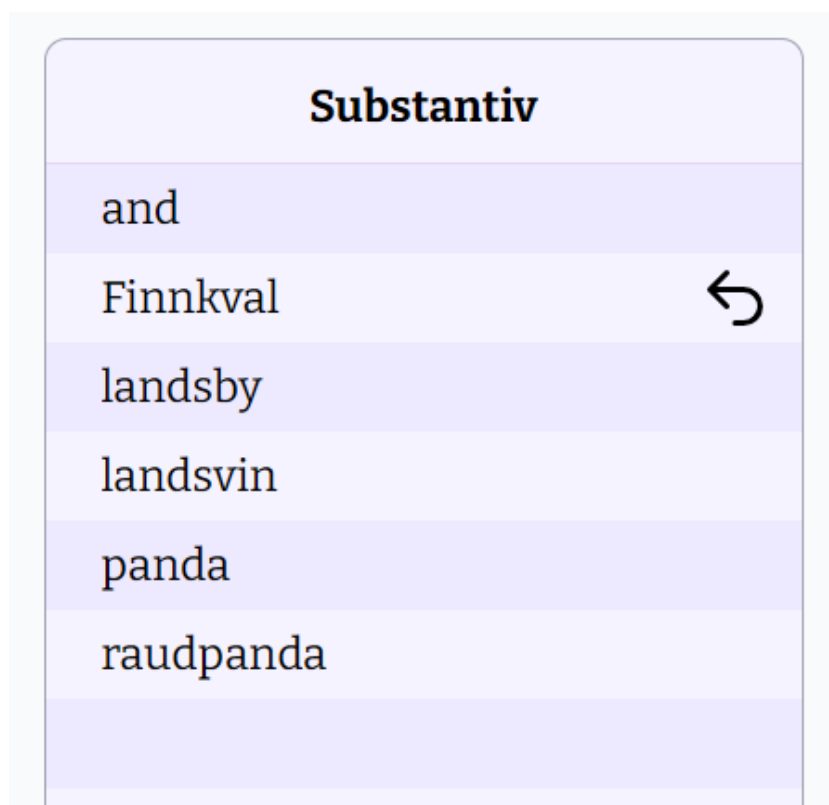
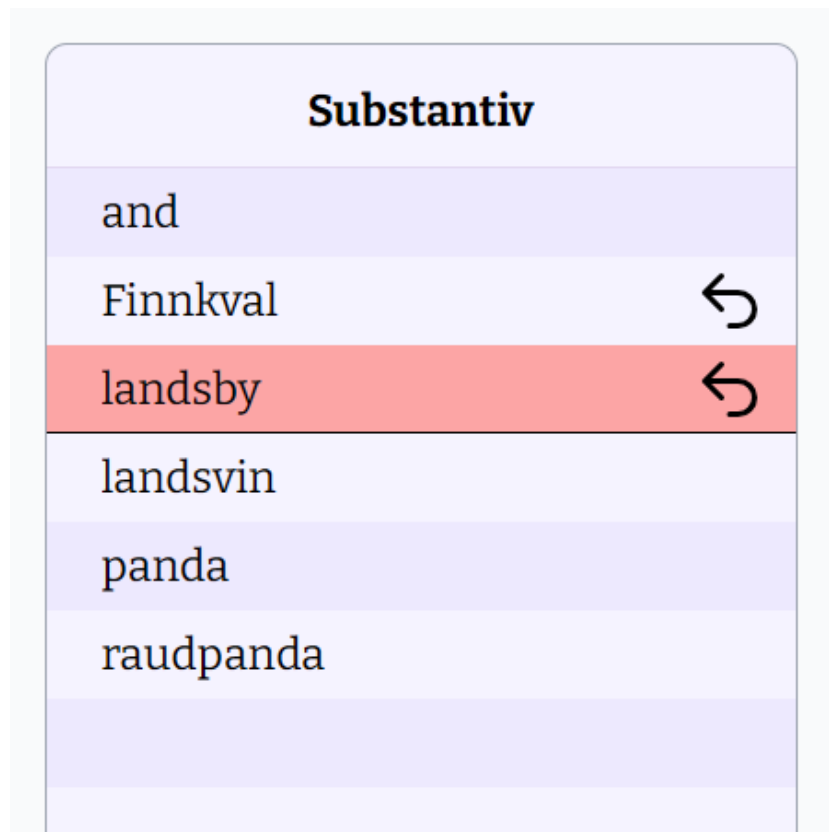


Figure 4.2.21: When a word is changed or deleted, an undo button will show up. The change is not saved immediately.



Substantiv	
and	
Finnkval	↶
landsby	↶
landsvin	
panda	
raudpanda	

Figure 4.2.22: A word that is marked for deletion. The change is not saved immediately.

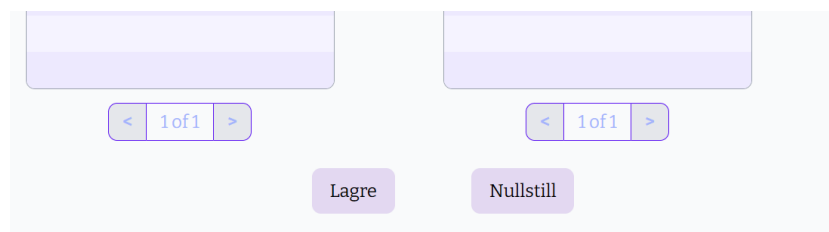


Figure 4.2.23: To save the changes, press the "Lagre" button. To reset all changes, press the "Nullstill" button.

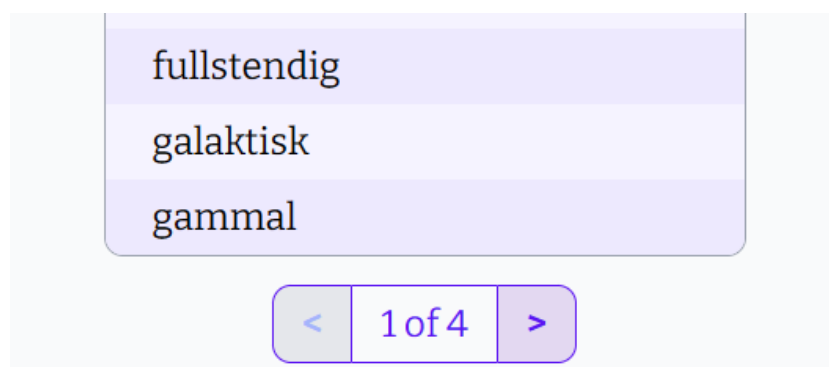


Figure 4.2.24: Below each table, there are buttons to change the page to see a different selection of words.

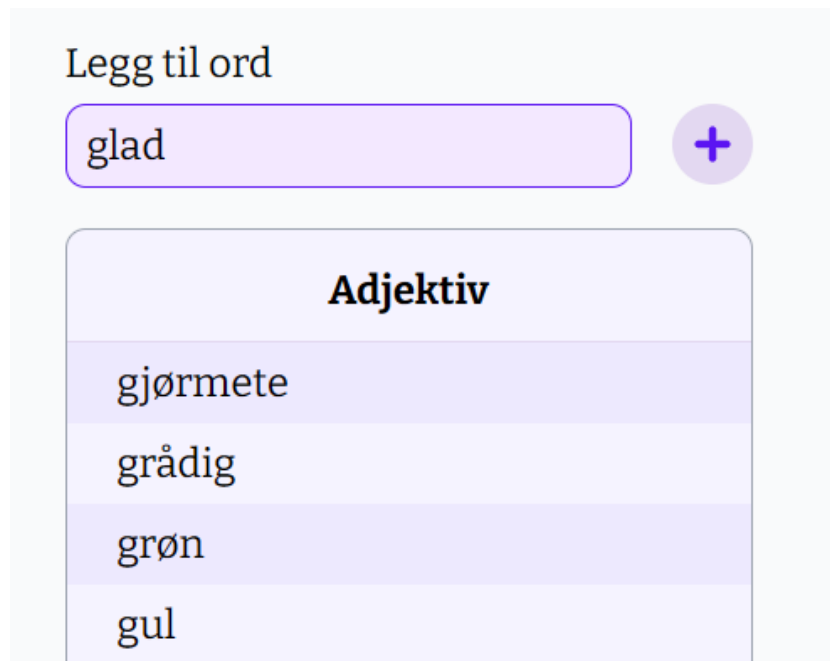


Figure 4.2.25: To add a new word to either list, write it in the input field above the table and click on the (+) button.

Users with the "organization admin" role have the ability to make default users into "quiz admin", which gives them access to administrator features, such as managing quizzes. This can be done by adding their e-mail address to a list. If the user already exists, they are immediately made into a quiz admin. If the user does not yet exist, they will get the role as soon as the account is created using that e-mail address.

4.2.13 Interaction Feedback

When a user interacts with elements, they get visual feedback that the interaction has been registered, and whether the action was successful or not. When an element is hovered or focused via keyboard navigation, a different style is applied. When an element is interacted with and it sends a request to the server, a loading indicator is displayed nearby while the response is awaited.

If the request fails on the server, an error message is returned to the client. If the user re-sends the request and it succeeds, the error message is no longer displayed.

4.2.14 Error Handling

The Echo framework supports centralized error handling. We handle errors locally as they may occur. Then we either return our own custom errors or the "echo.HTTPError" type, which includes HTTP status code and an error message.

In the centralized error handler, these errors are caught and processed. They result in proper HTTP responses back to the client. Non-"echo.HTTPError" errors turn into "500 Internal Server Error" responses and the error itself is logged. Other errors with status codes above 500 are returned with the actual error message hidden from the user. The error message is logged instead, to avoid leaking

potentially sensitive information. Errors with codes below 500, are returned to the user with the status code provided and the error message is included.

The response body is slightly different depending on the requested route. In the context of API calls, the error response is an HTML "p" tag with red styling, which is then displayed on the page somewhere. Routes that are meant to respond with a full page, respond with an error page instead. The page contains information about what has happened and a "back to safety" button.

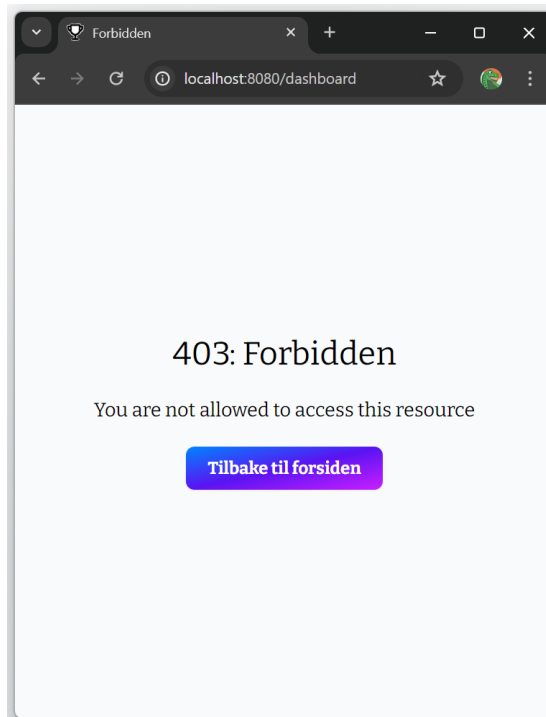


Figure 4.2.26: Error page displayed when a non-admin user tries accessing the admin dashboard.

4.3 Engineering Results

This section will feature content from the development, prototype and design phase that is relevant to the engineering of the final version of the project. The application language is Norwegian, since the target group speaks Norwegian. See appendix A for the GitHub repository.

4.3.1 Architecture

The application itself is a monolith following Model-View-Controller architecture.

The controller is implemented in the Echo Web framework for Go. It handles HTTP requests, interacts with the model layer and renders the views.

The views are HTML templates. They are implemented using Templ, they essentially use Go plus HTML markup.

The model layer defines the domain models and interacts with the database. The business logic is implemented in this layer. The database is used as a storage

mechanism and partly implements the business logic. More complex SQL queries, constraints, triggers, functions and views are used for this purpose.

The user interacts with the application via their Web browser. However, the browser does not talk directly with the application. Caddy Web server acts as a reverse proxy and is responsible for managing TLS certificates needed for HTTPS.

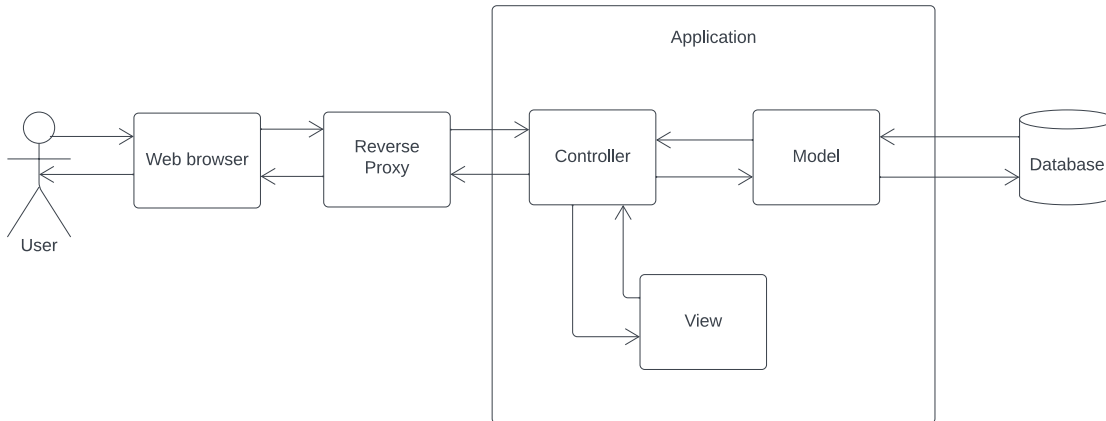


Figure 4.3.1: High level architecture overview.

4.3.1.1 Project Hierarchy

The project layout follows common patterns for open source Go projects (which are specified in *Standard Go Project Layout* repository on GitHub [129]), as the Go development team does not provide guidelines for project structure. Key points to notice are the entry points to binaries being in the "cmd" directory and modules written for this project living in the "internal" directory.

The application's entry point is in the "main.go" inside the "cmd/server/". Inside the "cmd" directory there are two additional directories, "db_populator" and "test_users". The former is a Go script for database seeding, and is used in development and testing environments. The latter is a helper Web server used in end-to-end testing with Bruno (discussed in section 4.5.5).

Inside the "internal/" directory, there are multiple sub-modules. Among others, connection logic to services such as the database or object store (bucket). There is a "models/" directory containing all domain models and business logic. There is also the "web_server" which contains a router, HTTP handlers for each endpoint, and other controller logic. The "web_server" module also contains the "views/" sub-module, which contains all templates and user-facing logic.

The following is a simplified file tree of the project. Items ending with a "/" are directories with content hidden for clarity and readability.

```

+-- assets/
+-- bruno/
+-- Caddyfile
+-- cmd
|   +-- db_populator/
|   +-- server
  
```

```
| | +-- main.go
| +-- test_users/
+-- data/
+-- db
| +-- db_integration_test_suite.go
| +-- db_populator/
| +-- migrations/
+-- docker-compose.yaml
+-- Dockerfile
+-- example.env
+-- go.mod
+-- go.sum
+-- internal
| +-- auth/
| +-- bucket/
| +-- config/
| +-- database/
| +-- models
| | +-- ai/
| | +-- articles/
| | +-- labels/
| | +-- questions/
| | +-- quizzes/
| | +-- sessions/
| | +-- users
| |   +-- access_control/
| |   +-- usernames/
| |   +-- user_quiz/
| |   +-- user_quiz_summary/
| |   +-- user_ranking/
| |   +-- user_roles/
| |   +-- users.go
| +-- utils/
+-- web_server
| +-- api.go
| +-- middlewares/
| +-- web
|   +-- handlers
|     | +-- api/
|     | +-- auth_handlers.go
|     | +-- dashboard_pages_handlers.go
|     | +-- error_handler.go
|     | +-- public_pages_handlers.go
|     | +-- quiz_pages_handlers.go
|     | +-- websocket.go
|     +-- router/
|     +-- views
|     +-- components
```

```

|         | +-- dashboard_components/
|         | +-- error_dialog.templ
|         | +-- error_text.templ
|         | +-- icons/
|         | +-- layout_components/
|         | +-- loading_indicator.templ
|         | +-- profile_components/
|         | +-- quiz_components/
|         | +-- terms_of_service.templ
|         | +-- tooltip_button.templ
|         | +-- tooltip.templ
|         | +-- user_management/
|         +-- pages
|             +-- dashboard_pages/
|             +-- public_pages/
|             +-- quiz_pages/
+-- LICENSE
+-- Makefile
+-- README.md
+-- scripts/
+-- tailwind.config.js

```

4.3.1.2 Environments and Secrets

The application works mainly in two environments: "dev" for development and "prod" for production. The main difference is which Docker services are ran. More on that in section 4.3.1.4.

There is also a key difference in the way the application itself is ran in the two environments. In production, the application image is ran in a Docker container. However in development, the application is ran directly on the host via Air utility. This way it is recompiled and restarted whenever the source files are changed.

Additionally, in development environment the application exposes a Web socket endpoint used to detect whether the application is running. In the "dev mode", all webpages include a simple JS script connecting to the said Web socket. The script refreshes the webpage when the connection is lost, i.e. the server restarts after a recompilation. This provides a simple solution for auto-refreshing when the server restarts.

A ".env" file is used to conveniently set environmental variables required in the project. These variables include general configuration for the application and services ran via Docker. Additionally, secrets are set using the environmental variables. This refers to things such as the database credentials and API keys. The repository contains an "example.env" file. Following is a slightly simplified content of this file.

```

1  COMPOSE_PROFILES=dev
2  DOMAIN_NAME=localhost
3

```

```
4 DB_NAME=nyhetsjeger
5 DB_PORT=5432
6 DB_HOST=localhost
7 DB_USER_ROOT=postgres
8 DB_PASSWORD_ROOT=password
9
10 DB_USR_APP=server_user
11 DB_PASSWORD_APP=password
12
13 PORT=8080
14 GOOGLE_REDIRECT_URL=http://localhost:8080/auth/google/callback
15 GOOGLE_CLIENT_ID=
16 GOOGLE_CLIENT_SECRET=
17 SESSION_SECRET=
18 AES_KEY=
19 ALLOWED_FRAME_ANCESTORS=
20
21 BUCKET_USER_ROOT=user
22 BUCKET_PASSWORD_ROOT=password
23 BUCKET_URL=localhost:9000
24 BUCKET_REGION=us-east-1
25 BUCKET_NAME=nyhetsjeger
26 BUCKET_ACCESS_KEY=key
27 BUCKET_SECRET_KEY=secret-key
28 BUCKET_USE_SSL=false
29
30 OPENAI_KEY=SomethingVeryVerySecret
```

Code 4.1: .env file

”COMPOSE_PROFILES” environmental variable is used to set Docker compose profile, and by the application to determine whether to run in ”dev” or ”prod” mode.

The CI/CD setup needs access to secrets in order to publish a docker image to the registry and to SSH into the server for deployment. These secrets are managed by GitHub secrets.

4.3.1.3 Application Image

The application can be built into a Docker image using a ”Dockerfile”. The ”Dockerfile” specifies a multistage build, resulting in a minimal final image. At each stage, the files needed for the current stage are explicitly copied into the environment and then utilities that are needed are ran.

We start by copying all files necessary for Tailwind to build the final CSS file. The Tailwind CLI is then invoked.

At the second stage, the files needed to compile the Go application are copied into the environment, and Templ CLI is invoked to generate Go source files from ”.templ” files. Finally the ”go build” command is invoked to build the application.

In the last stage, the Go binary, the Tailwind generated CSS file, and the static assets are copied into the environment.

Following is the content of the "Dockerfile".

```
1 FROM oven/bun:1 AS tailwind-builder
2 WORKDIR /app
3
4 COPY assets/css/styles.css ./assets/css/
5 COPY internal/web_server/web/views
  ↪ ./internal/web_server/web/views/
6 COPY tailwind.config.js ./
7
8 RUN bunx tailwindcss build -i assets/css/styles.css -o
  ↪ assets/css/tailwind.css
9
10 FROM golang:1.22 AS go-builder
11 WORKDIR /app
12
13 COPY go.mod go.sum ./
14 RUN go mod download
15
16 RUN go install github.com/a-h/templ/cmd/templ@latest
17
18 COPY cmd/ ./cmd/
19 COPY internal/ ./internal/
20
21 RUN templ generate
22 RUN go build -o ./bin/main ./cmd/server/main.go
23
24 FROM golang:1.22
25 WORKDIR /app
26
27 COPY --from=go-builder /app/bin ./
28 COPY assets/ ./assets/
29 COPY --from=tailwind-builder /app/assets/css/tailwind.css
  ↪ ./assets/css/tailwind.css
30
31 EXPOSE 8080
32
33 CMD ["/main"]
```

Code 4.2: Docker file

4.3.1.4 Docker Compose

Docker compose is used in both the development and production environments. We use a single Docker compose file, and "compose profiles" are utilized to specify which services are needed in which environments.

Services such as the database (service name "db") are ran in both the development and production environments. Contrary, Adminer is available only in the development environment, and the "server" service (which is the application image) is active only in production. In the development environment, the application is ran directly on the host, not via Docker. Additionally, the volumes are defined in the compose file. This way they are managed by Docker.

Following is the content of our "docker-compose.yaml" file.

```
1  version: "3.8"
2  services:
3    db:
4      image: postgres
5      restart: always
6      ports:
7        - ${DB_PORT:-5432}:5432
8      env_file:
9        - .env
10     environment:
11       POSTGRES_USER: ${DB_USER_ROOT}
12       POSTGRES_PASSWORD: ${DB_PASSWORD_ROOT}
13       POSTGRES_DB: ${DB_NAME}
14       TZ: ${TZ:-Europe/Oslo}
15     volumes:
16       - db-data:/var/lib/postgresql/data
17     profiles:
18       - dev
19       - prod
20
21     bucket:
22       image: quay.io/minio/minio
23       restart: always
24       ports:
25         - 9000:9000
26         - 9001:9001
27       env_file:
28         - .env
29       environment:
30         MINIO_ROOT_USER: ${BUCKET_USER_ROOT}
31         MINIO_ROOT_PASSWORD: ${BUCKET_PASSWORD_ROOT}
32         TZ: ${TZ:-Europe/Oslo}
33       volumes:
34         - bucket-data:/data
35       command: server /data --console-address ":9001"
36       healthcheck:
37         test: [ "CMD", "mc", "ready", "local" ]
38         interval: 5s
39         timeout: 5s
40         retries: 5
```

```
41     profiles:
42         - dev
43         - prod
44
45     adminer:
46         image: adminer
47         restart: always
48         environment:
49             TZ: ${TZ:-Europe/Oslo}
50             ADMINER_PLUGINS: "enum-types"
51             ADMINER_DESIGN: "dracula"
52         ports:
53             - 8081:8080
54         depends_on:
55             - db
56         profiles:
57             - dev
58
59     server:
60         image: ghcr.io/molnes/nyhetsjeger:main
61         restart: always
62         env_file:
63             - .env
64         ports:
65             - 8080:${PORT:-8080}
66         environment:
67             PORT: ${PORT:-8080}
68             TZ: ${TZ:-Europe/Oslo}
69         depends_on:
70             - db
71         volumes:
72             - ./data/articles:/app/data/articles
73         profiles:
74             - prod
75
76     caddy:
77         image: caddy:latest
78         restart: unless-stopped
79         env_file:
80             - .env
81         environment:
82             DOMAIN_NAME: ${DOMAIN_NAME:-localhost}
83             TZ: ${TZ:-Europe/Oslo}
84         ports:
85             - "80:80"
86             - "443:443"
87             - "443:443/udp"
88         volumes:
```

```
89     - ./Caddyfile:/etc/caddy/Caddyfile
90     - caddy_data:/data
91     - caddy_config:/config
92     depends_on:
93     - server
94     profiles:
95     - prod
96
97     volumes:
98     db-data:
99     name: nyhetsjeger-postgres-data
100    bucket-data:
101    name: nyhetsjeger-bucket-data
102    caddy_data:
103    caddy_config:
```

Code 4.3: Docker Compose file

4.3.2 Database

The database is the primary source of data for the application. Since the application itself is a stateless RESTful service, all state is persisted in the database. The data model is designed to reflect all possible application states.

Most unique identifiers are UUIDs. The exceptions for this are entities where one of the values is inherently unique, and the "http_sessions" table which is managed by a third-party session store implementation that we use.

Following is a diagram of the data model (figure 4.3.2). Primary keys are marked with underline, foreign keys are marked with an asterisk (*). Nullable and unique properties are **not** denoted.

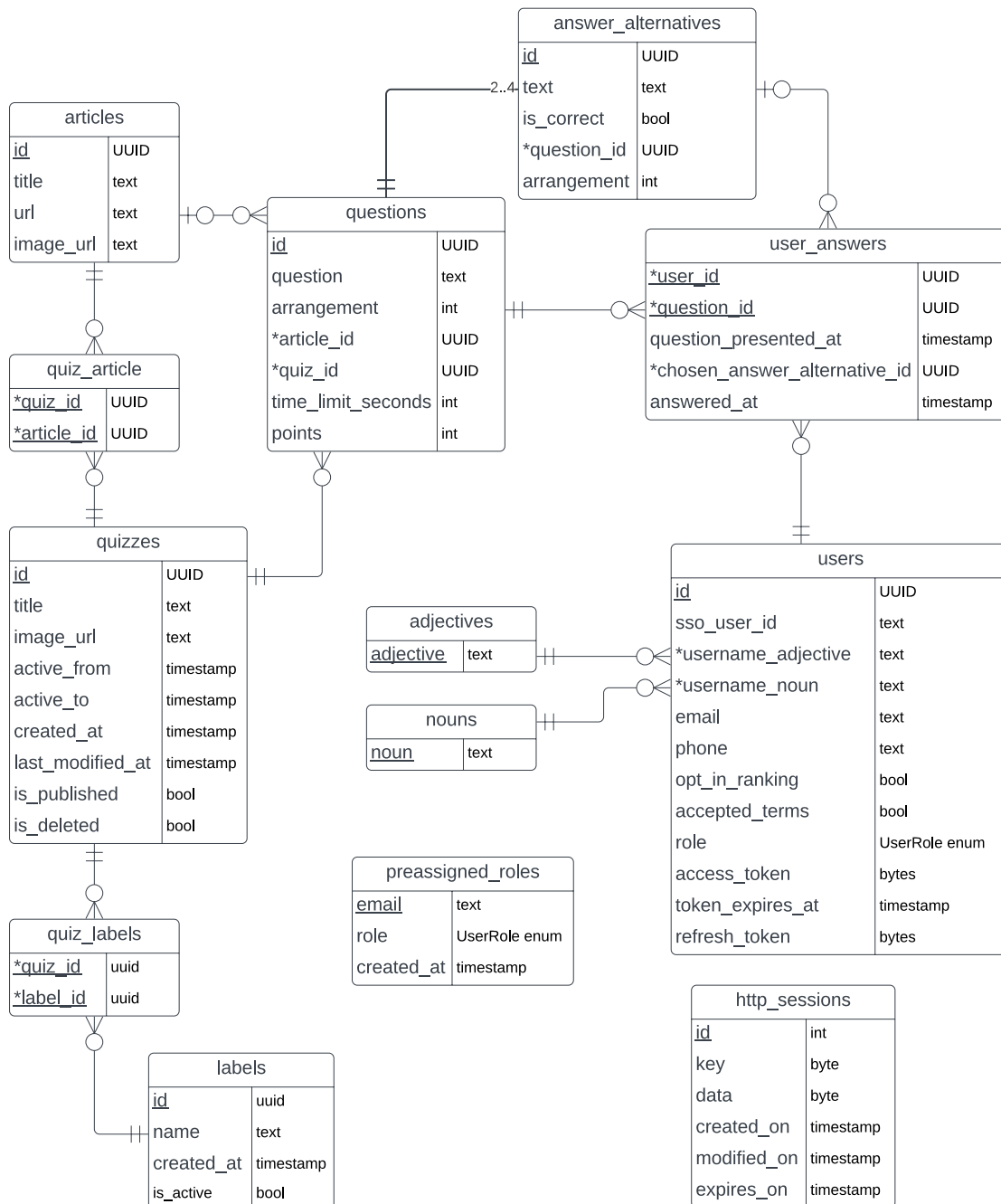


Figure 4.3.2: Database schema.

All application state is persisted in the database, even intermediate and incomplete data. This leads to the existence of some circular relations that are necessary from the logical standpoint. Let's consider the two circular occurrences.

First let's look at the tables "user_answers", "answer_alternatives" and "questions". For the quiz playing functionality, it was significant to know at which time the question was presented to the user, to calculate the points. The action of a user viewing a question for the first time is recorded in the "user_answers" table, but the properties "chosen_answer_alternative_id" and "answered_at" are set to *NULL*. This is because the user has not answered the question, only viewed it. After the user picks an answer, this row is updated and the two properties are no longer *NULL*. This is why "user_answers" must be connected to the "questions"

table, because it will not always be connected to "questions" via "answer_alternatives".

Second example is the circular relationship between the tables "articles", "questions", "quizzes" and "quiz_articles". The "quiz_articles" table is only necessary for the many-to-many relation between articles and quizzes, thus it can be ignored for this example. When creating a quiz, articles can be added to it; creating a relation between "quiz" and "articles". Then when adding questions to said quiz, the administrator can choose which article the question is based on from the list of articles related to the quiz; creating a relation between "questions" and "articles". This way a quiz can be based on multiple articles, even before creating questions "using" those articles. At the same time, players are only shown articles that are connected to the quiz questions. This way, they will not get suggested articles that are unrelated to the quiz they played.

We use database constraints and triggers to increase data integrity and simplify queries. All tables that include an "arrangement" property, have a trigger that will automatically set the appropriate value upon insertion, incrementing it with every insertion.

Database views are used to simplify more complex and commonly used "SELECT" queries. For example there is a "user_question_points" view, which takes care of joining multiple tables and displays exactly which questions a user has answered, and how many points they earned. It utilizes the "calculate_points_awarded(...)" SQL function mentioned in section 4.2.5. This view is visualized in the table 4.3.1.

user_id	question_id	quiz_id	chosen_answer_id	answered_at	points...
...

Table 4.3.1: "user_question_points" SQL view. The last column is "points_awarded".

Similarly, "user_quizzes" is a view summarizing results from the previous one, aggregating points. It is useful for displaying rankings. This view is visualized in the table 4.3.2.

user_id	quiz_id	total_points	is_completed	finished_at	answered...
...

Table 4.3.2: "user_quizzes" SQL view. The last column is "answered_within_active_time".

4.3.3 Domain Model

The team made two domain models that represents the main domains of the application, from the perspectives of the normal users and the administrators. We used these to ensure that everyone in the group was on the same page regarding general structure of our application.

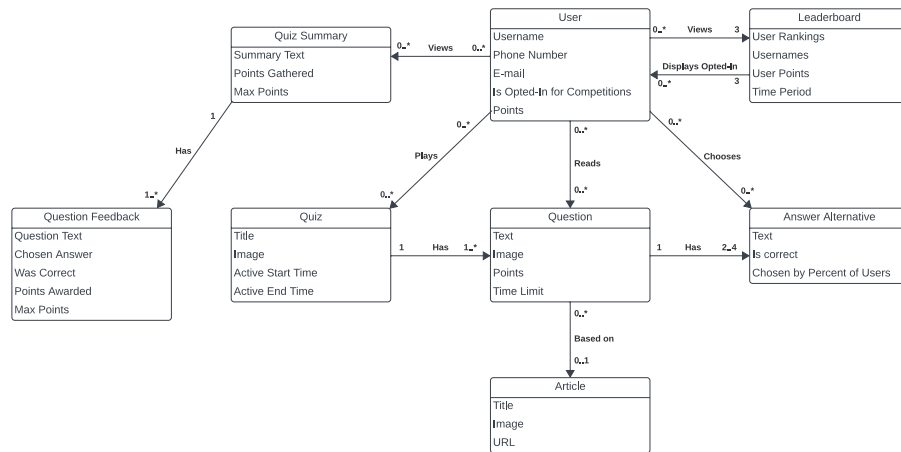


Figure 4.3.3: Domain model from the perspective of a Nyhetsjeger user.

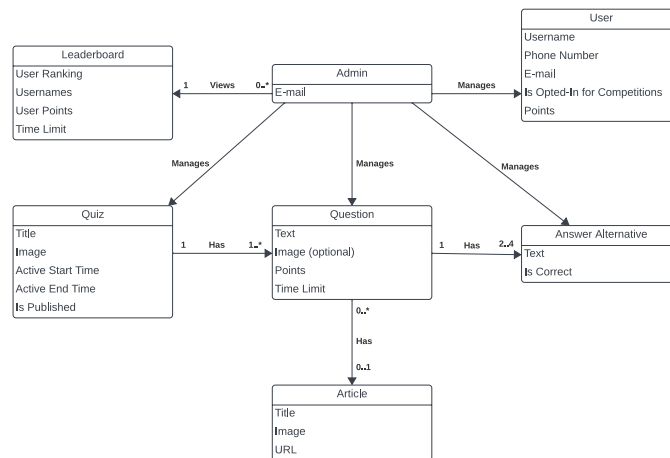


Figure 4.3.4: Domain model from the perspective of an administrator.

4.3.4 Prototyping

To develop the graphic part of our application, we used wireframes. The team made low-fidelity wireframes first, then high-fidelity designs after. Both included simple navigation between screens, to see how buttons and links would function and give an idea of the flow of the application.

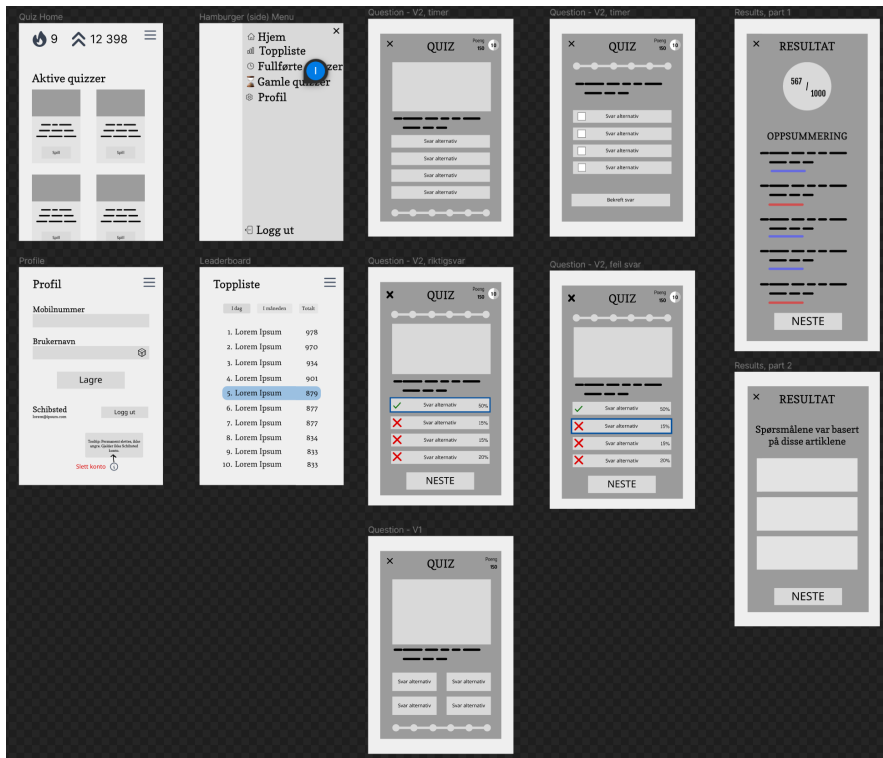


Figure 4.3.5: Early wireframes for the application.

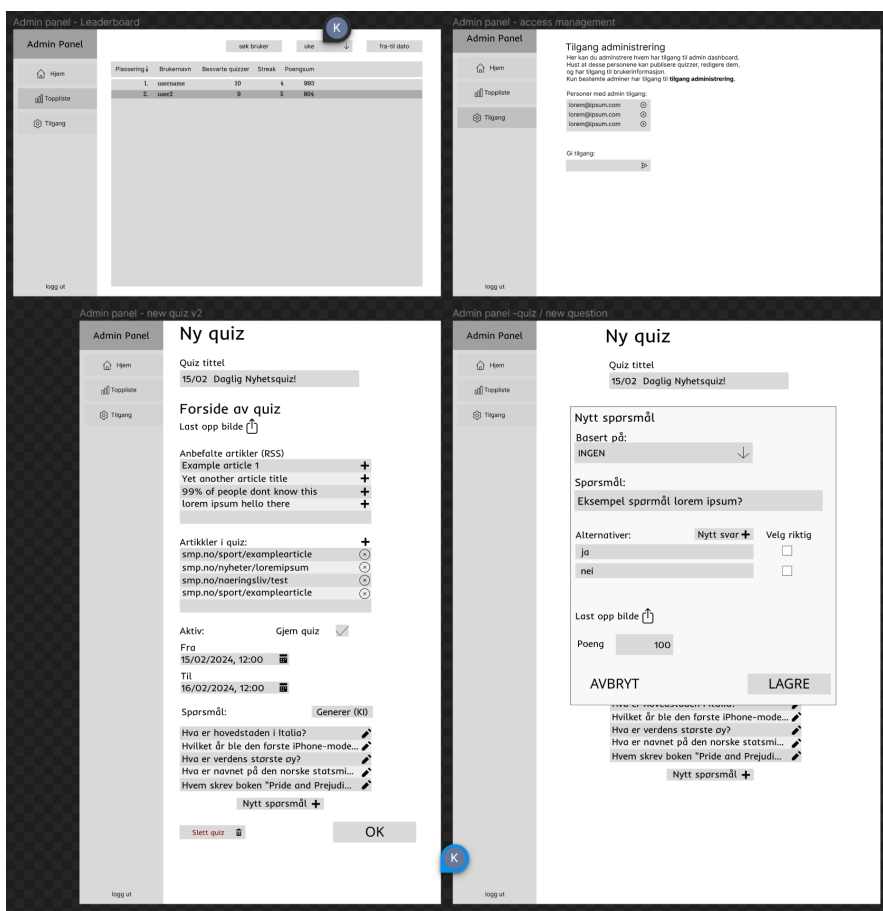


Figure 4.3.6: Early wireframes for the dashboard.

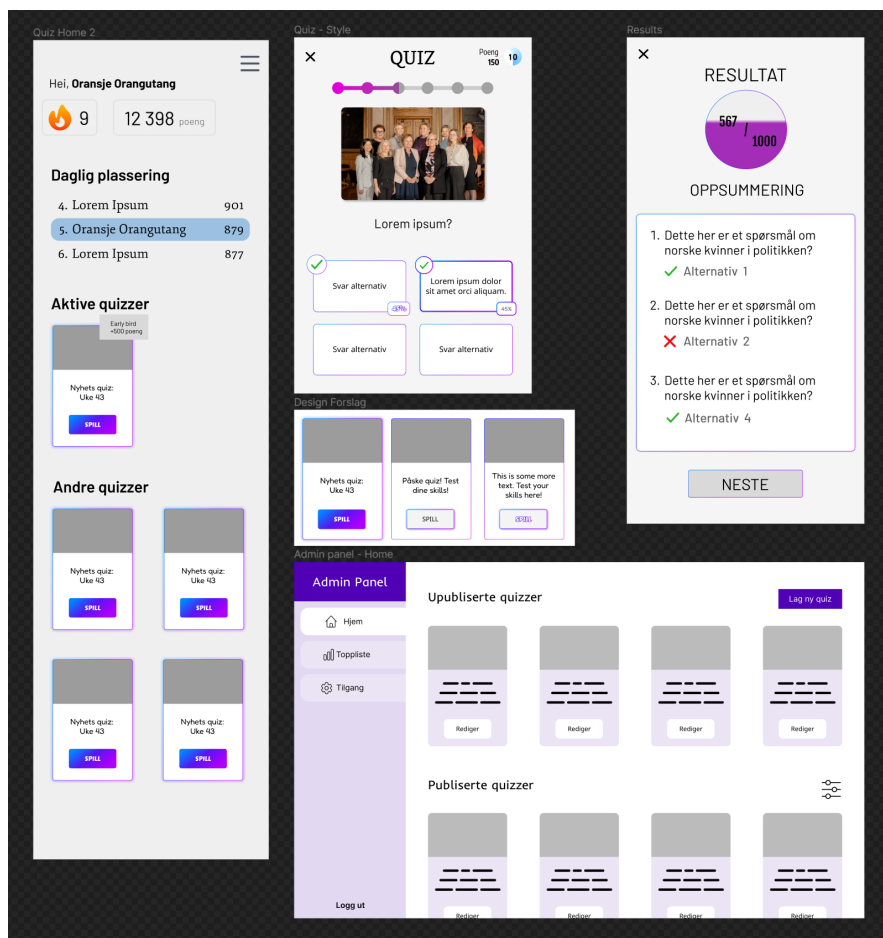


Figure 4.3.7: High-fidelity designs for parts of the application.

4.3.5 User Interface

The Web application was made using Go with Templ for generating the HTML itself, HTMX to swap out components, and Tailwind for styling. The application was split into two main parts: Nyhetsjeger for normal users, and Dashboard for the administrators. Both "sites" have a button for navigating to the other one, which is only visible to administrators.

The application was designed to be responsive and dynamic. It is responsive to a variety of devices, such as mobile and desktop. Some components look different based on the "state" of the application, such as if there are available quizzes or not. The team used the feedback from usability testing to make improvements on the design, which is discussed further in 4.5.6. Cross-browser compatibility was also taken into consideration, ensuring that any important feature worked on all major browsers.

The application follows guidelines for accessibility, such as WCAG. For example, all text and background colors have a higher contrast ratio than 4.5:1. We do not present information that depends solely on color. We use semantic HTML elements. The application was designed with keyboard navigation in mind. Long sections of navigation, such as the main menu and the username tables, can be skipped using a "skip" button. The application was also designed with high-contrast mode in mind.

The technologies used are lightweight compared to other popular Web frameworks, granting fast performance. Using server-side rendering gives a faster page-load time. The usage of HTMX allows for only loading the necessary parts of the application, giving a smooth experience. Tailwind generates a CSS file that only includes the used styles, making it a lightweight solution. Another optimization is that uploaded images are automatically converted to JPEG, a compressed image format.

4.3.6 HTTP Routes

We group together HTTP routes and different sets of middlewares are applied to those groups. Before a request can get to its appropriate handler, it is processed by all middlewares applied on the handler's group. Said middlewares are used to enforce authentication, enforce user role, or add values to "context.Context" associated with the request (see section 3.4.1 for details about "context.Context"). A middleware can return early with an HTTP error if needed. For instance, when the caller does not have the required role.

The request handlers take in query parameters and form data to return responses containing the status code and HTML. Additional information is sometimes added to the response headers, to tell the frontend where to place components on the page or trigger events. This is used when error messages should be placed on the page elsewhere from the successful response.

All API endpoints and page handlers require authentication, except for guest users and the authentication endpoints themselves. If the user does not have the necessary access to view a page, they are shown an error page.

4.3.7 Object Store

We needed a tool to store images uploaded from quiz admins. We solved that by using the S3-compatible bucket MinIO. The admin can either choose an image from a provided article, upload from a URL, or provide an image from their local file system.

To upload the images, we used the SDK provided by MinIO. We generated a random UUID, got the file type and size, and then uploaded it with the SDK provided function.

The bucket is hosted on the same server as the Web app itself, with the help of Docker compose.

4.3.8 Security Measures

To protect against SQL injection, we utilize prepared statements, which is built into Go's SQL package. We protect against cross-site scripting by using Templ. Their security measures are discussed in 3.4.4.

No secrets were hard-coded into the source code or included in the version control. Instead, they were loaded into the environment from the ".env" file, and the application read them using Go standard library's "os" module.

The application has working HTTPS using Caddy as a reverse proxy. The security benefits of HTTPS is discussed in section 2.5.1.

4.4 Theoretical Results

Theoretical findings regarding the project from the different phases will be presented in this section.

4.4.1 Gamification

The leaderboard feature was a requirement in the initial project specification, but it was up to the team to design it and make it appealing. The stakeholder informed us that they liked *Duolingo*'s leaderboard, emphasizing the need for a fun design.

The implementation of leaderboards necessitates a method for measuring player performance. The leaderboard system is further discussed in 4.2.7. The points are used both to gauge player performance, but also act as a reward. We animate the point counter by showing each point ticking up like an odometer (using the library `odometer.js`), and animate the results screen by "filling up" the points inside a circle.

4.5 Quality Assurance

Quality assurance is important when it comes to the development phase. It dictates how easy the code can be built upon for more features, and detecting and fixing issues.

4.5.1 Documentation

In the project, all functions have documentation that lives up to the goals described in 2.4.8.

4.5.2 Code Quality

Code reviews were essential to maintain code quality. To merge changes into the "dev" branch from other branches, a pull request had to be approved by another member of the group. If any part of the code was not up to the necessary standard, a request for changes was made in *GitHub*.

A tool for upholding maintainability and code quality was Sonarlint. It works as mentioned in 3.5.3. We regularly had a member of the group go through every file with Sonarlint to catch issues that we might have initially missed.



Figure 4.5.1: A *SonarLint* rule shown in *Visual Studio Code*.

Another point of quality assurance regarding the code is the formatter. As mentioned in 2.4.4, developers may not agree on how to format the code. To apply a cohesive style of code while developing, a specific formatter was used by all team members. The benefit was that the code would be easier to read. The formatters used in the project were the standard Go formatter and the standard Templ formatter.

High cohesion and low coupling are also indicative of code quality. To maintain low coupling, we set layers. We defined which layers each layer could access, limiting how much a layer needs to know about another layer. As mentioned in 2.4.5, coupling is the degree to which functions depend on each other. We tried to maintain a structure where each function was independent and did not rely on knowing about the implementation of other functions.

High cohesion was maintained by making sure each function only had a certain job. In case a function needed to do multiple jobs, it was refactored into multiple smaller functions. This also helped to avoid duplicate code.

4.5.3 Unit Testing

Since most of the functionality requires the database to run, the unit tests mostly test helper functions. We avoided using mock data, because they could show

results not reflecting how it actually is implemented. And it would be difficult to mock, since most of the methods are for inserting or selecting from the database.

Unit testing is implemented using the standard testing library provided by Go.

4.5.4 Integration Testing

The project's test collection consists mainly of integration tests since the majority of business logic lives in the database functions, SQL queries, and views. This limits the usability of unit testing. This is touched upon in section 2.7.2.

Integration testing is implemented using packages *Testcontainers* and *Testify*, as discussed in section 3.5.5. We implemented a base test suite with a *PostgreSQL* test container.

The base suite implements functions to be called at suite setup and tear down. The former is responsible for setting up the test container and establishing the database connection. The latter closes the database connection and terminates the container. Code 4.4 shows the implementation of these functions.

```

1 // Runs once at test suite setup.
2 // Creates test psql container, creates a database connection
  → and sets a pointer to it as the struct field.
3 func (s *DbIntegrationTestBaseSuite) SetupSuite() {
4     ctx, ctxCancel :=
5     → context.WithTimeout(context.Background(),
6     → 45*time.Second)
7     defer ctxCancel()
8
9     psqlContainer, err := newPostgreSQLContainer(ctx)
10    s.Require().NoError(err)
11    s.psqlContainer = psqlContainer
12
13    db, err := database.NewDatabaseConnection(
14    → s.psqlContainer.getDBUrl())
15    s.Require().NoError(err)
16    s.DB = db
17 }
18
19 // Ran when the suite is done. Closes the DB connection and
  → terminates the container.
20 func (s *DbIntegrationTestBaseSuite) TearDownSuite() {
21     ctx, ctxCancel :=
22     → context.WithTimeout(context.Background(),
23     → 5*time.Second)
24     defer ctxCancel()
25
26     s.Require().NoError(s.DB.Close())
27     s.Require().NoError(s.psqlContainer.Terminate(ctx))
28 }

```

Code 4.4: Setup and tear down functions for the base integration test suite.

This suite also implements functions to be called at test setup and tear down (before each and after each). The former migrates the database schema all the way up and seeds it with test data. The former brings the schema down, effectively purging all data. This way each test is independent. Code 4.5 shows both function implementations.

```

1 // Runs before each test.
2 // Migrates the database up, runs population (seeding)
  ↪ function.
3 func (s *DbIntegrationTestBaseSuite) SetupTest() {
4     migrator, err := getMigrator(s.psqlContainer.getDBUrl())
5     s.Require().NoError(err)
6     s.Require().NoError(migrator.Up())
7
8     db_populator.PopulateDbWithTestData(s.DB)
9 }
10
11 // Runs after each test.
12 // Migrates the DB all the way down, removing all data.
13 func (s *DbIntegrationTestBaseSuite) TearDownTest() {
14     migrator, err := getMigrator(s.psqlContainer.getDBUrl())
15     s.Require().NoError(err)
16     s.Require().NoError(migrator.Down())
17 }

```

Code 4.5: Setup and tear down functions for integration tests.

The base test suite acts as a framework for development of integration tests. All modules requiring integration testing setup an own test suite. Composition is used to include the base suite and its behaviour. Code required to setup a test suite for any module is in code 4.6.

```

1 type UsersIntegrationTestSuite struct {
2     db_integration_test_suite.DbIntegrationTestBaseSuite
3 }
4
5 // Users module integration test suite entrypoint.
6 func TestUsersIntegrationSuite(t *testing.T) {
7     suite.Run(t, new(UsersIntegrationTestSuite))
8 }

```

Code 4.6: Users module integration test suite setup.

4.5.5 End-to-end Testing

The project also includes some end-to-end tests, implemented as API tests with Bruno. Since the API endpoints of the application respond with HTML elements instead of JSON, it is rather cumbersome to verify the response bodies with JS. For this reason, tests rely on HTTP status codes, and assertions are made on those.

The Bruno test suite is set up to verify authentication and authorization of routes in different groups. It checks if endpoints requiring authentication respond with "401 Unauthorized" status code if the client does not have a valid session. It also verifies that routes requiring a special role respond with "403 Forbidden" if user associated with the session does not have the required role.

In order to assign valid sessions to the Bruno client, we implemented a helper web server. This program is separate from the application itself (see 4.3.1.1). The server has four endpoints. Three of which are used to assign valid HTTP user sessions with different user roles to the client, and the last endpoint is used as a shutdown signal.

To simplify the running of tests, we use a shell script. It has three things to do. Firstly, it starts the helper server, then runs the test suite using Bruno CLI, and in the end uses cURL to send a shutdown request to the helper server.

```
Requests:    31 passed, 31 total
Tests:       0 passed, 0 total
Assertions:  25 passed, 25 total
```

Figure 4.5.2: Bruno test results, from Bruno CLI

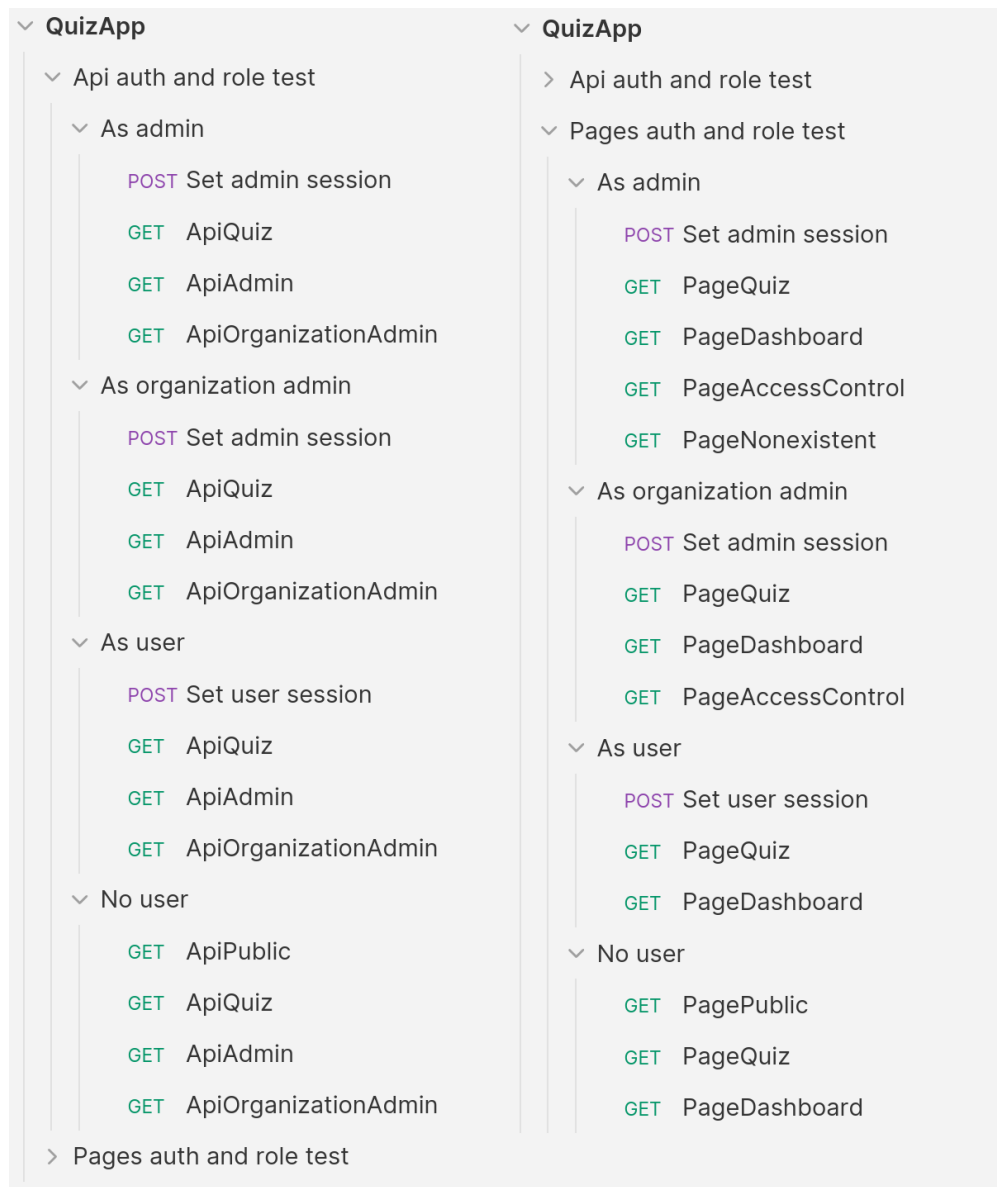


Figure 4.5.3: Bruno test suite

4.5.6 Usability Testing

As mentioned in 2.4.9, usability testing gives feedback on how users interact with the application; their impression of the design and any feature requests. Bugs can also be discovered in this process.

When preparing for a usability testing session, a guideline was created with the things that the team wanted to know. This guideline also included rules for the group, such as not helping if someone was struggling, in order to observe if the testers could figure it out themselves.

When doing the usability testing itself, notes were written on the observation of how the testers used the application, and struggled with or solved tasks.

One of the features requested was making the "No quizzes" warning text more humorous and encouraging, such as instead of saying "You have no active quizzes" say "Wow, you have done all the quizzes, good job!"

Another improvement was that some of the functionality of the options when creating a quiz was unclear. Simultaneously, they thought there was too much explanatory text on the top of the page, about how saving changes worked. To solve this without cluttering the page, tooltips were added. Clicking or hovering on it displayed a short explanation of what the input did.

During usability testing, we found bugs such as admins not being able to correctly upload images with URLs from the SMP network.

Feedback was also received on what was well done, such as easy and intuitive design, the color choices, and the use of gradients. They found the usernames funny, and liked the animation on the results page.

DISCUSSION

The goal of this project was to create a quiz application for Web and mobile, where users could play quizzes and view the leaderboard. This goal has been met. This chapter will discuss the results and the decisions made during the process leading to the results.

5.1 Theoretical Discussion

5.1.1 Technology

In the project, it was largely up to the team which technologies we wanted to implement. In the information collection stage, the team had to gather a list of advantages and disadvantages for different technologies.

The group chose to go use Go as the main language, for a multitude of reasons. It has a gentle learning curve, compiles very fast, is fast to develop in, and has great null safety by forcing developers to handle errors by default. We chose Templ over Go's standard templating package, since it supports types and did not require learning new syntax. We included Air for live reloading, making the development process much smoother and faster. Echo was used because it had nice error handling, and at the time, Go's standard library did not have as good routing pattern matching as Echo.

HTMX was a nice, lightweight solution. It is elegant in its simplicity, and is very flexible and versatile, because it can be implemented in many different technology stacks. We used Tailwind, which is the most popular CSS framework. It is very easy to create responsive design, which was vital for the team as it needed to be supported on both Web and mobile.

When it came to choosing a database management system, all the members of the group had the most experience with PostgreSQL, making it ideal. It had all the features we would require for the project. We used Docker, because it provides a consistent environment, which simplifies deployment and reduces the risk of bugs.

We wanted images for quizzes, since that makes them a bit more interesting and memorable. We could have pointed to external URLs, but that did not provide great stability, since it relied on external factors. Instead, we chose to host images with our own bucket. We chose MinIO for the bucket because of the compatibility

with the Amazon S3 API, scaling, and the ease of self-hosting. The SDK allows us to connect to any S3 compatible bucket, which lets us avoid vendor lock in. If the stakeholder prefer to use another bucket that is compatible, it would be easy to switch.

Creating quizzes is a long process, as we agreed with SMP that the recommended minimum should be five questions. The administrator needs to know the articles well enough to make questions with two to four alternatives. This is where generation of questions with AI comes in. It streamlines the process, as the admin only needs to double check the article if the question is accurate. If the AI includes any typos or strange sentence structures, it is easy for the admin to tweak the question. We chose to use ChatGPT, because Sunnmøreposten has the ChatGPT Enterprise subscription, making it the ideal tool to integrate into the application.

5.1.2 Development Process

During the development process, we wanted to make a new iteration of the application every sprint. In the first sprint, we prioritized having pages to show to the stakeholder. It looked close to the wireframes, as the high-fidelity design came later. In this sprint, we did not use the database, but hard-coded sample data. We implemented the database the following sprint. This was the philosophy throughout the project; implement features that complemented each other in the easiest way. Polishing features could be done after meeting with the stakeholder and getting their input and approval.

5.1.3 Gamification

A feature originally suggested and planned in the early phases was the "daily streak" bonus, as it incentivizes daily playing. In the project description, SMP wanted daily quizzes. However, after further deliberation during the prototype phase, it was decided that weekly quizzes would be more doable, as the daily news situation can vary a lot. This made the streak feature impractical, since if it would be on a weekly basis instead it would lose a lot the intended effect of being habit forming.

If daily streaks had been implemented, it would be a minor source of points, where a small multiplier or flat score could be applied to the final score. The streak bonus would increase by a tiny bit each day, until it reached a maximum amount that would not cause too unfair of an advantage.

5.1.3.1 Engagement

A goal for our project was to create an engaging platform. If users play the quizzes a lot, they will be more likely to read the articles. To do this, we used several gamification and design choices to make the platform as engaging as possible. The best way to do this was to make playing the quizzes a satisfying experience, and to encourage competitiveness.

The main gamification components were points and leaderboards. Only allowing "active" quizzes to count towards the leaderboard also creates a sense of

urgency, encouraging the users to play every time a new quiz is published. Animations and gradients were used to make the application more visually interesting and fun. In section 5.4, we discuss other gamification features that could be added to increase engagement, as well as a notification system that could serve as a reminder to play.

5.1.4 Anti-cheating Measures

As with any competitive game system, cheating is always a serious concern, especially when the prize has monetary value. When designing the implementations of the application, we always had to keep this in mind.

One of the possible ways of cheating the system was by having the client lie about the time it took to answer a question, so they could get more points. To avoid this type of cheating, we stored the timestamp the user first saw the question and when it was answered. Thus, when the points are calculated based on time, we trust that it is accurate. All calculations and verification of questions are done server-side to avoid possible manipulation of data on the client side.

Another possibility is that a user may create multiple accounts, so they can scout ahead the questions. It is not possible to avoid this problem entirely, but one solution is to make it harder to create multiple accounts. This is discussed further in section 5.4.3.

5.1.5 Leaderboard

Since Sunnmøreposten wanted to give out prizes to the users with the most points, we had to consider a system that would make it fair for all users.

Since quizzes had start and end times, we had to decide whether inactive quizzes should still be playable or not. After some discussions with Sunnmøreposten, we decided to allow users to play the old quizzes, but not receive points. That way, new users wouldn't get an advantage over older users (having more quizzes to play and therefore gain more points), by grouping their responses to certain months.

We created three main groups of leaderboards "this month", "this year", and "all time." These would automatically be updated to show the current month or year. But this presented another issue: should quizzes count in the month they started or ended? Quizzes were not limited and could span multiple months, for example the last week of a month would span into the next. If it only counted in one of the months, it would be unclear for the user what happened when they would see that they received points for playing it in the other month. However, if it counted for both months, then it could affect leaderboards that should technically be closed. For example, if a user played a quiz in February, it could also update the leaderboard for January.

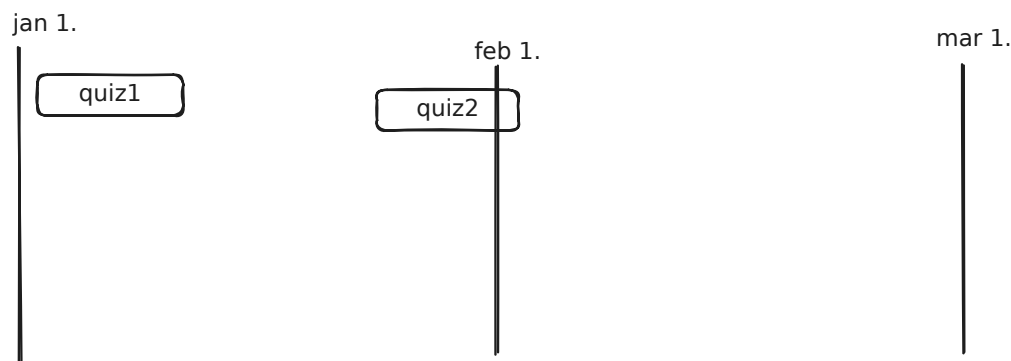


Figure 5.1.1: "quiz2" is active from some time in January, but it ends in February.

While discussing the dilemma with SMP, we came up with another solution; labels.

Labels allowed us to group quizzes arbitrarily, instead of by start and end times. Leaderboards would then show all points earned by active quizzes based on the label it belonged to. If the quiz spanned multiple months, it would be up to the admin which leaderboard(s) they wanted it to be applied to, and this would be made clear to the user. It also allows custom leaderboards, which could for example be used for Easter and Christmas season quizzes.

5.1.6 Design

Sunnmøreposten has a large and diverse user base, so universal design and user experience was very important to keep in mind, both when designing and implementing features. Examples of such considerations was discussed in sections 2.2 and 2.2.2. In Norway it is required of private companies to comply with a large subset of the WCAG 2.1 Level AA guidelines. In addition, we believe in inclusiveness and accommodating to different needs.

Since SMP uses blue as their primary color, we thought it would be nice to stick to a color scheme that does not deviate too far away, while still not identical. Therefore, we chose to stick with purple as the primary color, a lighter purple as the secondary color, and blue and pink as tertiary colors.

We included simple animations for things like points and the quiz "card". We wanted to make sure the animations are not distracting, but that they make the application seem more alive and fun. That creates a more engaging experience for the users.

When deciding on which icons to use, the most important was that they had licenses that fit our purposes. Design-wise, we stuck to outline based icons, instead of icons with fill. We felt that gave a "cleaner" look.

During development, went with a mobile-first approach for Nyhetsjeger, but desktop-first for the admin dashboard. This was because a lot of readers use their mobile phones to read news, but the administrators work on laptops or desktops.

5.1.7 Security Measures

For authentication, we opted to go with Google's single sign-on, for two main reasons. We trust Google to have better security than we could do ourselves, and

the team had experience with this implementation from a previous project and knew it would fulfill the project's needs. This way, we did not have to worry about securing passwords, etc.

5.2 Engineering Discussion

In this section we discuss our engineering results.

5.2.1 Inline Frame

The application is implemented in such a way that it can be included in a different website within an iframe (inline frame). We suggested this solution to SMP after we noticed they already include some third-party services this way. They were happy with the suggestion and agreed on this being the simplest and most flexible approach. This way it will work seamlessly for their website and mobile application, since their app is a wrapper for the Web app.

Our application's configuration includes "ALLOWED_FRAME_ANCESTORS" option, which is used to add values to the HTTP header "frame-ancestors". This is used to allow third party websites to include our application.

5.2.2 Usernames

Giving the administrators the ability to manage usernames gives them flexibility, so if they get more users in the future, then they can easily add more usernames. This implementation of usernames does not have great scalability, but with the improvement of AI, it becomes a trivial issue as they can easily generate more.

In the tables, only 25 words are visible at a time, unless overridden by the user using query parameters. This is done to make it a more manageable size, and not load in an unnecessary amount of words. That could overwhelm the user.

We initially contemplated allowing users to create their own usernames. However, this opened up the possibility of misuse, with bad actors potentially inputting profanities or other inappropriate words. Looking into a profanity filter for both Bokmål and Nynorsk did not show any promising results.

An alternative solution was to somehow generate unique usernames from a list of predefined options. We opted to go with a system similar to *BankID* on mobile, where two random words are combined [130].

The initial idea was to generate a list from the database dump of *Språkrådet*, available on *Nasjonalbibloteket's* website. However, the challenge was that it required manual review of each word to eliminate any inappropriate words.

The second approach involved using ChatGPT to generate a list of adjectives and nouns. However, when tasked with generating more than 100 words, it tended to repeat words or use words from other languages, making it an imperfect solution.

Ultimately, we found that manually writing each word was the most effective solution, with the help of ChatGPT, online word lists, and dictionaries. Consequently, we created a CSV file containing these words, which was loaded into the database.

Generated usernames also gives the benefit of anonymization and privacy, as users might be hesitant to share any personal details. The last factor is the

possibility to include humorous combinations, such as "galactic galaxy" or "dirty moose." This can make users more inclined to participate, to get their humorous username on the leaderboards.

5.2.3 Quiz Creation

There were many discussions and decisions made regarding the quiz creation process. The project requirements did not specify it, so it was up to the group to decide. We wanted to create strict enough guidelines that they could not make improper quizzes, but still give them enough freedom to allow for creativity.

A quiz must have minimum one question and has no upper limit, but the recommended amount is between five to ten questions. This recommendation is conveyed in a tooltip, but it is not a rule. We decided that this amount is ideal for a person to play during a lunch break. If there are too few questions, it is a boring quiz. If there are too many questions, it becomes tedious to play.

Each question should have two to four alternatives, which is a very common amount. We did not want to force a specific amount, so there is a range in options. We did not do more options, for two main reasons; user experience and limited screen size. If the users have too many options, it can lead to indecision and takes longer to read and process. It is ideal if everything on the quiz playing page fits on the screen at the same time. Too many options means the screen must be scrolled.

The ability to shuffle alternatives was not part of the initial plan, but was added on early in the development. We decided that the alternatives could not be displayed randomly for each user, due to the way we implemented the HTMX. We realised that people would have a bias in where they places the correct alternative(s), so shuffling it randomly in creation was the easiest way to prevent bias.

Originally, we wanted to scrape articles from an RSS feed to show them as suggestions. However, due to only being able to use articles from the data dump, this idea was scrapped. It could be a potential future improvement, but it is not a lot more convenient than copying the links from the webpage itself. Administrators will have to double check the content of the articles regardless, which cannot be done with the RSS feed.

SMP URLs are structured like, "https://smp.no/category/i/abcdef/article-title", where "abcdef" is the article ID. The URLs are self-healing, so as long as the path "/i/abcdef" is included, it will be changed to the default structure. This means that an endless amount of URLs can link back to the same article. Therefore, when an admin adds an article to a quiz using its URL, there is the risk of storing duplicates. To avoid this, we take the part after "/i" to find the article ID. Then we rebuild the URL to the format "https://smp.no/i/abcdef", only keeping the necessary information. This way we avoid storing duplicate articles in the database.

Questions can have images, but these are mainly for decoration. We warn admins that the questions themselves should not be based on images, because that causes issues with accessibility. Players who rely on screen readers would be unable to answer, unless we put the answer in the image's alternative text, which would also give an unfair advantage. This is not an issue when using AI to generate the questions, since it does not analyze the images either.

5.2.4 Play Quiz

The team initially created some wireframes of the "play quiz" page and presented them to Sunnmøreposten, but the design changed over time as we got feedback. One issue was that the timer was not obviously a timer. To fix that issue, we placed it more centered and made it larger. Additionally, we animate a gradient background, so it resembles an old fashioned countdown timer. However, the way it is implemented means that, as of May 2024, it does not have full support in all major browsers. Firefox users do not see the background animation, only the time decrementing. The team decided to stick with the implementation regardless, since the functionality still works and does not detract from the experience.

One concern was that everything should fit onto the screen at the same time, so that users will not be misled in case not all the alternatives are visible on the screen. Therefore, the design for mobile is much more compact than on wider screens.

We display a progress bar, so that players can see which question they are on and how many total questions are in the quiz. One issue with this was that the progress bar looks weird if it only has two questions, and if there are a lot of questions the progress bar becomes too long and will cause scrolling. Therefore, if the quiz has less than three or more than ten questions, it does not show the progress bar. Instead, it displays text such as "Question 1 of 2".

5.2.5 Guest Mode

Normally, users have to sign in in order to play quizzes. This is required so their progress can be: saved, associated with the user account and ranked against other users. To lower the barrier of entry, guest mode has been implemented.

Implementation of the guest mode differs from the default quiz playing mode due to the fact the latter relies heavily on the application state living in the database. For instance, in normal mode the time question was presented to the user is stored in the database (this is also discussed in terms of anti-cheating in section 5.1.4). Since guest user data is not stored, we decided to use the user browser's local storage and a bit of JavaScript. This way any necessary data is stored on the client, and it can be included with the requests sent to the server.

When deciding which quiz the guest users should be able to play, we considered the implications it may have on the rankings. If the currently active quizzes were available without signing in, users would be able to abuse the system. They could first view a quiz in guest mode and then answer while being signed in, gaining an unfair advantage.

Initially we considered having one predefined guest quiz, that would not change. It would showcase the quiz playing process itself, but questions would not necessarily be similar to ones actively being created by the administrators. This led us to showcase the latest quiz that is no longer active. This way the quiz is "fresh" and still relevant, while not exposing the currently active quizzes; preventing cheating.

5.2.6 Point System

As mentioned in section 4.2.5, the point system was iterated upon and changed significantly under development.

Initially the points rewarded was a simple set amount of points per correct answer. The amount of time used by the player for each question was ignored.

Later, the time between when the question was presented to the user and their answer time was accounted for. Initially, there were three stages with different score ratios when answering correctly. The user could answer quickly (within the first 25% of the question's time limit), in the middle range (within 50% of the time limit), or slowly (anything slower than previous thresholds). If the user answered quickly, they would be rewarded with 100% of the points for this question, slower meant 50% of points, and the lowest possible reward would be 25% of the maximum points. Note that the user is rewarded for answering correctly, even if they answer after the time limit has passed. Graph in figure 5.2.1 illustrates these time thresholds and point stages.

All following graphs in this section are representative for a question with the same configuration of maximum points and time limit; one hundred points and thirty seconds.

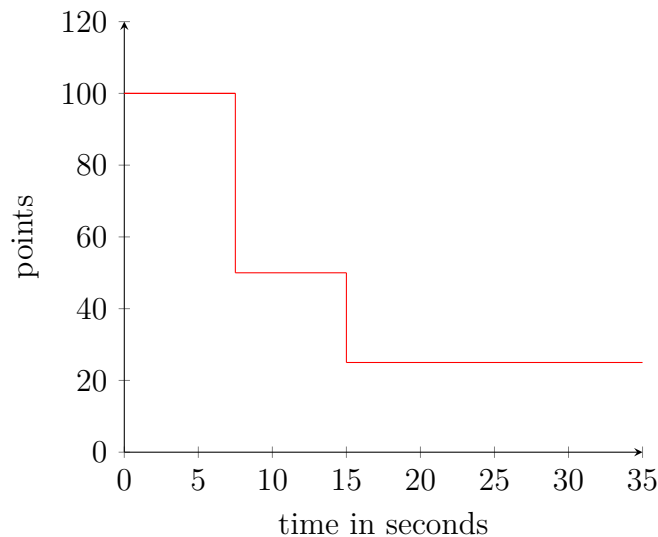


Figure 5.2.1: Second iteration of the point system. The range of points rewarded is limited to three constants.

The three stages with predefined ratios turned out to be rather lacking and resulted in little variations in the user results. We quickly moved on to points decreasing linearly as the time passes, becoming constant (20% of the maximum points) after reaching the question's time limit. This point system allowed user scores to be more spread out between the maximums and minimums. This resulted in more variety in user scores, decreasing the likelihood of many users ending up with the exact same number of points. This is illustrated in figure 5.2.2.

The line can be described as a linear function, intercepting the y axis at the maximum points value, as well as the minimum points value at time t equal to the question's time limit. The function can be expressed within the domain

$[0, time_{limit}]$. Any answer with t greater than $time_{limit}$ is rewarded with a constant number of points equal to $points_{min}$.

$$f(t) = \frac{points_{min} - points_{max}}{time_{limit}}t + points_{max}$$

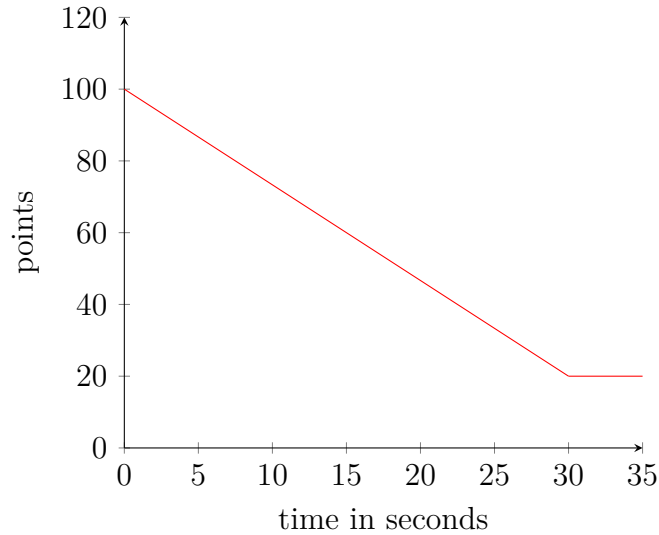


Figure 5.2.2: Third iteration of the point system. The value decrease linearly until it reaches the minimum (20% of the maximum points) at the question's time limit.

Linearly reducing the amount of points rewarded has proven to be more interesting and provided more variety in the user results. There was however a drawback with this approach; it was no longer possible to achieve a perfect score. This could be frustrating for players and in turn leave a bad impression of the game. To solve this, a grace period was added. A question answered within the first few seconds would be rewarded with the maximum points.

The inclusion of a grace period in the reward calculation slightly complicated the linear function expression, because the line no longer intersects the y axis at the maximum points value. However, the two intersection points are still known. The final implementation, it is described in detail in the results section 4.2.5.

As the function for reward calculation evolved over time, so did the way the points were stored in the database. Initially the rewarded points were stored directly in the database, alongside the answer picked by the user. The team quickly realised this approach was not ideal - it denormalized the database unintentionally. This denormalization led to the database storing redundant data and could lead to reduced data integrity. Additionally it restricted the point system in such way that if an administrator updated the maximum points for a question after some users answered it, the change would not affect them, only new users. This in turn would be unfair.

All data required to calculate the reward was already stored in the database. The points can be calculated at any point after a question is answered. For this,

a PostgreSQL function is defined. This function can be used in SQL queries or views, simplifying the points retrieval.

5.2.7 Limitations of HTMX

While HTMX has many advantages, such as making it incredibly easy to send HTTP requests, there are some limitations.

One such issue was error handling. If you want a specific behaviour to occur on an error, instead of the default HTMX behaviour, you would either need to write specific JS function, or use an HTMX extension. In our case we chose to use the *response-target* extension.

```

1   <button
2       hx-get="/api/images?id=5"
3       hx-swap="outerHTML"
4       hx-target="image-wrapper"
5       hx-target-error="next .error-text"
6   >Get Image</button>

```

Code 5.1: Example of how "hx-target-error" can be set up in a button.

As seen in figure 5.1, "hx-target-error" allows for handling HTTP codes that "hx-target" doesn't support, as "hx-target" only supports 200 OK. If you only wanted to handle 4xx error codes, you could do "hx-target-4*" to only run when a response is 4xx. If you wanted a specific target in case of HTTP code, such as 404, you can use "hx-target-404".

Another issue with HTMX is the inability to change the "hx-swap" target on error, such as if you want to target "innerHTML" instead of "outerHTML" when you get error codes. To solve this, one possibility is to set the HTTP response header to include "HX-Reswap" as shown below.

```

1   context.Response().Header().Set("HX-Reswap", "innerHTML")

```

Code 5.2: Example of setting the response header in Go's Echo framework

This unfortunately creates coupling between the API handlers and the view, as the controller now needs to know more about the view implementation, to set the appropriate swap behaviour.

During development, we also encountered what we believe is a bug. If "hx-target" is set to an element inside an "dialog" element, then the "hx-target-error" no longer works as intended. If we target another element outside the "dialog" in the HTML, then set "HX-Retarget" in the HTTP response header to target the correct element, it works as intended. This was the solution we went with, but it couples the API handler to the view.

5.2.8 Code Quality

A notable feature of *SonarLint* is its measure of cognitive complexity. This metric quantifies the difficulty a developer might face when reading a function. During development, one issue encountered with cognitive complexity was its generic nature. As *SonarLint* supports multiple languages, the thresholds for cognitive complexity must be tailored to each language.

In the case of *Go*, error handling increases the cognitive complexity by a varying amount. This means that handling errors five times in a function could potentially push the cognitive complexity value over the default limit of 15 by itself.

5.3 Reflection

The team was largely satisfied with the development and the finished product. It was a great collaborative process, both within the team and together with Sunnmøreposten. However, there are a few things that we realized could have been done differently in hindsight.

Due to the issues with HTMX, as discussed in 5.2.7, we should have done error handling differently. It would have been simpler to not use the "response targets" extension, and rather have used a single JavaScript function to catch all errors and display an alert. This alert could have a fixed position on the page, avoiding layout shifts and ensuring it is always visible when an error was sent. If there was a button to dismiss the alert or it disappeared after a certain time, we would not have to worry about clearing errors on successful requests either. It would have decoupled the user interface from the server.

5.4 Future Work

5.4.1 Gamification

A potential feature to reward consistent playing is an "early-bird" bonus, as an alternative to the "daily streak" bonus. If a quiz is played within the first 24 hours of its release, the player will get bonus points.

Avatars was a feature that was considered, but not implemented due to time constraint. During the prototype phase, discussions arose regarding its potential inclusion. Several ideas were proposed, such as the use of points or an alternative currency earned through quiz performance to purchase cosmetics. This could foster competitiveness and engagement, causing increased activity.

"Achievements" or "Badges" were considered already during the information collection phase. These badges could be earned for meeting certain achievements such as a perfect quiz, get to top 10 on a leaderboard, or for certain point thresholds. These could potentially be implemented together with the avatar system, so certain cosmetics would be earned by completing different achievements. While the potential was interesting, it was not prioritized due to the time constraint.

5.4.2 Notifications

Since Sunnmøreposten expressed a desire for the quiz to be integrated into their mobile app, a notification system could be interesting. A notification can be sent whenever a new quiz is available. It can be used to make users aware of new quizzes being published. An alternative for Web users is to have them opt-in to receive an e-mail that is sent when quizzes are published.

5.4.3 User Authentication and Verification

In order to read some of the articles at Sunnmøreposten (SMP), users need a subscription that is linked to their *Schibsted* account. For this reason, in the earliest stages of the project we assumed we could use *Schibsted* as the identity provider for the application. This however was not ideal. Firstly, SMP was not able to grant us access to use their *Schibsted* APIs. And secondly, SMP expressed that a lot of their readers don't necessarily use their own accounts to read their articles. It's quite common for people to use an account belonging to a family member. Thus, they said it would be nice if a different sign in method would be used.

Users can sign in using their *Google* accounts, this has several benefits. First of all it is a convenient way of signing in. Most people have a *Google* account from before, and our application does not become yet another app you need to remember a password to. Additionally, we can trust that the e-mail provided is verified.

Using a trusted third party identity provider also has advantages in terms of security. We don't need to worry about securely storing and managing user secrets (passwords), or implement extra security measures like multi-factor authentication. We trust *Google* to do a better job of it, since they are a large organization with dedicated security teams.

Early in the project the stakeholders expressed desire to get verified mobile phone numbers of users. This is so they could easily contact users in order to award them.

For this we considered implementing our own mobile phone verification. This would require us to use a SMS gateway service, to send verification messages. However, these services tend to be costly. Briefly we considered implementing a fake/mock service that would not actually send messages. In the end we decided that implementing this is out of the project scope.

We proposed to SMP that *Vipps* could be used as identity provider, instead of *Google*. They would provide us with a verified phone number. This has additional benefit; most people have just one *Vipps* account, and contrary to *Google*, it is not as easy to have multiple accounts. This would counteract cheating options that were discussed in context of guest mode (section 5.2.5). The stakeholders were happy with this suggestion, and this stands as a possible future work.

5.4.4 Exclusive Leaderboards

Quizzes belong to different leaderboards based on "labels" assigned to a quiz. A possible feature could be "limited access" labels, in which quizzes belonging to

certain labels are only accessible to users with specific roles. A use case of this would be for local competitions such as school quizzes, where students of a class compete against each other.

5.4.5 Bucket

Currently, old images are kept in the bucket, even after they are no longer used anywhere. It would be ideal if we had a routine running that removed unused images, i.e. images that are no longer referenced in the database.

CONCLUSIONS

We have achieved a highly functional quiz application, which is both responsive, and accessible for persons with disabilities. The application style was designed with the target group in mind. It also features an administration panel and leaderboards, fulfilling the initial requirements provided by SMP.

Additional functionality was also added and improved upon. Questions are created with ChatGPT. SMP can manage predefined usernames. It has support for multiple leaderboards based on labels, which also open up the possibility for custom leaderboards independent of months, years, etc.

The regular meetings between the team and the stakeholder were productive and useful in communicating their expectations and our progress and ideas.

Sunnmøreposten is satisfied with the final product and seems interested in further development based on the project's source code.

The group has learnt much from this project. We have become much more familiar with Go and HTMX, which the group had minimal to no experience with beforehand. We also used knowledge learnt from the NTNU courses in every aspect of the project.

There are features that could be implemented in the future, but was not prioritized due to the time constraint. Ultimately, the team wanted to deliver a polished project and is happy with the final state of the product.

REFERENCES

- [1] Gudleiv Forr et al. *Sunnmørsposten i Store norske leksikon*. <https://snl.no/Sunnmørsposten>. Apr. 10, 2024. (Visited on 05/10/2024).
- [2] Hanna Relling Berg. “Her er vår redaksjonelle årsrapport”. In: *Sunnmørsposten* (Feb. 22, 2024). (Visited on 05/01/2024).
- [3] Wikipedia contributors. *Mass media — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Mass_media&oldid=1210168945. 2024. (Visited on 02/28/2024).
- [4] David Shedden. *News Media Timeline (1980)*. <https://www.poynter.org/archive/2004/new-media-timeline-1980/>. Dec. 16, 2004. (Visited on 04/18/2024).
- [5] Wikipedia. *Nettavis — Wikipedia*. <https://no.wikipedia.org/w/index.php?title=Nettavis&oldid=21301004>. 2021.
- [6] Statistisk sentralbyrå. *11556: Andel som har brukt tradisjonelle medier og internettmedier en gjennomsnittsdag, etter medietype, statistikkvariabel og år*. URL: <https://www.ssb.no/statbank/table/11556/tableViewLayout1/> (visited on 04/18/2024).
- [7] Carlo Prato. *How can publishers use games and puzzles to increase subscribers?* <https://www.twipemobile.com/how-can-publishers-use-games-and-puzzles-to-increase-subscribers/>. Oct. 11, 2023. (Visited on 05/14/2024).
- [8] Wikipedia contributors. *Wordle — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Wordle&oldid=1218968989>. 2024. (Visited on 04/24/2024).
- [9] Marc Tracy. *The New York Times Buys Wordle*. <https://www.nytimes.com/2022/01/31/business/media/new-york-times-wordle.html>. Jan. 31, 2022. (Visited on 04/25/2024).
- [10] Jacqueline Zenn. *Designing Habit-Forming Games*. <https://gameanalytics.com/blog/designing-habit-forming-games/>. Mar. 27, 2018. (Visited on 05/10/2024).
- [11] Oxford English Dictionary. *quiz*. https://www.oed.com/dictionary/quiz_n?tab=meaning_and_use. (Visited on 05/14/2024).

- [12] Sebastian Detering. “The Lens of Intrinsic Skill Atoms: A Method for Gameful Design”. In: *Human–Computer Interaction* (June 15, 2015). (Visited on 02/08/2024).
- [13] Christopher Zichermann Gabe & Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. O’Reilly Media, Inc, 2011. (Visited on 02/08/2024).
- [14] Ashton Anderson et al. “Steering user behavior with badges”. In: *Proceedings of the 22nd International Conference on World Wide Web*. WWW ’13. Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 95–106. ISBN: 9781450320351. DOI: 10.1145/2488388.2488398. (Visited on 02/08/2024).
- [15] Richard N. Landers and Amy K. Landers. “An Empirical Test of the Theory of Gamified Learning: The Effect of Leaderboards on Time-on-Task and Academic Performance”. In: *Simulation & Gaming* 45.6 (2014), pp. 769–785. DOI: 10.1177/1046878114563662. eprint: <https://doi.org/10.1177/1046878114563662>. (Visited on 02/08/2024).
- [16] Leonard A. Annetta. “The “I’s” Have It: A Framework for Serious Educational Game Design”. In: *Review of General Psychology* 14.2 (2010), pp. 105–113. DOI: 10.1037/a0018985. eprint: <https://doi.org/10.1037/a0018985>. (Visited on 02/08/2024).
- [17] A. Spanellis and J.T. Harviainen. *Transforming Society and Organizations through Gamification: From the Sustainable Development Goals to Inclusive Workplaces*. Springer International Publishing, 2021. ISBN: 9783030682071. URL: <https://books.google.no/books?id=STItEAAAQBAJ>.
- [18] Dominic Macbean. *Experimenting with Answer Streaks to Help Make Learning Awesome*. <https://medium.com/inside-kahoot/experimenting-with-answer-streaks-to-help-make-learning-awesome-3b3357e42595>. June 10, 2016. (Visited on 02/08/2024).
- [19] Josh Ye. *China announces rules to reduce spending on video games*. <https://www.reuters.com/world/china/china-issues-draft-rules-online-game-management-2023-12-22/>. Dec. 22, 2023. (Visited on 02/08/2024).
- [20] Wikipedia contributors. *Visual design elements and principles — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Visual_design_elements_and_principles&oldid=1207220517. 2024. (Visited on 02/27/2024).
- [21] Sachin Rekhi. *Don Norman’s Principles of Interaction Design*. <https://medium.com/@sachinrekhi/don-normans-principles-of-interaction-design-51025a2c0f33>. Jan. 23, 2017. (Visited on 02/27/2024).
- [22] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. <https://www.nngroup.com/articles/ten-usability-heuristics/>. Jan. 30, 2024. (Visited on 05/14/2024).
- [23] Wikipedia contributors. *Website wireframe — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Website_wireframe&oldid=1186560773. 2023. (Visited on 02/05/2024).


- [24] Wikipedia contributors. *Congenital red–green color blindness* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Congenital_red%E2%80%93green_color_blindness&oldid=1214525131. 2024. (Visited on 03/21/2024).
- [25] Wikipedia contributors. *Visual impairment* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Visual_impairment&oldid=1214270098. 2024. (Visited on 03/21/2024).
- [26] MDN contributors. *ARIA - Aaccessibility*. <https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA>. Mar. 15, 2024. (Visited on 03/21/2024).
- [27] MDN contributors. *prefers-reduced-motion*. <https://developer.mozilla.org/en-US/docs/Web/CSS/@media/prefers-reduced-motion>. Aug. 21, 2023. (Visited on 03/21/2024).
- [28] Wikipedia contributors. *Web Content Accessibility Guidelines* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Web_Content_Accessibility_Guidelines&oldid=1217022216. 2024. (Visited on 04/03/2024).
- [29] *Web Content Accessibility Guidelines*. <https://wcag.com/blog/web-content-accessibility-guidelines-wcag-by-the-numbers/>. Feb. 24, 2022. (Visited on 04/03/2024).
- [30] Tilsynet for universell utforming av ikt. *WCAG-standarder*. <https://www.uutilsynet.no/wcag-standarder/wcag-standarder/86>. (Visited on 04/03/2024).
- [31] *What's the Difference Between Monolithic and Microservices Architecture?* <https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/>. (Visited on 03/21/2024).
- [32] GeeksforGeeks. *MVC Design Pattern*. <https://www.geeksforgeeks.org/mvc-design-pattern/>. Feb. 19, 2024. (Visited on 03/21/2024).
- [33] baeldung. *The DTO Pattern (Data Transfer Object)*. <https://www.baeldung.com/java-dto-pattern>. Jan. 8, 2024. (Visited on 03/29/2024).
- [34] Martin Fowler. *Data Transfer Object*. <https://martinfowler.com/eaCatalog/dataTransferObject.html>. Jan. 2023. (Visited on 03/29/2024).
- [35] Atlassian. *What is version control?* <https://www.atlassian.com/git/tutorials/what-is-version-control>. 2020. (Visited on 01/24/2024).
- [36] Wikipedia contributors. *Version control* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Version_control&oldid=1192261201. 2023. (Visited on 01/24/2024).
- [37] Gitlab. *What is version control?* <https://about.gitlab.com/topics/version-control/>. 2023. (Visited on 01/24/2024).
- [38] Wikipedia contributors. *Git* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Git&oldid=1193006801>. 2024. (Visited on 01/24/2024).
- [39] Conventional Commits. *Conventional Commits*. <https://www.conventionalcommits.org/en/v1.0.0/>. (Visited on 02/08/2024).

- [40] Kent Beck et al. *Agile manifesto*. <https://agilemanifesto.org/>. 2001. (Visited on 03/24/2024).
- [41] Wikipedia contributors. *DevOps — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=DevOps&oldid=1216062177>. 2024. (Visited on 03/27/2024).
- [42] Sonar. *code quality developer's guide*. <https://www.sonarsource.com/learn/code-quality/>. (Visited on 03/29/2024).
- [43] Sonar. *linter developer's guide*. <https://www.sonarsource.com/learn/linter/>. (Visited on 03/29/2024).
- [44] Nicholas C. Zakas. *Why Coding Style Matters*. <https://www.smashingmagazine.com/2012/10/why-coding-style-matters/>. Oct. 25, 2012. (Visited on 05/14/2024).
- [45] Ganesh Pagade. *Difference Between Cohesion and Coupling*. <https://www.baeldung.com/cs/cohesion-vs-coupling>. Nov. 9, 2022. (Visited on 03/30/2024).
- [46] Ganesh Pagade. *Refactoring*. <https://refactoring.com/>. (Visited on 03/30/2024).
- [47] Wikipedia contributors. *Code refactoring — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Code_refactoring&oldid=1215784820. 2024. (Visited on 03/30/2024).
- [48] Gitlab. *What is a code review?* <https://about.gitlab.com/topics/version-control/what-is-code-review/>. 2023. (Visited on 02/27/2024).
- [49] IBM. *What is software testing?* <https://www.ibm.com/topics/software-testing>. 2024. (Visited on 03/30/2024).
- [50] AWS. *What is Unit Testing?* <https://aws.amazon.com/what-is/unit-testing/>. (Visited on 03/30/2024).
- [51] Ham Vocke. *The Practical Test Pyramid*. <https://martinfowler.com/articles/practical-test-pyramid.html>. Feb. 26, 2018. (Visited on 05/03/2024).
- [52] Wikipedia contributors. *Functional testing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Functional_testing&oldid=1215853964. 2024. (Visited on 03/31/2024).
- [53] Kate Moran. *Usability Testing 101*. <https://www.nngroup.com/articles/usability-testing-101/>. Dec. 1, 2019. (Visited on 03/31/2024).
- [54] Wikipedia contributors. *Exploratory testing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Exploratory_testing&oldid=1215212435. 2024. (Visited on 05/03/2024).
- [55] Martin Fowler. *Test Pyramid*. <https://martinfowler.com/tags/testing.html>. May 1, 2012. (Visited on 05/02/2024).
- [56] Martin Fowler. *On the Diverse And Fantastical Shapes of Testing*. <https://martinfowler.com/articles/2021-test-shapes.html>. June 2, 2021. (Visited on 05/03/2024).

- [57] Tim Bray. *Testing in the Twenties*. <https://www.tbray.org/ongoing/When/202x/2021/05/15/Testing-in-2021>. May 15, 2021. (Visited on 05/03/2024).
- [58] Raunak Jain. *Unit Testing Vs Integration Testing – Important Differences*. <https://testsigma.com/blog/unit-test-vs-integration-test/>. Mar. 6, 2024. (Visited on 05/03/2024).
- [59] Kent C. Dodds. *The Merits of Mocking*. <https://kentcdodds.com/blog/the-merits-of-mocking>. Nov. 5, 2018. (Visited on 05/03/2024).
- [60] Mozilla Foundation. *HTTP*. Oct. 25, 2023. (Visited on 03/03/2024).
- [61] Mozilla Foundation. *An overview of HTTP*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>. Dec. 16, 2023. (Visited on 03/21/2024).
- [62] Mozilla Foundation. *HTTP request methods*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. Apr. 10, 2023. (Visited on 03/21/2024).
- [63] R Fielding et al. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. <https://www.ietf.org/rfc/rfc2616.txt>. June 1999. (Visited on 03/21/2024).
- [64] R Fielding et al. *RFC 9110 - HTTP Semantics*. <https://httpwg.org/specs/rfc9110.html#overview.of.status.codes>. July 2022. (Visited on 03/24/2024).
- [65] Colin Walls. *Embedded Software: The Works*. Newnes, 2005. ISBN: 0-7506-7954-9.
- [66] MDN contributors. *HTML: HyperText Markup Language*. <https://developer.mozilla.org/en-US/docs/Web/HTML>. (Visited on 02/08/2024).
- [67] Wikipedia contributors. *HTML — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=HTML&oldid=1203886289>. 2024. (Visited on 02/16/2024).
- [68] Wikipedia contributors. *CSS — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=CSS&oldid=1207061972>. 2024. (Visited on 02/16/2024).
- [69] MDN contributors. *<iframe>: The Inline Frame element*. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>. Feb. 28, 2024. (Visited on 05/15/2024).
- [70] Wikipedia contributors. *JavaScript — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1207507795>. 2024. (Visited on 02/16/2024).
- [71] Anuj Tomar. *10 Best Web Development Frameworks to Use in 2024 [Updated]*. https://medium.com/@anuj.t_8752/10-best-web-development-frameworks-to-use-in-2024-updated-ff988e8f85cb. Dec. 4, 2023. (Visited on 02/16/2024).
- [72] *Build fast, responsive sites with Bootstrap*. <https://getbootstrap.com/>. (Visited on 02/24/2024).

- [73] Tailwind Labs. *Get started with Tailwind CSS*. <https://tailwindcss.com/docs/installation>. (Visited on 02/24/2024).
- [74] Scott Gary. *Server Side Rendering in JavaScript – SSR vs CSR Explained*. <https://www.freecodecamp.org/news/server-side-rendering-javascript/>. (Visited on 03/19/2024).
- [75] Cloudinary. *Server Side Rendering: Benefits, Use Cases, and Best Practices*. <https://cloudinary.com/guides/automatic-image-cropping/server-side-rendering-benefits-use-cases-and-best-practices>. Nov. 24, 2023. (Visited on 03/19/2024).
- [76] IBM. *What is a REST API?* <https://www.ibm.com/topics/rest-apis>. (Visited on 03/21/2024).
- [77] Wikipedia contributors. *REST — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=REST&oldid=1211950882>. 2024. (Visited on 03/21/2024).
- [78] Kurt Nørman. *Overview of the four main programming paradigms*. https://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html. July 2, 2013. (Visited on 02/09/2024).
- [79] Mozilla Foundation. *Multi-Paradigm Programming Language*. <https://web.archive.org/web/20130821052407/https://developer.mozilla.org/en-US/docs/multiparadigmlanguage.html>. June 21, 2013. (Visited on 03/01/2024).
- [80] J W Lloyd. *Declarative Programming in Escher*. English. WorkingPaper. Other: CSTR-95-013. Department of Computer Science, University of Bristol, 1995. URL: <https://web.archive.org/web/20240301213649/https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=937eafaca5b54a619d1a51c4d7a3a856ac99c187> (visited on 03/01/2024).
- [81] David J. Barnes and Michael Kölling. *Objects First with Java*. Pearson, 2017. ISBN: 978-1-292-15904-1.
- [82] J. Hughes. “Why Functional Programming Matters”. In: *Computer Journal* 32.2 (1989), pp. 98–107. URL: <https://www.cse.chalmers.se/~rjmh/Papers/whyfp.html> (visited on 03/01/2024).
- [83] Wikipedia contributors. *Database — Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Database&oldid=1221791353>. 2024. (Visited on 05/03/2024).
- [84] Wikipedia contributors. *Relational database — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Relational_database&oldid=1217554613. 2024. (Visited on 05/03/2024).
- [85] IBM. *What is a database management system?* <https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-what-is-database-management-system>. (Visited on 05/03/2024).
- [86] IBM. *Structured Query Language (SQL)*. <https://www.ibm.com/docs/en/db2/11.5?topic=fundamentals-sql>. (Visited on 05/04/2024).

- [87] Wikipedia contributors. *SQL* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=SQL&oldid=1221268124>. 2024. (Visited on 05/03/2024).
- [88] Martin Fowler. *Domain Logic and SQL*. <https://martinfowler.com/articles/dblogic.html>. Feb. 2003. (Visited on 05/05/2024).
- [89] Wikipedia contributors. *Database normalization* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Database_normalization&oldid=1220845896. 2024. (Visited on 05/05/2024).
- [90] Wikipedia contributors. *Denormalization* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Denormalization&oldid=1041768115>. 2021. (Visited on 05/14/2024).
- [91] Instagram Engineering. *Instagrator Pt. 2: Scaling our infrastructure to multiple data centers*. <https://instagram-engineering.com/instagrator-pt-2-scaling-our-infrastructure-to-multiple-data-centers-5745cbad7834>. Nov. 11, 2015. (Visited on 05/05/2024).
- [92] Wikipedia contributors. *Database seeding* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Database_seeding&oldid=1205131562. 2024. (Visited on 05/05/2024).
- [93] Prisma contributors. *What are database migrations?* <https://github.com/prisma/dataguide/blob/23c2469a2813daf0da2e4eb57c10d8bf137beb0b/content/03-types/02-relational/03-what-are-database-migrations.mdx#L2>. (Visited on 05/14/2024).
- [94] Wikipedia contributors. *Public-key cryptography* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Public-key_cryptography&oldid=1220802772. 2024. (Visited on 05/01/2024).
- [95] Wikipedia contributors. *Authentication* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=Authentication&oldid=1216154985>. 2024. (Visited on 04/01/2024).
- [96] Wikipedia contributors. *SQL injection* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=SQL_injection&oldid=1219894917. 2024. (Visited on 04/21/2024).
- [97] *Using prepared statements*. [urlhttps://go.dev/doc/database/prepared-statements](https://go.dev/doc/database/prepared-statements). (Visited on 05/10/2024).
- [98] Wikipedia contributors. *Cross-site scripting* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Cross-site_scripting&oldid=1216463548. 2024. (Visited on 04/21/2024).
- [99] PortSwigger. *Cross-site scripting*. <https://portswigger.net/web-security/cross-site-scripting>. (Visited on 05/14/2024).
- [100] Wikipedia contributors. *Virtual machine* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=1214952220. 2024. (Visited on 04/01/2024).
- [101] Wikipedia contributors. *OS-level virtualization* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=OS-level_virtualization&oldid=1211197903. 2024. (Visited on 04/01/2024).

- [102] Ragna Marie Tørdal. *Bruk av kunstig intelligens i journalistikk*. <https://ndla.no/subject:1:576cc40f-cc74-4418-9721-9b15ffd29cff/topic:2:9cb15fe1-6e3d-4698-99c1-62165405f278/topic:1:41abfef8-4bc2-468d-a3d6-55c6def45d0d/resource:24560371-eddb-4986-8465-1e2d579439bb>. Dec. 5, 2023. (Visited on 05/14/2024).
- [103] Inc. Cloudflare. *What is a large language model (LLM)?* <https://www.cloudflare.com/learning/ai/what-is-large-language-model/>. (Visited on 04/30/2024).
- [104] IBM. *UML models and diagrams*. <https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-uml-models>. Sept. 21, 2023. (Visited on 03/27/2024).
- [105] Figma. *Figma: The Collaborative Interface Design Tool*. <https://www.figma.com/>. (Visited on 02/07/2024).
- [106] Wikipedia contributors. *Docker (software)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Docker_\(software\)&oldid=1209712745](https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1209712745). 2024. (Visited on 03/03/2024).
- [107] Wikipedia contributors. *Go (programming language)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Go_\(programming_language\)&oldid=1215962053](https://en.wikipedia.org/w/index.php?title=Go_(programming_language)&oldid=1215962053). 2024. (Visited on 04/01/2024).
- [108] *context package - context - Go Packages*. [urlhttps://pkg.go.dev/context](https://pkg.go.dev/context). (Visited on 05/10/2024).
- [109] *Echo*. <https://github.com/labstack/echo>. (Visited on 05/07/2024).
- [110] osmtrek & Contributors.  *Air - Live reload for Go apps*. <https://github.com/cosmtrek/air>. (Visited on 02/02/2024).
- [111] Adrian Hesketh & Contributors. *A HTML templating language for Go that has great developer tooling*. <https://github.com/a-h/templ>. (Visited on 02/02/2024).
- [112] *Security*. <https://templ.guide/security/>. (Visited on 05/08/2024).
- [113] HTMX contributors. *HTMX Documentation*. <https://htmx.org/docs/>. Apr. 25, 2024. (Visited on 05/05/2024).
- [114] *SortableJS*. <https://sortablejs.github.io/Sortable/>. (Visited on 05/15/2024).
- [115] *Odometer*. <https://github.hubspot.com/odometer/docs/welcome/>. (Visited on 05/15/2024).
- [116] Tailwind Labs. *Tailwindcss*. <https://tailwindcss.com/>. (Visited on 05/15/2024).
- [117] Inc Free Software Foundation. *GNU make*. <https://www.gnu.org/software/make/manual/make.html>. (Visited on 02/08/2024).
- [118] *About PostgreSQL*. <https://www.postgresql.org/about/>. (Visited on 04/01/2024).

- [119] Wikipedia contributors. *MinIO* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=MinIO&oldid=1220098394>. 2024. (Visited on 05/07/2024).
- [120] *Caddy - The Ultimate Server with Automatic HTTPS*. <https://caddyserver.com/>. 2024. (Visited on 05/15/2024).
- [121] Wikipedia contributors. *Caddy (web server)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Caddy_\(web_server\)&oldid=1202934778](https://en.wikipedia.org/w/index.php?title=Caddy_(web_server)&oldid=1202934778). 2024. (Visited on 05/07/2024).
- [122] Inc. Github. *The world's most widely adopted AI developer tool*. URL: <https://github.com/features/copilot> (visited on 04/22/2024).
- [123] Wikipedia contributors. *ChatGPT* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=ChatGPT&oldid=1219989079>. 2024. (Visited on 04/21/2024).
- [124] Lauren Farrell. *What is Google Lighthouse and How Can it Improve Website UX?* <https://blog.hubspot.com/website/google-lighthouse>. Dec. 13, 2023. (Visited on 04/11/2024).
- [125] Go contributors. *Add a test*. <https://go.dev/doc/tutorial/add-a-test>. Feb. 14, 2023. (Visited on 05/05/2024).
- [126] *Welcome to Testcontainers for Go!* <https://golang.testcontainers.org/>. (Visited on 05/05/2024).
- [127] *Testify - Thou Shalt Write Tests*. <https://github.com/stretchr/testify>. (Visited on 05/05/2024).
- [128] Anoop. *Re-Inventing the API Client*. <https://www.usebruno.com/>. (Visited on 02/22/2024).
- [129] *Standard Go Project Layout*. <https://github.com/golang-standards/project-layout>. (Visited on 05/08/2024).
- [130] Maria Elsness. “Derfor dukker «utro sild» opp i nettbanken din”. In: *NRK* (Dec. 6, 2016). URL: <https://www.nrk.no/livsstil/derfor-dukker-utro-sild-opp-i-nettbanken-din-1.13254442> (visited on 05/01/2024).

APPENDICES

GITHUB REPOSITORY

All code used in this document are included in the GitHub repository linked below. Further instructions on how to run the application are given in the readme-file.

- <https://github.com/Molnes/Nyhetsjeger>

DIAGRAMS

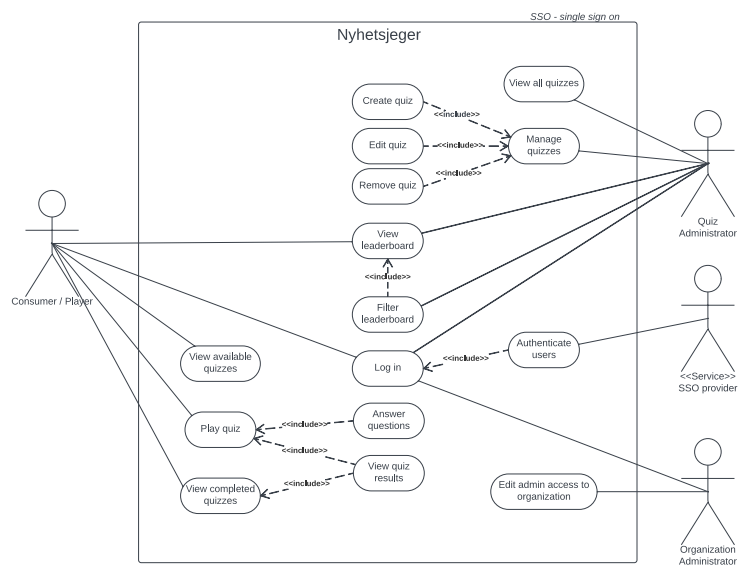


Figure B.0.1: Use case diagram of the Nyhetsjeger portal.

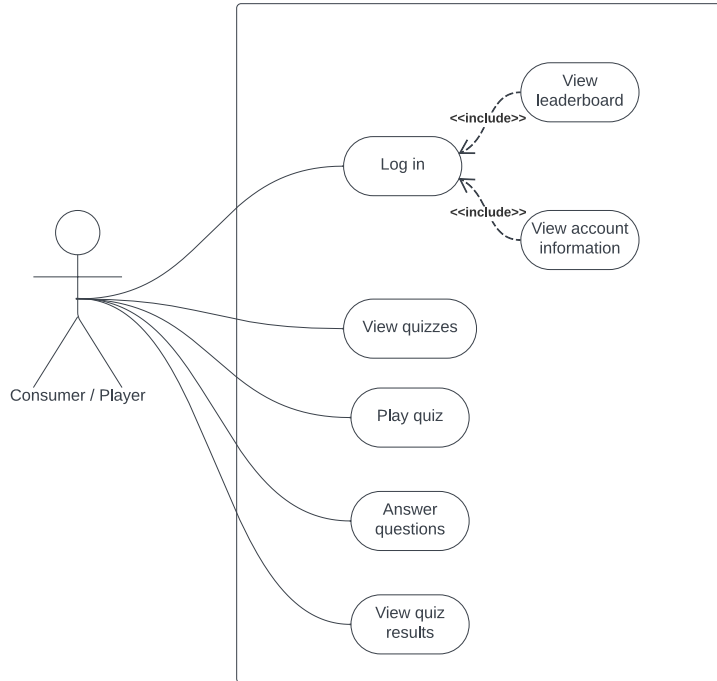


Figure B.0.2: Use case diagram of the player.

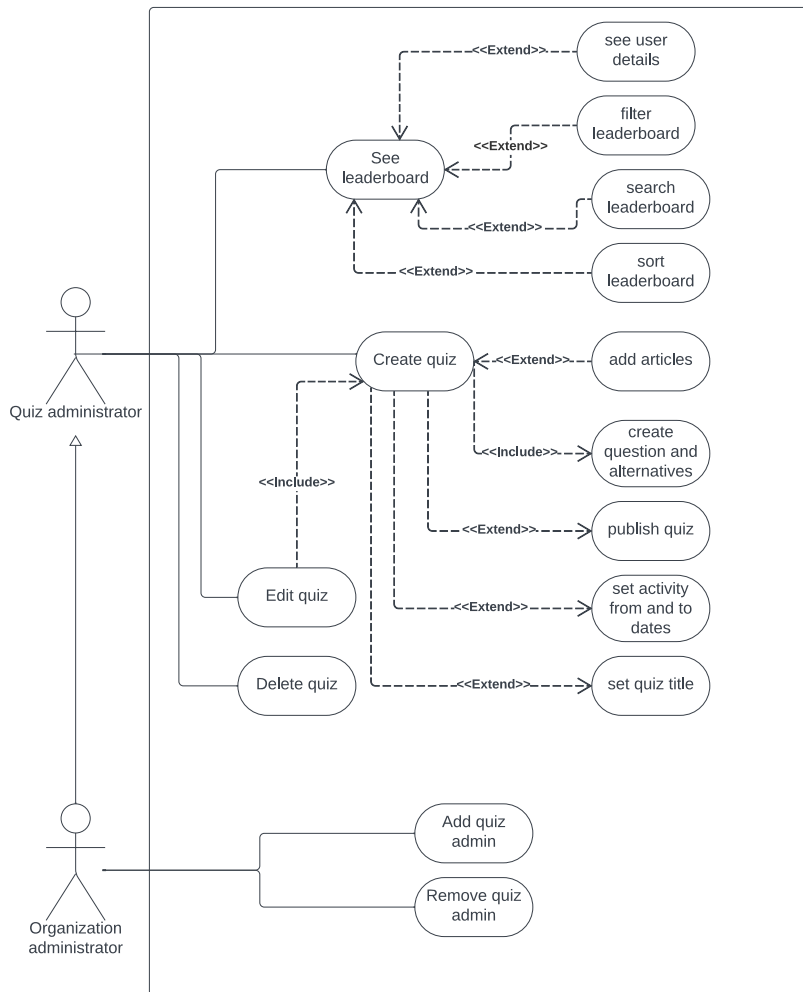


Figure B.0.3: Use case diagram of the administration.

USER STORIES

- As an admin, I want to generate suggestions for questions based on a list of articles, because this will speed up the process.
- As an admin, I want to be able to go to the dashboard from the quiz page and vice versa, because this will make navigation easy.
- As a user, I want to see my progress, so I know how much I have done
- As an admin, I want to be able to upload images to some questions to make them look more interesting.
- As a user, I want to create a username, for privacy.
- As a user, I want music, so that the experience feel more entertaining.
- As a user, I want to get bonus points multiplier depending on my daily login streak (consecutive quiz completions), so that I am rewarded for my eagerness.
- As a user, I want my answers to be timed, so that I am rewarded for being quick.
- As an admin, I want a warning whenever the amount of questions is outside of a soft limit, so that I am reminded of the ideal amount of questions.
- As an admin, I want to whitelist articles, so I can specify which articles the questions should be generated from.
- As a user, I want to report issues I experience, so that they can be fixed.
- As an admin, I want to be able to receive bug reports from users, so that I am aware of the problems and can fix them.
- As a user, I want to see the distribution of answers, since it is fun to see my performance compared to others.
- As a user, I want to be able to see the leaderboard, so that I can see if I am winning.

- As a user, I want to be able to see all available quizzes, so that I can participate in them.
- As a user, I want to be able to opt in and out of the leaderboards, because I want to keep my privacy.
- As a user, I want to be able to see my summary after finishing a quiz, so I know how well I did.
- As a user, I want to see which quizzes I have completed earlier, so that I know which quizzes I can't improve my score on.
- As a user, I want to be able to see which answer that is selected, so that I know which alternative I selected.
- As a user, I want to see the question and alternatives so that I can participate in the quiz.
- As a user, I want to log in to my Schibsted/SMP account to participate in the leaderboards and save my progress.
- As a user, I want to see what the prizes are so that I know what I am opting in on.
- As an admin, I want to see who got the top score and their information this month so that I can give them their prize.
- As a journalist, I want to create questions for the quiz so that I can use the quiz in my article.
- As a user, I want to be able to see my points during and at the end of the quiz, so that I know how well I am doing.

