Institutt for Informasjonsteknologi
Postadresse: Postboks 4 St. Olavs plass,
0130 Oslo
Besøksadresse: Holbergs plass, Oslo

PROSJEKT NR.
00-10

TILGJENGELIGHET
Åpen

Telefon: 22 45 32 00

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
Livsmestring app	**23.05.24**
	ANTALL SIDER / BILAG 148
PROSJEKTDELTAKERE **Alan Blanco Alquézar (139) - s354528** **Oleksandra Chornous (142) - s349584** **Petter Halsne (176) - s320947** **Sari Siraj Abdalla Ali (109) - s336154**	INTERN VEILEDER **Way Kiat Bong**

OPPDRAGSGIVER	KONTAKTPERSON
Oslo Voksenopplæring avdeling Helsfyr	**Lisa Sveen Larsen**

SAMMENDRAG

Denne bacheloravhandlingen handler om utviklingen av Livsmestring applikasjonen på vegne av Oslo Voksenopplæring avdeling Helsfyr. Applikasjonen er utviklet for mobile enheter og er rettet mot å tas i bruk i undervisningssituasjoner for elever med innvandringsbakgrunn i voksenopplæringen. Applikasjonen inneholder brukergrensesnitt på en rekke ulike språk som omhandler videoavspilling av temaer knyttet til områder som karriere og helse i det norske samfunnet. I tillegg til å se videoer på ønsket språk, kan brukeren også måle progresjonen innenfor de ulike områdene, uten å måtte opprette bruker.

3 STIKKORD

Flutter

Videoavspilling

Oversettelse

The Livsmestring App

Preface

This academic report covers the work undertaken for Oslo Metropolitan University (OsloMet) during the spring semester of 2024, culminating in our final bachelor project. Our decision-making process led us to select a project offered by *Utdanningsetaten*, which we believed closely aligned with our capabilities and values.

Upon reviewing the available projects, our group was immediately drawn to one in particular. The project demanded proficiency in the four most spoken languages within the *Utdanningsetaten* classes: Ukrainian, Russian, Arabic, and Tigrinya. As it turned out, our diverse team composition included skilled translators for these languages and more. In addition to that, this project's aim—to provide educational support to individuals from foreign backgrounds—resonated deeply with us, given our group's blend of international members and a dedicated teacher.

In October of 2023, we initiated contact with the project's coordinator, Lisa Sveen Larsen, to express our interest in the project. Our proposal was met positively, and following several discussions to assess compatibility, we were entrusted with the development of the *Livsmestringsapp* project.

We would like to thank our internal supervisor, Way Kiat Bong, whose guidance was essential in structuring our approach and navigating the project's technical complexities. We extend our gratitude to Lisa Sveen Larsen too, whose dedication and passion for the project's success were evident, and who constantly offered us support throughout the project.

Table of Contents

1. Introduction

1.1. The Project

In this section of the report, we will describe our project's problem statement and what the solution is. It will provide information about the client and briefly describe some ethical considerations in the project.

1.1.1. Problem Statement

Livsmestring—translated as life mastery in English—is a concept used by the Directorate of Integration and Diversity (IMDi, derived from *Integrerings- og mangfoldsdirektoratet*). It is designed to cover all of the skills needed to learn about life in a new country (IMDi, n.d.).

The concept of Livsmestring serves as a comprehensive framework for any course aimed at teaching immigrants about life in Norway. It outlines key components, including the legal requirements such as the definition of the content and the minimum duration for each topic. For example, at least 10 hours of guidance must be provided within the topics of area of migration, health, and diversity, as decreed by the Integration Act of 2023 (Lovdata, 2020). It also lays the foundation for what the goals of any life mastery course should be. Broadly speaking, a life mastery course aims to help students to:

- Enhance awareness of their own knowledge, skills, qualities, attitudes, and values.
- Gain more knowledge of opportunities and learn to assess opportunities and limitations in relation to work, education and participation in society.
- Make more informed and reflective choices related to education and work.
- Understand and adapt to varying circumstances.
- Effectively manage the transition from refugee to student, employee, or active community member.

These goals aim to optimize the quality of education provided to foreigners integrating into Norwegian society. As it was mentioned on the webpage of the Institute for social research, inclusive integration policies soften immigration scepticism (Christensen, 2024). In another project, they also noted the positive effects on economic integration in contrast to other forms of integration, which naturally is benefited from a good educational foundation.

These facts are important to pay attention to, because as it was underlined in *The Integration Barometer 2024*, more than 20% of the population in Norway had an immigration

background in 2023 (Brekke et al., 2024, p. 27). Furthermore, as of 2019, data from Statistics Norway (SSB–*Statistikk sentralbyrå*) indicates that 1 in 2 refugees and their families have low education levels, compared to the rates of 1 in 5 native born individuals (OECD, n.d.).

Despite the well-intentioned goals and efforts, the implementation of life mastery courses faces substantial challenges. One significant issue is the linguistic diversity of the participants, many of whom possess limited skills in Norwegian or English, which complicates their integration process. Additionally, the wide range of countries of origin among the course participants introduces another layer of complexity to the issue. This diversity not only makes it more challenging to secure skilled translators for each of the students, but also significantly increases the logistical and financial burdens associated with these efforts.

1.1.2. Problem Solution

In response to these challenges, *Oslo Voksenopplæring* (henceforth referred to as Oslo V.O.) and the Norwegian Education Agency (*Utdanningsetaten*) have collaborated for the past two years to develop the Livsmestring project. This initiative consists of two primary objectives:

- The implementation of a course directed towards the integration of foreigners into Norwegian society, following the IMDi goals and guidelines.
- The development of a digital Livsmestring application (app) designed to assist students that are not proficient in Norwegian or English during their participation in the aforementioned course.

Our responsibility in this project centered on the development of the Livsmestring app. The app is meant to function as a companion app given to the students enrolled in the Livsmestring course. Specifically, the app's role in this project is to centralize all the necessary information from the course and make it accessible in the students' native languages. To make this possible, the course material has been adapted into video format in various different languages and integrated into the app. The Livsmestring project's development and oversight were led by Lisa Sveen Larsen.

1.1.3. Client

Oslo V.O. is an educational institution that provides a variety of courses and training programs for adults. This institution is part of Utdanningsetaten and together they offer a wide range of courses, from basic education to prepare for higher education, to language courses (such as courses in Norwegian aimed at adult immigrants).

Their goal is to make education accessible to adults at different stages of their lives, allowing them to participate more fully in society, improve their career prospects, or simply pursue new interests. The programs are tailored to meet the needs of adult learners, considering the flexibility they might need due to work, family, or other commitments. These educational services are provided across multiple facilities managed by the municipality, such as the adult learning centers found in Nydalen, Sinsen and Helsfyr.

Figure 1.- *Logo for Oslo Kommune*

1.1.4. Ethical Considerations

In our interviews, we aimed to gather information about the participants' age, education, and gender. We were also interested in considering their experience with technology, as well as obtaining feedback about our app after they had interacted with it. To maximize the value of these tests, we planned to record both video and audio, so it was necessary to apply to Sikt (self-described as the Norwegian Agency for Shared Services in Education and Research) for permission to keep these records. The application form is included in *Appendix B: User Testing Documentation*.

1.2. Contributors

1.2.1. Students

Table 1.- Table with information about the students that contributed to this project.

Name	**Number**	**Bachelor**
Alan Blanco Alquézar	s354528	Applied Comp. Technology
Oleksandra Chornous	s349584	Applied Comp. Technology
Petter Halsne*	s320947	Information Technology
Sari Siraj Abdalla Ali	s336154	Software Engineering

* Group representative

1.2.2. Contacts

Table 2.- Table with contact information of people involved with the project.

Name	Contact	Role
Way Kiat Bong	way-kiat.bong@oslomet.no	Associate Professor at OsloMet, internal supervisor
Lisa Sveen Larsen	lisa.kongsli@osloskolen.no	Project Leader at Oslo V.O. Helsfyr, external supervisor/client

2. Methodology

2.1. Planning

When embarking on a lengthy journey, it is important to have a plan prepared ahead of time. Resources have to be managed meticulously, and anticipatory measures are taken so as to circumvent potential obstacles.

Working on a project—no matter the scale—is similar to setting off on a journey. Identifying and understanding the challenges to be addressed is as critical as devising strategies for their resolution. Given the limited timeframe within which we must operate, time is the most valuable resource at our disposal. Therefore, finding a methodological approach to optimize our time allocation was our first priority. Our planning phase revolved around three main considerations:

1. devising a solution to the problem at hand.
2. dividing the workload among team members.
3. synchronizing our collective efforts in an efficient manner.

The initial step involved determining the most effective strategy for developing the app, which was required to support both Android and iOS platforms. The decision boiled down to two primary approaches: focusing on native language development or using a cross-platform framework instead.

Opting for native development presented a challenge—the necessity to master Swift, a language unfamiliar to our team. Moreover, native iOS development nearly mandates the use of Apple products—such as Macs and iPhones—for coding and testing, which poses a significant financial hurdle.

Conversely, we explored the potential of using Flutter, a cross-platform framework. Despite our initial unfamiliarity with Flutter, its promise of enabling simultaneous Android and iOS development from a single code presented an appealing solution. Ultimately, despite the customization limitations when compared to native coding, Flutter's time-efficiency and cost-effectiveness led us to choose this route.

To organize our efforts, we separated the workload into 'front-end' and 'back-end' components. We then formed two sub-teams, each tasked with either front-end or back-end development. This division was preliminary, with the understanding that responsibilities such as documentation, testing, and communication would be distributed as the project progressed.

Finally, to ensure smooth progress and timely updates, we adopted the Scrum methodology. Bi-weekly meetings were scheduled to facilitate comprehensive status reports and discussions on each member's developments, challenges, and next steps.

2.2. Tools

Table 3.- List of software used during the project.

Software	Function
Discord	Sometimes used to share content and/or have video-communication between group members.
Zoom	Used regularly to hold meetings with the internal supervisor.
FB Messenger	Main tool for fast communication between group members. Used to exchange files, pictures, or information.
Google Drive	Used for organizing and sharing project related files within group members and the internal supervisor.
Google Docs	Used for writing shared documents between all group members.
MS Word	Used for writing and formatting the final report.
Zotero	Used in the main report to add citations and list them out at the end of the report.
Android Studio	Used for all code development, both front-end and back-end by taking advantage of the Flutter Framework. Used to test code by running it on emulators.
Flutter	An open-source framework that can be utilized in different IDE`s for creating cross-platform applications from a single codebase. Almost all of our code is written in dart, and uses the libraries supported by the Flutter Framework.

GitHub	The flutter-project was shared on GitHub between group members and internal supervisor. Every update to the project was uploaded here after the first prototype was created.
Figma	Used to create early prototypes of the User Interface, as well as basic diagrams.
Firebase	Used to create a bucket in Cloud Storage for storing and reading video files.
Inkscape	Used to create the app's launcher icon.
ChatGPT 3.5	Used in code development and in assist of writing the report.

2.3. Agile Methodology

To provide structure to the project, we followed the agile methodology. As the name indicates, agile methodology was born from the need for rapid software development and processes that can handle changing requirements (Sommerville, 2016). Sommerville also notes that agile methods work well in situations where it is possible to have continuous communication between the product manager and the development team, as in the case with our project.

By contrast, traditional plan-driven software development is more linear, and as a result less flexible. This approach was developed for large teams responsible for building long-lived systems where rigor and formalized quality assurance are at the forefront (*e.g.,* the control systems for an aircraft). It demands complete specification of the requirements, and then designs, builds, and tests the system (Sommerville, 2016). However, within agile methodology, the requirements and the design can be developed simultaneously.

We have chosen to use the Scrum approach as the baseline. The most notable characteristic of Scrum is that it fragments the workload into sprints. Sprints are commitments to certain objectives within a timeframe. Because of this, Scrum highly values regular meetings and planning, in order to carefully prepare the right amount of work for each sprint. With the exception of the first sprint, which would be one week, we determined that our sprints would last for two weeks each. During this time period, we held regular meetings between ourselves, our supervisor, and the project coordinator.

2.3.1. Scrum

A comprehensive list containing brief descriptions of the most commonly used Scrum terminology can be found below:

- Product Owner (P.O.). The product owner represents a key stakeholder in the development of the product. Whether it is an individual or a group, the product owner is intimately involved during the development phase, as they are responsible for defining the requirements for the product and accepting–or rejecting–the updates performed on the product during a development cycle.
  In this project, Oslo V.O. is the product owner with Lisa Sveen Larsen acting as their spokesperson.
- Product Backlog. The product backlog is a list of tasks that need to be handled by the Scrum team. These tasks can range from product requirements that need to be implemented, to the creation of user stories or other features.
- Scrum Master. The Scrum Master ensures that Scrum methodology is applied throughout the project. They help everyone stay on track by facilitating meetings and guiding the team.
- Scrum Meeting. Whether they are daily or weekly, meetings are held with all team members to review progress and decide the next priorities. In our case, we held meetings bi-weekly.

2.3.2. Schedule

The schedule we created early on spans from the end of January to the end of May. The final deadline for the project's report is on May 24th. The workload is fragmented into 2-weeks long Sprints, with the exception of the first Sprint which was 1 week-long. An infographic of the schedule as well as complete walkthrough of the tasks tackled in each sprint is located in *Appendix A: Sprints*.

2.4. Requirements

The requirement specifications are the list of features that the product must have. Initially, the list of the minimum requirements is provided by the project giver. This list is paramount to the development process, as it informs us of our client's wishes while allowing us to manage our time and work depending on the number of features that the client expects. A product that satisfies these minimum requirements is called the Minimum Viable Product (MVP), and creating an MVP is our final goal during this project.

To have a clear understanding of the features we wanted to implement in the app we were building, we created a list of all the requirements and separated them into functional requirements and non-functional requirements.

The functional requirements explicitly dictate how the system should function, the type of services that it should provide, and how it should respond to certain kinds of input. They may also dictate how the system should not respond in certain cases (Sommerville, 2016).

On the other hand, non-functional requirements are not concerned with the specific functions of the app. What non-functional requirements do specify, however, is how the system provides its functionality. These kinds of requirements can pertain to speed, performance, security measures, quality of the content, scalability, accessibility, and so on.

2.5. Evaluations

Our evaluation of the Livsmestring app prototypes was twofold:

- Non-user evaluations: using a cognitive walkthrough and heuristic evaluations.
- User evaluations: conducted through structured user testing.

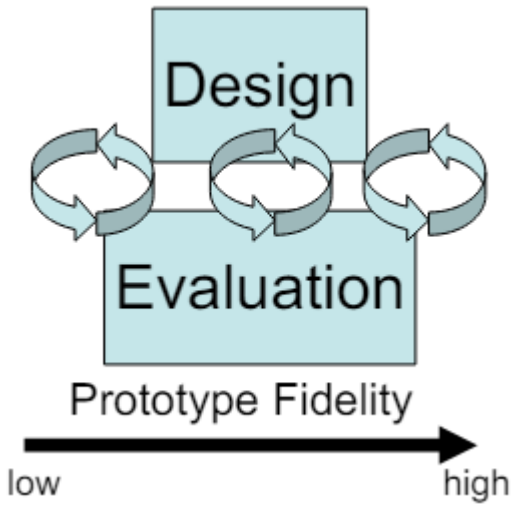


Figure 2.- *An iterative design approach (Rohs, 2010)*

By leveraging these methodologies, we consistently gathered insights that guided our design choices across all stages of prototyping and helped us take advantage of an iterative design model. Iterative processes are acclaimed for their incremental knowledge gain, facilitation of concurrent processes, and seamless integration of modifications. However, it is important to acknowledge that this iterative approach may lead to extended project timelines and increased costs (Wynn & Eckert, 2017, p. 153).

2.5.1. Non-user Evaluations

2.5.1.1. Cognitive Walkthrough

After finalizing our Figma prototype, we conducted a cognitive walkthrough to assess its learnability and ease of use. This method, pioneered by Lewis and Wharton (1997), focuses on the user's cognitive process and prior knowledge (Bligård & Osvalder, 2013). It is particularly beneficial during the development phase to identify design issues affecting user learnability. Early exploration of conceptual prototypes allows teams to measure how users interact with the interface and identify necessary improvements (Nielsen, 1994a).

The main objective of this walkthrough was to evaluate system usability and user learning efficiency. We aimed to confirm whether the system guides users from broad goals to specific intent and subsequent actions effectively. Key actions analyzed included:

- Language selection
- Tracking of learnability progress
- Accessing the first video in the career chapter

Each action was scrutinized through specific questions:

- Does the user's action result match their goal?
- Can users easily find and recognize the actionable elements?
- Are users confident that the selected action will fulfill their intent?
- Do users understand the feedback after completing an action?

These questions were integral not only during the cognitive walkthrough, but also throughout the user testing phase.

For interface design, we adhered to Ben Shneiderman's *8 Golden Rules of Interface Design* (1986) to ensure intuitiveness, ease of learning, and user productivity enhancement (Shneiderman & Plaisant, 2005, p. 7).

2.5.1.2. Heuristics

Our heuristic evaluation involved a systematic examination of potential usability issues, design flaws, and functional shortcomings based on Nielsen's *10 Usability Principles* (Nielsen, 1994b). This method relies on the evaluator's experience and intuition rather than scripted scenarios, encouraging exploration and critical analysis to pinpoint usability concerns.

As mobile technology evolves, so do heuristic evaluations. The shift from push-button to touchscreen phones illustrates this evolution. Touchscreens, while convenient, lack the tactile feedback of traditional buttons, which can lead to input errors (Inostroza et al., 2013). As a result, we included specific heuristics tailored for touchscreen interfaces.

Additionally, recognizing the unique needs of mobile interfaces, Oliveira, Branco, Carvalho et al. (2022) proposed integrating three new heuristics from the MATCHMED scale into our evaluation: minimization of human-computer interaction, ergonomics, and readability for quick viewing (Oliveira et al., 2022, p. 3).

This comprehensive evaluation approach not only addressed traditional usability aspects but also adapted to the specific challenges posed by mobile platforms, enhancing the overall effectiveness of the Livsmestring app.

In preparation for each round of user testing, we also employed heuristic testing to identify potential usability issues within the Livsmestring app. Unlike the comprehensive evaluations conducted by Oliveira et al. (2022), who assessed 78 screens of a mobile application, our heuristic evaluation was more limited, but still informative. Our testing focused on several principles critical to UI design, giving us valuable insights that lead to future refinement of the app:

1. We prioritized clear feedback mechanisms within the app. For example, we ensured that all translated text was descriptive and free from ambiguous special characters. We also added a loading screen as a padding between some screen transitions. These type of changes help the user stay informed and understand the system's status at all times.
2. We ensured that there was consistency across the application. Giving consistency to the app's elements, such as the font sizing and the color schemes, helps to foster predictability and ease of understanding.
3. To improve user interaction, we optimize the interface by adding dynamically responsive text, ensuring that no title is too lengthy. Additionally, we organized all language options in a comprehensive manner.
4. By displaying all languages in their native alphabets and using native numerals for non-Latin numerical systems, we reduced the cognitive load on our users. This design choice supports the heuristic of aiding user recognition over recall, making the interface more intuitive and less reliant on memory.
5. We favored a more minimalist design to simplify the usability and understandability of the app.

These adjustments reflect our ongoing commitment to optimizing the app's design to better meet the needs of its users.

2.5.2. User Testing

User testing for the Livsmestring app was conducted at the Oslo V.O. Helsfyr facilities with students from the Livsmestring course. In both instances of the testing, we

gathered five participants. The primary distinction between the two sessions was the testing environment; the first round of testing was conducted in a single room which limited privacy. This setup introduced the possibility of user behavior biases, specifically learned behavior, where participants may modify their actions or responses by observing others (Hufnagel & Conca, 1994).

This scenario risked the authenticity of the feedback we received, and potentially affected the effectiveness of the testing. In response, the second round of testing was conducted with each user individually, to better isolate their experiences and opinions.

2.5.2.1. Participants

The participants involved in the testing were students of the integration course at V.O. Helsfyr, and the majority of them possessed minimal specialized knowledge about mobile device technology. The group of participants had diverse backgrounds, reflecting a range of historical, social, cultural, and economic influences, which was essential for highlighting the users' different needs when interacting with the app. Above all else, ensuring that the future users' needs were met was our top priority when conducting these tests.

2.5.2.2. Data Collection

In our efforts to gather a diverse group of interviewees, we ensured representation from various nationalities and age groups, while avoiding the inclusion of vulnerable groups or individuals. This approach eliminated the need for special considerations during the interviews. To facilitate this process, we developed interview consent forms in English, Norwegian and Ukrainian (the English version can be found in *Appendix B: User Testing Documentation*).

To ensure that ethical standards were also met, we abided by the principles of privacy, anonymity, and confidentiality. For instance, we did not ask for sensitive information that was irrelevant to the study, such as the participants' names. All participants were also provided with comprehensive details about the study they were participating in.

We conducted face-to-face semi-structured interviews using a tailored questionnaire, and we recorded the video and audio of the testing process. The interviews employed a mix of open and closed questions to gather feedback. Our goal was to formulate clear, concise questions to understand the users' experiences and thought processes without leading them towards specific responses.

Most questions were designed to build upon previous responses, creating a coherent and fluid conversation. Nevertheless, maintaining flexibility during the interviews was essential to ensure the participants' comfort and to gather data effectively. As suggested by

Bird (2016), deviations from the script can be beneficial when spontaneous questions or topics arise, as it can add more depth and relevance to the conversation.

We also prepared a questionnaire based on the system usability scale (SUS) employing a Likert scale rather than a semantic differential scale. Likert scales allow the participant to rate their agreement with a statement on a scale from 1 (‘strongly disagree’) to 10 (‘strongly agree’), facilitating feedback regardless of linguistic backgrounds.

Moreover, we incorporated elements to evaluate the users’ ability to adapt to changes within the system, by asking them to navigate between various sections of the app and measuring the time that they took to complete each task. This approach helped us assess a participant’s flexibility and confidence when using the system, providing valuable data on ease of learning and adaptation.

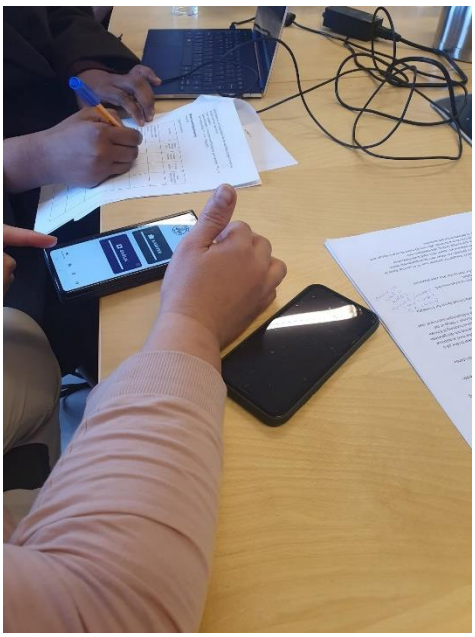
2.5.2.3. Testing Protocol

2.5.2.3.1. Tasks

The testing procedure consisted of a warm, introductory briefing of us and the project, followed by a sequence of tasks that the participants had to perform in the app. These tasks were designed to evaluate the app’s functionality and user interface (UI), including:

- Selection of a language and navigation to specific chapters
- Operations within the video player, such as starting the video, stopping it, changing the playback speed, and adjusting the screen size
- Navigation between pages and settings adjustments

The tasks were completed using the interviewer’s mobile device. The participants were encouraged to use the Think Aloud technique while performing the tasks to gain insight into their cognitive processes. After the users completed each task, we noted down the user’s ability to complete the task and any notable interactions or difficulties encountered. We also measured the time taken to complete each task. The interviewers strived to remain silent and provide no cues to the users while performing a task. However, constructive interactions were allowed, where the participant and the interviewer discussed app elements and their functions.



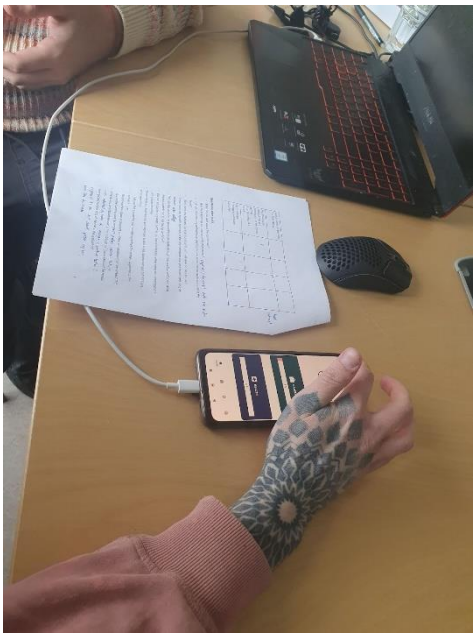


Figure 3.- *First and second round of user testing, respectively*

2.5.2.3.2. Questions

Post-task completion, participants were engaged in a semi-structured interview designed to obtain feedback on specific elements of the application, including the video player, navigation, and overall UI. Our questions focused on:

- User satisfaction with the video player and its features
- Opinions on the effectiveness of the navigation
- Personal preferences and observations regarding the layout and functionality of the chapter listings
- Experiences with locating and using the language settings, and experiences understanding the purpose of specific interface elements (*e.g.,* the progress bar)

After all the questions were answered, we additionally asked the participants to complete a Likert scale questionnaire. The entire usability testing protocol can be found in *Appendix B: User Testing Documentation*. A deeper look into the thought process that went behind crafting the user tests, as well as the conclusions that we gathered from them can be found in *Appendix C: User Testing Evaluations.*

3. Results

This part of the report presents the results derived from our methodology. First, the requirements for the development of the product are presented. Then comes the first iteration, where the Figma prototype, user testing 1, and the two first high-fidelity versions of the application are outlined. At the second iteration, the applied changes from user testing 1, the results from user testing 2, and the final applied changes are presented.

3.1. Requirement Analysis

3.1.1. Functional Requirements

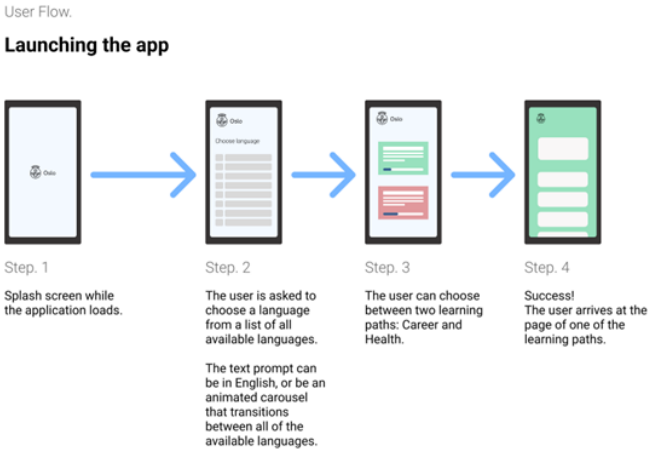
According to the client's wishes, the main functional requirements we had to contend with were:

1. The user can watch videos on the application.
2. The user can choose and change the language in the application, or the language is automatically changed based on the phone's settings.
3. The teachers can see the progress of the user, but without the user needing to create an account.
4. The app can read the text content aloud for further accessibility options.

Based on the list of requirements, we created some diagrams in order to structure the flow and hierarchy that the different pages of the app would have.

3.1.1.1. User Flow

User flow diagrams–or just flow diagrams–provide insight into the different ways that a user might interact with the system. Before creating the app's wireframe, we illustrated three different user flow diagrams.



The first user flow involves the series of steps involved in launching the app for the first time.

Figure 4.- *Flow diagram for launching the application*

The second user flow represents the path that a user would take in order to complete a lesson. At this time of the development stage, we believed that the users might need to answer some questions in order to ensure that they have watched the video and validate their progress. Later on, this functionality would be scrapped.

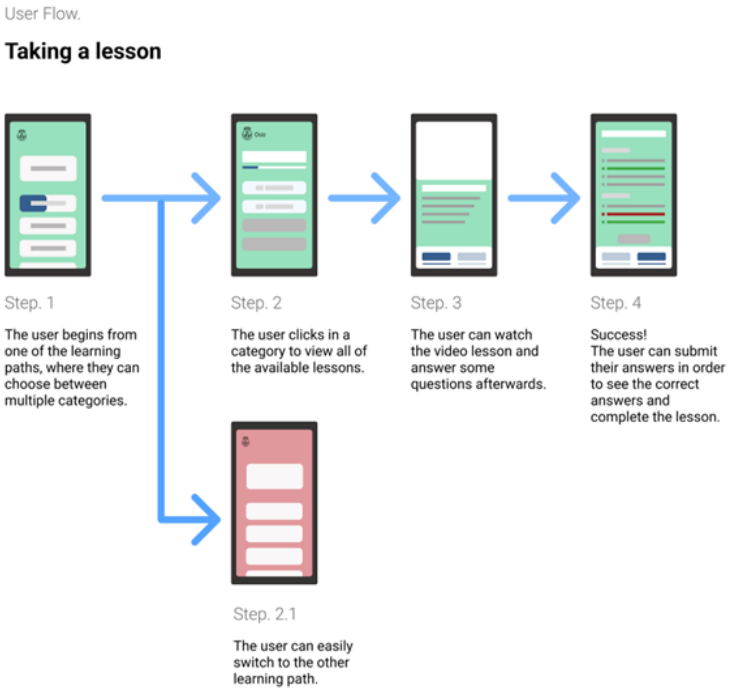


Figure 5.- *Flow diagram for taking a lesson*

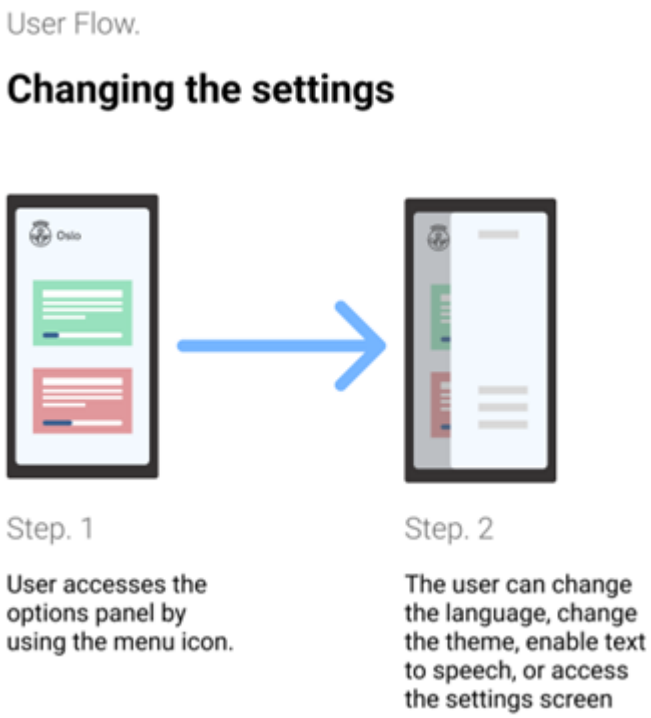


Figure 6.- *Flow diagram for changing the settings*

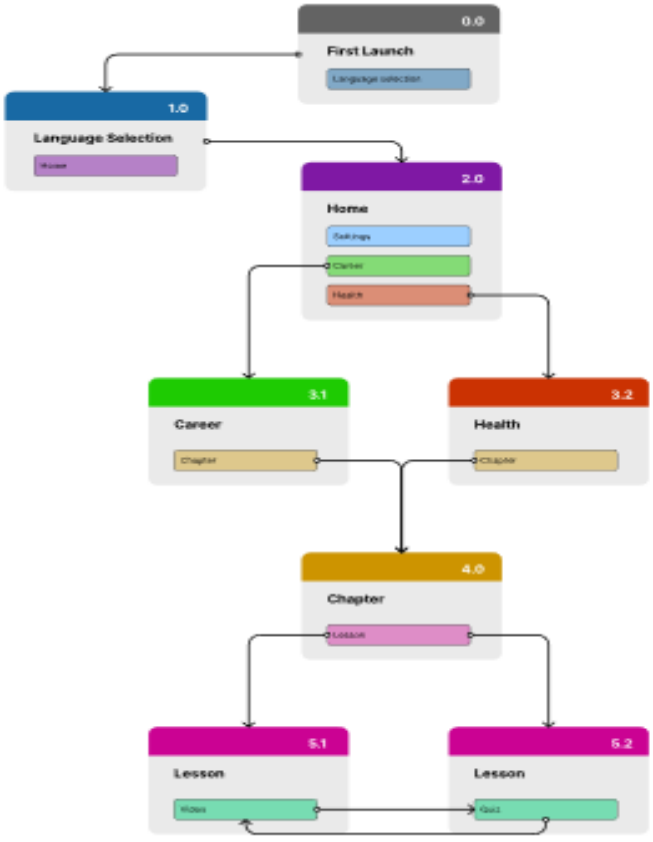


Figure 7.- *Sitemap of the Livsmestring application*

Finally, we needed to showcase how the user might have access to the settings page, which is illustrated in our final user flow diagram. Initially, as seen in the diagram above, we determined that a drawer. Initially, as seen in the diagram above, we determined that a navigation drawer would be a suitable solution, but that would later get scrapped in favor of a different– and hopefully more intuitive navigation system.

3.1.1.2. Sitemap

The diagram on the left (*Figure 7*) illustrates the sitemap for the Livsmestring app, which assists us in structuring and organizing all the pages of the system. Here, we can see the different layers of the application, and how they relate to each other, just like a real-world map provides overview over the structure of a place.

3.1.2. Non-functional Requirements

Table 4.- List of non-functional requirements and their priority level.

Requirement	Description	Priority
Performance	The application must effectively use data in a way that it functions as a mobile application. This means that it needs to be responsive.	Medium
Security and privacy	Does not collect unnecessary information about the user, and stores information securely.	Medium
Usability	The application can be used effectively on a vast variety of tablets and phones.	Medium
Aesthetics	The application has an appealing design.	Low
Universal design	Focus on that the application is perceivable and robust. So that the functionality of the application is satisfying. Data needs to be presented to the user in a way so that it becomes perceivable. The framework and codebase must be developed in a way that promotes accessibility (WCAG 2.1).	Medium
Maintainability	The codebase must be developed in a way that can be modified, updated, or extended. Consistent coding, documentation and testing is essential to achieve this.	High
Documentation	The application is well documented with commented and readable code.	High
Delivery	The application is finished and launched at the end of May.	High

3.1.3. Risk Analysis

A risk analysis is a part of the risk assessment process, where each identified risk gets assigned a value based on its probability of occurring as well as the ramifications of the event actually taking place.

Table 5.- List of identified risks, their probability of them happening, and the consequence levels.

#	Risk	Probability	Consequence
1	The performance of the application is not good enough to handle the data stream	Medium	Serious
2	The application's documentation and maintainability is inadequate.	Low	Serious
3	We misjudge the amount of time and do not deliver a fully functional product or lacking product in time.	Medium	Serious
4	The application appearance is not appealing	High	Less serious
5	The application is lacking in terms of Universal Design guidelines	High	Less serious
6	The application gathers too much information about the user and potentially violates privacy.	Low	Less serious

3.2. First Iteration

In this section, the first iteration is described. The development of the Figma prototype was the first step in the development of the final product. The first high-fidelity version was made based on this prototype, which was a version of the application built on static data. The second high-fidelity version implemented a new design for one of the navigation paths and implemented dynamic data usage and translation. This part culminated in the first user testing.

3.2.1. Figma Prototype

In this sub-chapter, the Figma Prototype gets outlined. In the process of making the prototype, a design was necessary that took advantage of different components, styles, and plugins.

3.2.1.1. Design

The development of the Livsmestring app commenced with a focus on UI design, setting the foundation for the application. This phase involved collaboration with the P.O. to refine both functionality and aesthetics using various UI design tools such as Sketch, Proto.io, and notably, Figma (Stevens, 2024). Clients often require visualization aids to fully grasp software capabilities and limitations, hence the necessity for an iterative design process to reach a mutually agreeable layout for development.

We chose Figma because it offers several advantages specifically tailored for mobile app design by a remote and distributed team:

1. Responsive Design: Figma helps to create responsive layouts, optimizing interfaces for diverse screen sizes and orientations, which is essential for mobile app design.
2. Preview on Mobile Devices*:* Figma Mirror allows previewing the design on actual mobile devices, ensuring accurate representation, and allowing for real-time testing and feedback.
3. Prototyping for Mobile Interactions: Figma's prototyping tools facilitate the creation of interactive prototypes tailored for mobile interactions, including gestures such as swiping, tapping, and other user actions.
4. Component Libraries: Figma libraries can be created and maintained specifically for mobile app design, promoting consistency and efficiency across screens and interactions.

5. Developer Handoff for Mobile Platforms: Figma provides developers with the design resources and specifications needed to accurately implement the design on mobile devices, such as iOS and Android.
6. Mobile-specific Plugins: Figma plugins support mobile app design needs by offering functions like generating iOS and Android assets, optimizing images for mobile screens, and integrating with mobile-specific APIs.
7. Real-time Collaboration for Remote Teams: what enables seamless communication, feedback, and iteration regardless of geographical location of our remote team members.
8. Cloud-based Accessibility: As a cloud-based platform, Figma can be accessed from any device with an internet connection, which is beneficial for our remote team.
9. Version Control and History: Automatic saving of version history, enables designers to track changes and revert to previous versions if needed, ensuring the integrity and consistency of mobile app design throughout the design process.
10. Integration with Mobile App Development Tools: Figma integrates with mobile app development tools, streamlining the design-to-development process for mobile projects.

Through these features, we initially crafted a low-fidelity prototype, progressively refining it to a high-fidelity version that closely mirrored a live mobile application. This iterative process was vital for achieving uniformity in design elements such as colors, typography, and layouts before moving to development.

3.2.1.2. Components and Styles

Following Google's Material Design guidelines ensured that the app's UI was both intuitive and visually appealing, particularly for Android users (Google, n.d.). The design manual for Oslo municipality's visual identity was also instrumental in maintaining brand consistency across various visual elements such as logos, colors, and typography (Oslo Kommune, n.d.). These guidelines also served as a reference for stakeholders to ensure that the brand covered the required specifications by Android and Oslo.

We built a prototype of the Livsmestring app in Figma using the client's original wireframe as a foundation, emphasizing design aspects like color, imagery, and typography. As a rule, the UI encompasses the visual presentation, interface design, navigation system, and underlying processes that facilitate the app's functional and structural. Simultaneously, we kept in mind the importance of maintaining consistency and intuitiveness across all these elements for a smooth user experience (UX). Our prototype demonstrates how users will interact with the app, showcasing its functionality through interactive elements.

Figma's components and styles simplify the creation of mobile app wireframes. These features enable the systematic reusing of UI elements and properties, ensuring consistent designs on a large scale. When modifications are required, such as adjusting a button's color, changes made to the main component or style reflect automatically across all design instances, securing efficient and synchronized updates (Figma, n.d.-b).

Components serve as reusable building blocks for UI elements such as buttons and navigation bars. Designing components at the project's inception can significantly reduce the time needed to implement widespread modifications across multiple screens. We identified recurring elements and transformed them into components, serving as fundamental units that can be reused throughout the project. By nesting instances of these components within others, consistency is ensured across all elements of the app. Any changes made to the original component automatically extends to all instances, simplifying maintenance and updates across all of the screens.

Our list of the components includes UI components, Oslo municipality logo, icons, device mock-ups, platform OS components (for Android and iOS), flow arrows and flowchart shapes. Styles enable uniformity in design attributes, such as typography and colors, enhancing the clarity and coherence of the wireframe. Grids and effects like shadows and blurs were chosen voluntarily to enhance the visual hierarchy and depth within the UI design, elevating the overall UX.

We also considered that components (*e.g.,* buttons and text) change the colors and font size in active/passive status. We did not work on designing themes like light or dark, since the client did not consider it as an essential functional requirement. Due to the time limit, we focused on the essential elements and produced the Livsmestring app in a more traditional light theme.

We created pages in both design and flow modes in Figma. Initially, we focused on identifying and designing reusable components that would be used across multiple screens. Consequently, our design page consists of separate frameworks containing these reusable components, along with predefined colors and fonts. The components and pages can be seen in Appendix D: Figma Prototype.

Additionally, we developed a flow page illustrating the user's streamlined journey through the app which can be found Appendix D: Figma Prototype. After several discussions with the client, the app's design on Figma was changed.

3.2.1.3. Plugins

Figma offers a variety of plugins designed to enhance functionality and workflow. For example, the 'contrast' plugin enables designers to evaluate color contrast within their designs, ensuring accessibility and compliance with standards such as the Web Content

Accessibility Guidelines (WCAG) (Figma, n.d.-c). The 'Color Contrast Checker' plugin allows designers to assess the color contrast of text and background elements within their designs, ensuring accessibility and readability for all users (Figma, n.d.-a). Due to the limitations of Figma's Education account, lacking plugin access, we advanced to the development stage, intending to test the high-fidelity prototype with real users afterward.

3.2.2. First High-Fidelity Prototype

As mentioned in *3.2.1.2. Components and* Styles, the project employs Google's Material Design, which offers standards and elements for crafting contemporary, visually attractive, and cohesive UIs across various platforms.

The app prototype was developed using the Flutter framework, employing a combination of programming languages including Dart, C++, CMake, Swift, HTML, and C, as illustrated *Figure 8*. This prototype encompasses multiple classes and pages, showcasing a comprehensive structure and an array of functionalities. In the following sections a concise overview is provided, highlighting its structural elements and the functionalities it encompasses.

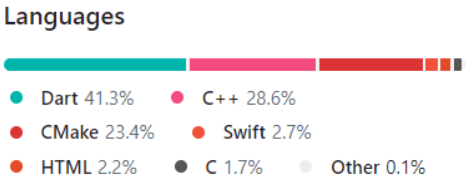


Figure 8.- *Programming language distribution as depicted in Github*

3.2.2.1. Project Architecture

3.2.2.1.1. Structure

The project comprises various directories, with the primary ones being lib, assets, and web. Within the test directory, there exists a file housing a rudimentary Flutter widget test. Furthermore, the *pubspec.yaml* file contains essential details regarding the dependencies employed in the prototype.

In general, the above-mentioned project's directories have the following features. The assets directory holds static elements (*e.g.,* files of the logo of Oslo commune), which are referenced in the Flutter code to display images and custom typography in the app's UI.

The lib directory includes dart files organized based on the structure of the application. It serves as the organizational hub for application logic, UI components, and business logic.

The web directory includes files for the web version of the app in case the web platform will be developed for use along with mobile. It helps to organize web-specific assets and configurations for possible Flutter web applications. As a result, a seamless and

responsive UX across different platforms could be provided. Illustrations of the project structure are found in *Appendix E: Technical Specifications*.

3.2.2.1.2. Dependencies

In the initial version of the app, the core dependencies for maximizing utility were narrowed down to include *cupertino_icons* (with iOS-style icons), *shared_preferences* (for efficient storage of key-value pairs), *video_player* (for seamless video playback functionality), and *chewie* (for facilitating smoother control over video playback).

Furthermore, development dependencies were incorporated, such as: *flutter_test* (for simplifying unit and widget testing) and flutter_lints (to enforce good coding practices). Together, these dependencies streamline both the development and functionality of the Livsmestring app prototype, guaranteeing smooth performance and effective testing procedures.

3.2.2.1.3. Functionality

The *SplashScreen* serves as an initial loading screen, directing users to the appropriate page based on their language selection. The *LanguagesScreen* allows users to explore language options. Chosen language is saved after the first time of choosing it and can be changed by the user anytime. The *NavigationPage* manages navigation between such sections of the Livsmestring app as Home, Work, Health, or Settings. Both pages can be seen in *Figure 9.*

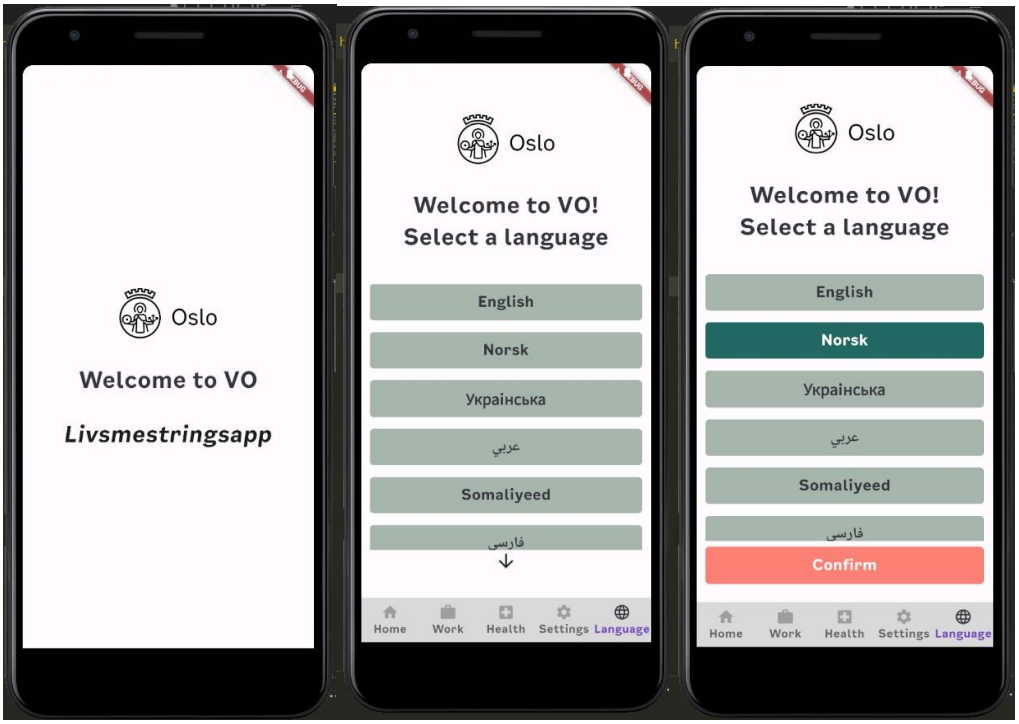


Figure 9.- *SplashScreen and LanguageScreen, respectively*

The *HomeScreen* serves as the entry point, providing buttons for navigation to career and health sections. The *CareerPage* and *LanguagesScreen* allow users to explore career and language options respectively. The *VideoPage* displays videos related to career or health topics. A screenshot of these pages can be seen in *Figure 10.*

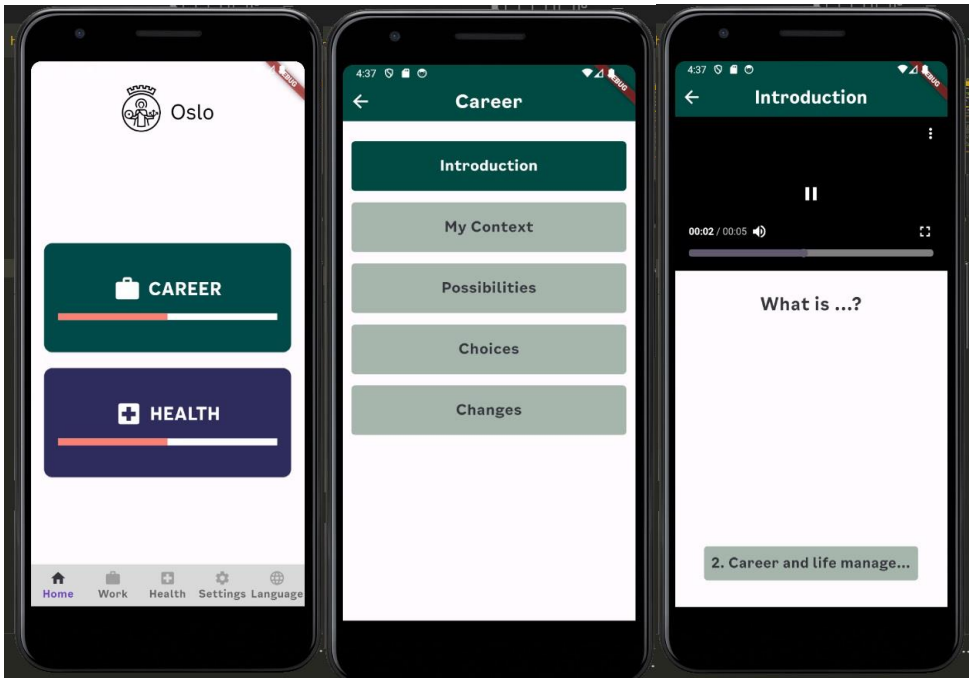


Figure 10.- *HomeScreen, CareerPage and VideoPage, respectively*

Overall, the Livsmestring app is a robust Flutter application designed to provide users with extensive resources and information on integration into Norwegian society, and, particularly, on career and health topics. With its multilingual capabilities, it caters to diverse audiences, offering an intuitive interface and various features to enhance the learning experience.

3.2.2.2. Reusable Components

The project employs UI elements such as buttons and cards to improve the interface and functionality. These components promote an interactive UX and can also offer feedback when there are user-application interactions. We achieved this by using reusable components, which was an extremely efficient way of using assets, as it cut down the development time spent creating each component while establishing a consistent theme for the application. The reusable components that we dealt with were:

- *ConfirmButton*
- *LanguageButton*
- *HomePageCard*
- *ListButton*

A full rundown of these components as well as accompanying illustrations are located in *Appendix E: Technical Specifications*.

3.2.2.3. Styling

The pivotal management of the app's visual aesthetics is incorporated by the files *colors.dart* and *fonts.dart*. As noted in *3.2.1.2. Components and* Styles, color definitions, font choices, and text styles for the Livsmestring app were established during the Figma prototyping phase, aligning with the specifications outlined in the Oslo commune Design Guidelines.

Within these files, color constants and text styles serve as fundamental components consistently integrated across various components and screens within the application. This approach ensures the cultivation of a unified and cohesive design language throughout the application's interface. Essentially, *colors.dart* and *fonts.dart* function as repositories for establishing the app's color palette, font choices, and text formatting standards, thus contributing significantly to its overall visual coherence.

```
class AppColors {
  static const ashGrey = Color(0xffA7B6AC);
  static const casal = Color(0xff206964);
  static const tune = Color(0xff35343E);
  static const white = Color(0xffffffff);
  static const salmon = Color(0xfffc8074);
  static const lightGrey = Color(0xffD9D9D9);
  static const weakedGreen = Color(0xff034B45);
  static const spaceCadet = Color(0xff2A2859);
  static const spindle = Color(0xffA1BDE7);
}
```

Figure 11.- *Color palette for the Livsmestring app*

The *AppColors class* within *colors.dart* encapsulates a set of standardized color constants utilized extensively throughout the application, as depicted in *Figure 11*. Similarly, the *Fonts class* within *fonts.dart* delineates an array of text styles which are associated with various interface elements such as *LanguageButton* and *LanguageButtonActive* (*e.g., Figure 12*). These selections adhere closely to the Oslo commune Design Guidelines, thereby ensuring conformity to prescribed visual standards across the entirety of the application.

```dart
static const header1 = TextStyle(
  fontFamily: 'Oslo',
  fontWeight: FontWeight.bold,
  fontSize: 30,
  color: AppColors.tune,
);
```

```dart
static const bottomNavLabelSelected = TextStyle(
  fontFamily: 'Oslo',
  fontWeight: FontWeight.bold,
  fontSize: 15,
  color: AppColors.tune,
);
static const bottomNavLabel = TextStyle(
  fontFamily: 'Oslo',
  fontWeight: FontWeight.bold,
  fontSize: 15,
  color: AppColors.ashGrey,
);
```

Figure 12.- *On the left, text style for Header 1, on the right, text styles for bottom navigation labels*

3.2.3. Second High-Fidelity Prototype

After the first version of the application was developed in Flutter, we had a meeting with our contact from the P.O.; the feedback was good, but one of the developers got an idea of making an added feature to the app: some sort of overview to illustrate the structure of the application. This idea developed into a new navigation path where the user could see the overview of the content and then enter it by clicking on it in the overview instead of going into folder after folder before reaching the content. This added feature, combined with dynamic data, paved the way for the second version of the application.

3.2.3.1 Structure

At its core, the idea of the structure remained the same, which was to divide the front-end (UI) and the back-end logic in a comprehensive way. However, some of the folders were renamed, and some of the content in the files inside the folders was changed or extended: the classes folder got renamed to models, the reusable folder got renamed to widgets, and the services folder was added. This last folder consisted of the data handling side: back-end logic for the pages and service calls to the database.

The *Firebase Options* file was created by Flutter by default when the application was integrated into Firebase Storage. This file contains data regarding app IDs for both Android and iOS. It is essential for the application to be able to identify and authenticate itself with Firebase. The *main.dart* file is the entry point to the application. It contains a main method that ensures that the app runs.

First, it initializes the database used in the application (Firebase). Then it sets the preferred orientation of the screen to 'portrait_up.' This means that the phone only renders the views in portrait mode and does not allow landscape mode. Though there is one exception with the video player in full-screen mode, At the end, the method ensures that the

chosen language in the application is fetched from Shared Preferences and that the Locale is set by it. If the value is null, then Norwegian is set by default as the Locale. At the end, the method invokes the *runApp()* method, which passes the *MyApp* class following the current language and the Locale.

MyApp class is a stateful widget that needs to initialize the current chosen language and the Locale. After defining the initialization, it creates a void method called *updateLocale*(), which is called later in the application in order to restart the app and set the new changes to the Locale and thereby the language. The class defines the widget it builds at the end, which returns the *GetMaterialApp* class.

The *GetMaterialApp* class can set and define different values and settings for the application, but it also determines which class to go to next to render the view; refer to *Figure 13* below.

```
69      @override
70      Widget build(BuildContext context) {
71        return GetMaterialApp(
72          debugShowCheckedModeBanner: false,
73          translations: LocaleString(),
74          locale: _locale,
75          title: 'life_mastery_app'.tr,
76          theme: lightMode,
77          darkTheme: darkMode,
78          home: SplashScreen(
79            selectedLanguage: _selectedLanguage,
80          ),  // SplashScreen
81        );  // GetMaterialApp
82      }
83    }
```

Figure 13.- *Widget.build in main.dart file*

The first field ensures that we remove the debug banner shown in the top-right corner of the screen. This was shown in the frames in the first iteration. The second and third fields are explained in the sub-chapter *3.2.3.4* Translations. The title field takes in a string that sets the name of the application. The theme and dark theme fields set the light and dark themes. These features are not implemented but rather made possible for future development. At the end, the home field takes in the class that is the home/start of the application. This means that the widget being shown when the app starts is the widget of the *SplashScreen*.

3.2.3.2 Front-End

In the first version, the pages that were covered there remained largely consistent in terms of UI, but with minor adjustments. However, a significant shift occurred in their characteristics as we transitioned from handling static to dynamic data. Consequently, all pages had to be implemented as stateful widgets instead of stateless widgets. While many pages retained their original layout, some underwent substantial changes, particularly the *BottomNavigationBar* widget. Changes that affected the original classes were the introduction of new navigation features, dynamic data transitions, and the implementation of translation. Additionally, to accommodate these changes and features, new pages were created. Details of the pages can be found in *Appendix E: Technical Specifications*.

Figure 14.- *HealthPage*

HealthPage, a stateful widget that can be observed in *Figure 14*, serves the same purpose as the *CareerPage* covered in the initial version. Rather than maintaining these pages within the same class, as done previously, we opted to segregate them. Although their underlying logic remains largely unchanged, they necessitate fetching different text strings and employing distinct color schemes for the UI. Thus, instead of housing them in a single file with state differentiators, we chose to distribute them into separate classes: *CareerPage* and *HealthPage*.

The *HealthDetailsPage* is a stateful widget that functions similarly to the *HealthPage*, see *Figure 15*. However, its distinguishing feature is the requirement for an index from the previous class. This index enables the *HealthDetailsPage* to access the appropriate sub-chapters and display them as buttons.

When a list button is clicked in the *HealthDetailsPage*, it forward the index of the clicked item, along with the index of its parent and the name of the clicked item. This functionality is leveraged in the *HealthDetailsOfDetailsPage*, which operates similarly to the *CareerDetailsPage*. It requires access to the appropriate text strings and video URLs, listing them as buttons until all topics within the current sub-chapter are displayed.

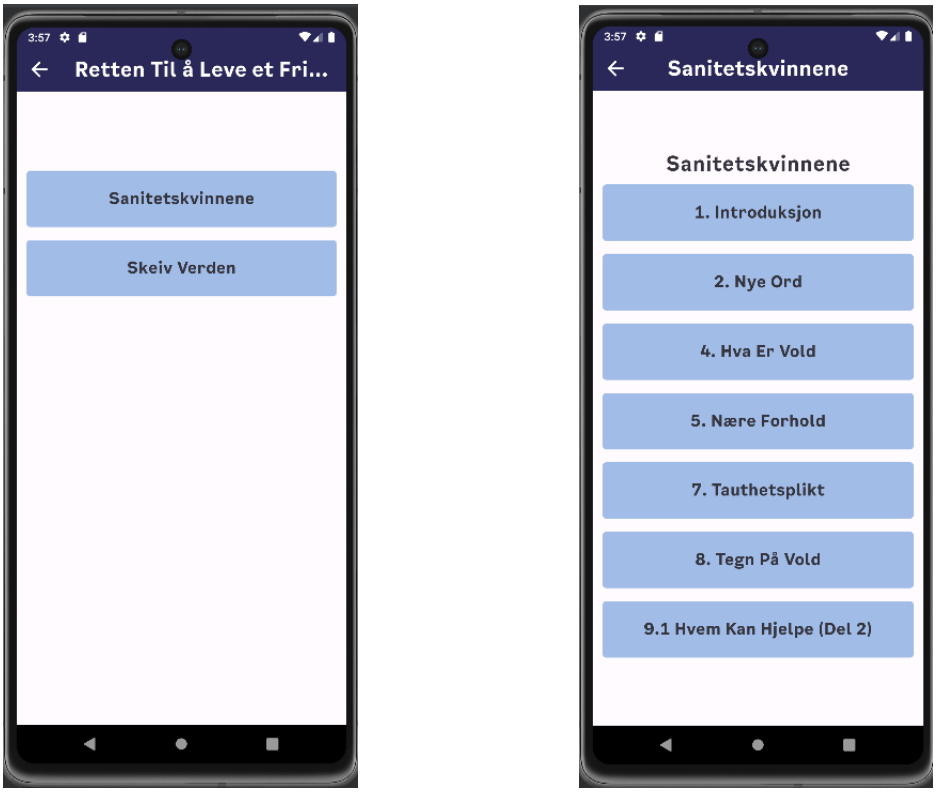


Figure 15.- *HealthDetailsPage on the left, HealthDetailsofDetailsPage on the right*

In addition to implementing the old navigation for the Health section, we introduced the new navigation feature. For the Career section, this involved creating the *CareerPageNew* class (see *Figure 16*). This class optimizes navigation by utilizing nested lists. It employs a *ListView.builder* to display chapter names and utilizes nested lists within a *SingleChildScrollView* to present topics within each chapter as list items. Essentially, *CareerPageNew* combines the functionalities of *CareerPage* and *CareerDetailsPage* into a single class. Alongside enhancing navigation efficiency, this class also includes a visual indicator—a green tick—to illustrate if the video inside the respective topic has been watched.

HealthPageNew and *HealthDetailsPageNew* also incorporate elements of the *CareerPageNew* logic, though with some adjustments to accommodate the structure of the health section. Due to the inclusion of sub-chapters alongside chapters, the content is divided between two distinct classes. *HealthPageNew* mirrors *CareerPageNew's* approach by utilizing nested lists, but it lists chapters and their corresponding sub-chapters rather than chapters and subsequent topics.

Clicking on a sub-chapter in *HealthPageNew* navigates the user to *HealthDetailsPageNew* where the selected sub-chapter serves as the heading for the list. The topics within this sub-chapter are then listed out, following the format established in the *CareerPageNew* class. HealthPageNew and HealthDetailsPageNew can both be seen in *Figure 17*.

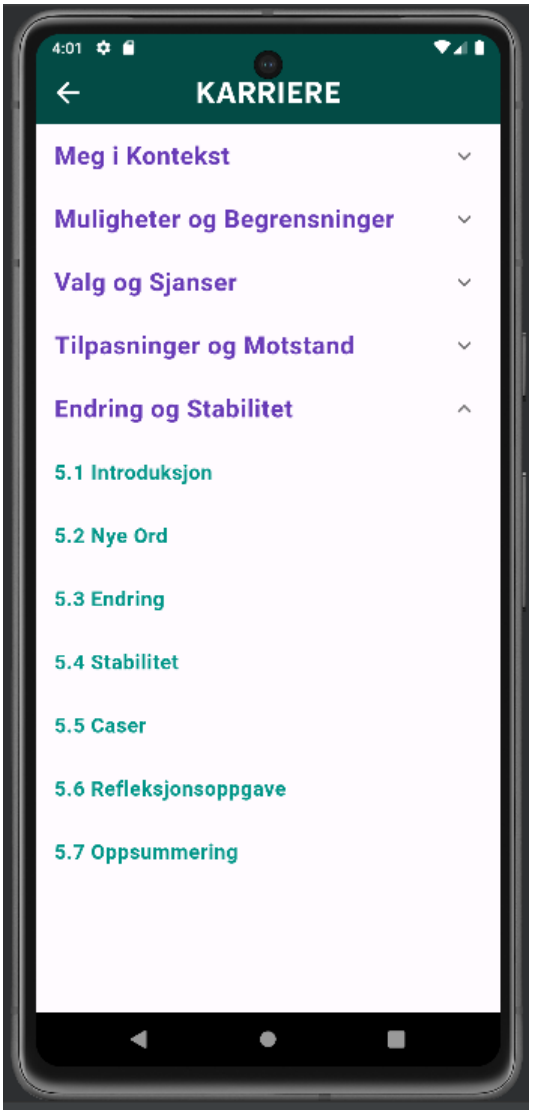


Figure 16.- *CareerPageNew*

The *BottomNavigationBar* widget underwent modifications where we removed the language navigation option and retained the previously described four functionalities (see *Figure 18*). Typically, language settings are found under settings, so we introduced and made the *languagePageNav* class a feature within this context. The file name reflected our anticipation of potential future features for the settings navigation.

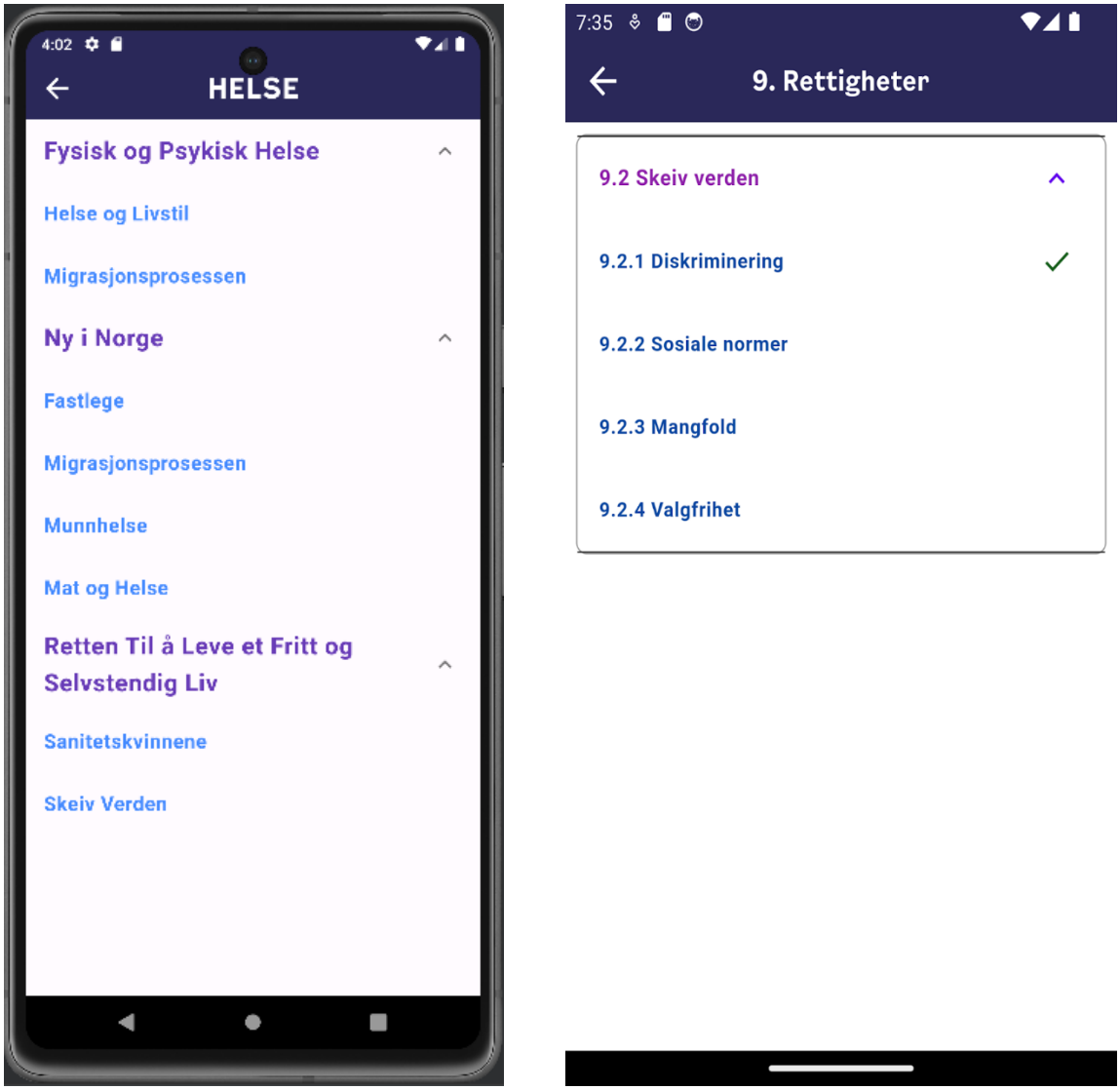


Figure 17.- *HealthPageNew on the left, HealthDetailsPageNew on the right*

The languagePageNav class renders a view with a button. Upon clicking this button, it constructs an *AlertDialog* containing a *ListView* displaying all available languages for selection. Upon choosing a language, it saves the selection in Shared Preferences and restarts the application to apply the changes.

Finaly, the *VideoPlayerPage* class is responsible for displaying and managing video playback. It utilizes the *Appinio* video player package for rendering videos. This video player replaced *Chewie* from the previous version. Upon initialization, it sets up the video player and initializes necessary variables. The dispose method handles the saving of video playback progress to SharedPreferences. The UI provides not only a video player, but also an optional next video button. They can be seen in *Figure 19.*

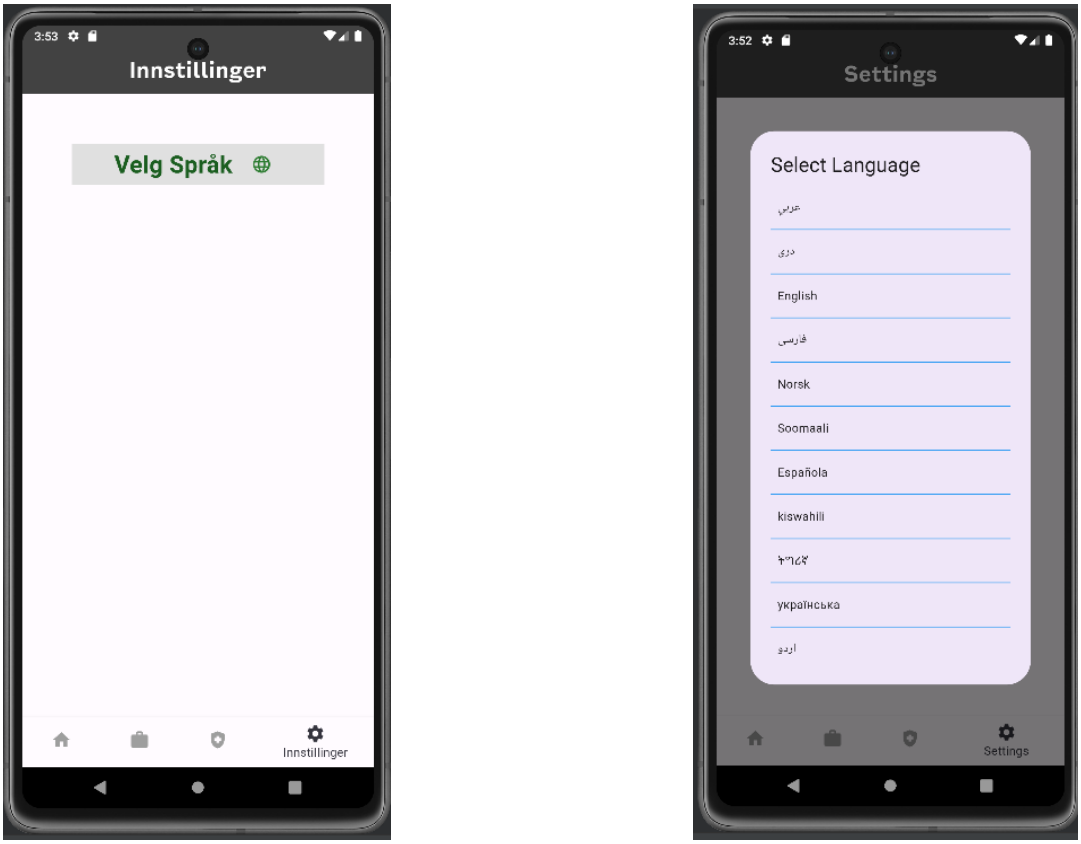


Figure 18.- *SettingsTab selected in BottomNavigationBar on the left, AlertDialog on the right*

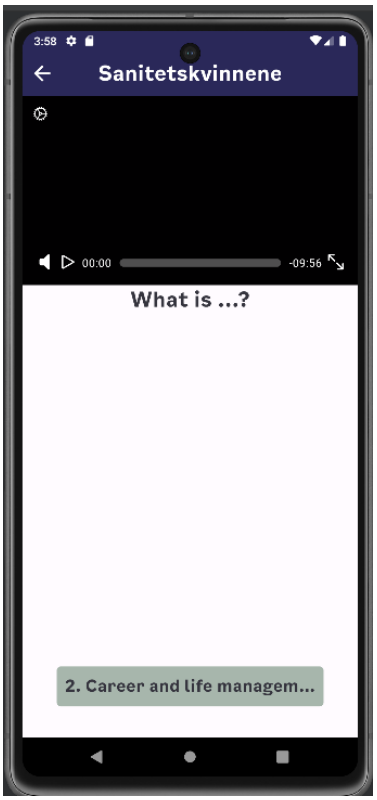


Figure 19.-*VideoPlayerPage for both sections Career and Health*

3.2.3.3 Back-End

A backend can often consist of CRUD operations: create, read, update, and delete. Our project only implements Read. This feature consists of reading the video URLs from the database. The reasons for us not implementing the other operations are multiple. End-users should only have the option to read data. The administrator, or P.O., has the videos produced on computers through an AI video-generating site called 'Synthesia.' When the P.O. wants to create, update, or delete video files in the database, they access the database on the computer and alter it directly there.

Since none of the videos are created on the phone, there is little point in implementing methods for such operations on that platform. The content will be fairly static and, therefore, rarely changed. The database will be better described in the section below, but one can say that it is user-friendly, making manually uploading files directly to it very efficient.

3.2.3.3.1 Initializing the Database

From the process of the pre-project report (Halsne et al., 2024), we knew that we only needed a database for uploading and fetching videos to the application. Some of the group members had experience with Google Firebase and knew that they could offer a solution that was easy to use with good documentation and scalable storage for a reasonable fee. Since we only needed to store video files, we decided to use Firebase Storage, which offers an easy structure of folders and files called 'buckets.'

To access the content of a bucket, one needs a reference. This reference can be created by adding the application credentials to Firebase Storage or by referencing the full path to the bucket through a URL reference (Google for Developers, n.d.). We added the application credentials to the storage so that we only need to create an instance of Firebase to have access to the storage. When the instance is obtained, one can choose to provide a full path to a video to fetch that specific video URL or provide a path to a folder and iterate through underlying folders and files to get all the video URLs.

It was necessary for the user to have read access to the database without having to authenticate or authorize This was possible to change in the rules file for the database (see *Figure 20*). This file also states that authorized users have access to upload files. At this point, this feature is not included in the application but can be included in the future.

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      // Allow anyone to read from storage
      allow read: if true;

      // Allow write access if the user is authenticated
      allow write: if request.auth != null;
    }
  }
}
```

Figure 20.- *Photo of the rules from Firebase Storage*

The bucket for the application contains a folder at root called ‘videos.’ This folder contains 14 sub-folders that are named based on the different languages the application supports, for example, ‘English,’ ‘Norwegian,’ and so on. The structure inside these folders is the same. They contain the sub-folders ‘Career’ and ‘Health.’

These folders again consist of their sub-folders until one reaches the videos of topics in their respective chapters. For example, the section ‘Career’ consists of 5 chapters. One of them is called ‘Me in Context.’ This chapter consists of 11 topics. If one wanted to retrieve the video URLs for all these topics, the path for providing the reference would be: ‘/videos/engelsk/karriere/1.Meg_I_Kontekst’.

3.2.3.3.2 Security

By utilizing Google Cloud Storage as the host for the database, data is automatically encrypted in transit between the application and the database (*Trust Center - Security and Compliance*, n.d.). Additionally, with the implementation of strict access controls, where each user is granted only read access to the database, both the application and user actions are further secured. This combination of encryption in transit and limited user access ensures robust protection of data integrity and confidentiality.

3.2.3.3.2 Code Snippet Description

In this section, most methods return values wrapped in a Future<>. This design choice enhances application responsiveness and efficiency by taking advantage of asynchronous operations. Utilizing Futures enables Flutter to seamlessly execute tasks like fetching data from the internet, reading files, or performing complex computations without obstructing the UI. When processing data from the database for UI rendering, our classes display a loading indicator to signify progress. Each method invocation from the UI to the

data file is accompanied by a.then() callback, ensuring that main thread blocking is avoided. This approach ensures timely data delivery without hindering the main thread, which continues to render the loading indicator until the data is ready for display.

Most of the methods presented below require strings, lists of an object type, and/or nested lists in their parameters. To ensure robust logic, every method has wrapped its code in a try-catch block. When returning the data back to the UI, it would often be in the form of a string or list of an object type. As stated in the previous high-fidelity version, the *career_data.dart* file contains the data and methods for handling the logic in the pages related to career. Though some of the methods being used here are also invoked in the *health_data.dart* file. Some of the methods written in these two files will again be explained later in the sub-chapter *3.2.3.4* Translations. All of the method illustrations that are not present in this section, can be found in *Appendix E: Technical Specifications* instead.

Throughout our application, we utilize a single database method located in the *database_service.dart* file: *fetchAllVideos* (see *Figure 21*). This method recursively iterates through the reference, navigating through folders and files until all are processed. When encountering a video file, the method invokes *getDownloadUrl()* to fetch the video's URL.

```dart
//This method takes in string that is a folderpath in the parameter.
//In the logic inside it calls itslef recursively in order to fetch all
// of the folders and references contained inside the first path.
//It then returns a list of video urls as strings in an array.
Future<List<String>> fetchAllVideos(String folderPath) async {
  try {
    List<String> downloadUrls = [];
    print("This is folderPath: $folderPath");
    Reference videosFolderRef = storage.ref(folderPath);

    ListResult result = await videosFolderRef.listAll();

    List<Future<List<String>>> futures = [];
    for (var folder in result.prefixes) {
      print("Folder: $folder");
      String fullPath = folder.fullPath;
      print("FullPath: $fullPath");

      // Collect futures from recursive calls
      futures.add(fetchAllVideos(folder.fullPath));
    }

    for (var item in result.items) {
      String downloadUrl = await item.getDownloadURL();
      print('Download URL: $downloadUrl');
      downloadUrls.add(downloadUrl);
    }
    // Wait for all recursive calls to complete
    List<List<String>> subFolderUrls = await Future.wait(futures);

    // Wait for all recursive calls to complete
    List<List<String>> subFolderUrls = await Future.wait(futures);

    // Flatten the nested list hiearchy:
    for (var urls in subFolderUrls) {
      downloadUrls.addAll(urls);
    }

    print("This is downloadUrls: $downloadUrls");
    return downloadUrls;
  }
  catch (e){
    print("Error in fetchAllVideos(): $e");
    return [];
  }
}

}
```

Figure 21.- *fetchAllVideos() method in database_service.dart file*

groupAndTranslateWithoutHeadlinesCareer() is a method that returns a list that contains all the video titles in the current chosen language (see *Figure 22*). In order to create this list, we first fetch a list of all the titles used in career, then we filter on if the string consists of at least two or three numbers separated by dots. If a string matches one of those requirements, then we know for sure that it is a video title.

```dart
//This method returns a list of strings that are all the video topics in career.
//For the different languages one has to use different regexes in order to traget
// and find the right strings to add to the list:
Future<List<String>> groupAndTranslateWithoutHeadlinesCareer() async {
  try {
    List<String> items = groupAndTranslateSubModulesOrVideosTittle(CareerItems, "")
    List<String> itemResult = [];
    String language = await getCurrentLanguage();

    RegExp regex1;
    RegExp regex2;

    if (language == "arabisk") {
      regex1 = RegExp(r'^[٠-٩]+\.[٠-٩]+\s');
      regex2 = RegExp(r'^[٠-٩]+\.[٠-٩]+\S');
    }
    else if (language == "pashto" || language == "urdu") {
      regex1 = RegExp(r'^[۰-۹]+\.[۰-۹]+\s');
      regex2 = RegExp(r'^[۰-۹]+\.[۰-۹]+\S');
    }
    else if(language == "farsi" || language == "dari"){
      regex1 = RegExp(r'^[۰-۹]+\.[۰-۹]+\s');
      regex2 = RegExp(r'^[۰-۹]+\.[۰-۹]+\S');
    }
    else if(language == "tamil"){
      regex1 = RegExp(r'^[௦-௯]+\.[௦-௯]+\s');
      regex2 = RegExp(r'^[௦-௯]+\.[௦-௯]+\S');
    }
    else if(language == "thai"){
      regex1 = RegExp(r'^[๐-๙]+\.[๐-๙]+\s');
      regex2 = RegExp(r'^[๐-๙]+\.[๐-๙]+\S');
    }
    else{
      regex1 = RegExp(r'^\d+\.\d+\s');
      regex2 = RegExp(r'^\d+\.\d+\S');
    }

    for (String item in items) {

      final match1 = regex1.firstMatch(item);
      final match2 = regex2.firstMatch(item);

      if (match1 != null) {
        itemResult.add(item);
      }
      else if (match2 != null) {
        itemResult.add(item);
      }
    }

    return itemResult;
  }
  catch (e) {
    print('Error in groupAndTranslateWithoutHeadlinesCareer: $e');
    return [];
  }
}
```

Figure 22.- *groupAndTranslateWithoutHeadlinesCareer() method*

GetVideoUrlsCareer and *getVideoUrlsHealth()*, only needs the first string of the list of items that are about to be rendered in the UI. The filtered match sets a path for fetching the videos. This differs for the method *getAllVideoUrls()*, which provides a path where all videos for the current language are fetched (see *Figure 23)*. In all these methods, the path is

passed to the previously explained method *fetchAllVideos()*. The list returned from that method is provided back to the UI.

nextVideoItems() is a method that becomes utilized in both Career pages and Health pages. It takes in two lists in the parameter: URL and title. These lists have to be the same size, if that is not the case, then the method returns an empty list back to the UI class. In the case where the lists are the same size, we start iterating through the lists from the back. At each iteration a *videoItem* is created, where it stores the previous item iterated in addition to title and URL. By doing this, we ensure that each object knows the next object. When this iteration is done, we iterate through the list again to reverse the whole list.

```
479   //This list simply aims to retrieve all the videoUrls in Career and return them back to the view
480   Future<List<String>> getAllVideoUrls() async{
481     try{
482       Database database = Database();
483       List<String> videoUrls = [];
484
485       String currentLanguage =  await getCurrentLanguage();
486
487       String videoPath = "videos/" + currentLanguage + "/karriere/";
488
489       videoUrls = await database.fetchAllVideos(videoPath);
490
491       print("Returning Videos");
492
493       return videoUrls;
494     }
495     catch (e){
496       print('Error in getAllVideoUrls(): $e');
497       return [];
498     }
499   }
```

Figure 23.- *getAllVideoUrls() method displayed*

nextVideoItemsNested() takes in a list of video items and a list of video topics for iterating. While iterating we check if the numbers at the beginning of the video topics contain the same first number, if not, we know that it is a different chapter. We then add a new nested list into the list and add the following iterated items. This goes on until we have iterated through the titles and video items.

getAllVideoUrlsNested() takes advantage of the structure of *nextVideoItemsNested()*. The method iterates through it where it adds the URLs consecutively (see *Figure 24*). By doing this, we got a nested list with the URLs for the current object, and a list with the next video item, both for the UI.

```dart
Future<List<List<String>>> getAllVideoUrlsNested(List<List<VideoItem>> videoItemsNested, List<String> urls) async{
  try{
    List<List<String>> videoUrlsNested = [];
    int i = 0;

    for (List<VideoItem> videoList in videoItemsNested) {
      videoUrlsNested.add([]);

      for(VideoItem video in videoList){
        videoUrlsNested[i].add(urls.first);
        urls.remove(urls.first);
      }

      i++;
    }

    return videoUrlsNested;
  }
  catch (e){
    print('Error in getAllVideoUrlsNested(): $e');
    return [];
  }
}
```

Figure 24.- *Code for getAllVideoUrlsNested() method*

The *getDataAboutUserHaveSeenVideos()*, *getDataAboutUsersHaveSeenVideosOtherLanguage()*, and *getDataAboutUserHaveSeenVideosHealth()* methods works in similar ways. They all iterate through lists of headings to check if the user has seen the videos or not. Returning a list of Booleans to set the green ticks in the UI.

getCurrentLanguage() are used in both the *career_data.dart* and the *health_data.dart* files (see *Figure 25*). This method fetches the integer representing the current language. It then passes the value to a check to see which language string the integer matches. It then returns the string containing the language.

```dart
//A method that returns the currentLanguage in the application:
Future<String> getCurrentLanguage() async{
  String currentLanguage = "";
  try {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    int? language = prefs.getInt('selectedLanguage');
    if (language == 0) {
      currentLanguage = "engelsk";
    }
    else if (language == 1) {
      currentLanguage = "spansk";
    }
    else if (language == 2) {
      currentLanguage = "swahili";
    }
    else if (language == 15) {
      currentLanguage = "tigrinja";
    }
    else {
      print("getCurrentLanguge: Den er ikke en del av denne listen");
    }
    return currentLanguage;
  }
  catch (e){
    print('Error in getCurrentLanguage(): $e');
    return currentLanguage;
  }
}
```

Figure 25.- *getCurrentLanguage() method*

getHeadlineNotTranslatedHealth() is a method used in the HealthDetailsPageNew class and which can be found in *Figure 26*. It uses the required data to iterate through the list which contains the translated headings and compares the headings to the string that contains a heading. When they are a match, we know the position of the heading, and use it to return the untranslated version to the UI.

getCareerProgress() and *getHealthProgress()* are methods that are almost completely similar (see *Figure 27*). The only difference is that they fetch different lists in the beginning of the method in order to iterate through all video topics under their respective sections. For each video topic that is saved as true, one is applied to a sum. When the iteration is done, the returned sum is equal to itself divided on the amount of video titles that were iterated through.

```
String getHeadlineNotTranslatedHealth(List<String> translatedResult, String translatedHeading, List<String> unTranslatedResult){
  try{
    String result = "";
    int i = 0;

    for(String heading in translatedResult){
      if(heading == translatedHeading){
        result = unTranslatedResult[i];
        break;
      }
      i++;
    }

    print("This is result before returning: $result");

    return result;
  }
  catch (e){
    print('Error in getHeadlineNotTranslatedHealth(): $e');
    return "";
  }
}
```

Figure 26.- *getHeadlineNotTranslatedHealth() method*

```
Future<double> getCareerProgress() async {
  double result = 0.0;
  try {
    // Initialize SharedPreferences
    SharedPreferences prefs = await SharedPreferences.getInstance();

    // Get translated items
    List<String> itemResult = await groupAndTranslateWithoutHeadlinesCareer();

    // Check if itemResult is empty
    if (itemResult.isEmpty) {
      throw Exception('itemResult is empty.');
    }

    // Calculate career progress
    for (String item in itemResult) {
      bool? checkValue = prefs.getBool(item);
      if (checkValue != null && checkValue) {
        result++;
      }
    }

    // Perform division
    result = result / itemResult.length;

    return result;
  }
  catch (e) {
    print('Error in groupAndTranslateDetailsWithoutChaptersCareer: $e');
    return result;
  }
}
```

Figure 27.- *Displays the logic of getCareerProgress(), and much of the logic of the very similar getHealthProgress() method.*

3.2.3.4 Translations

First and foremost, it is essential to recognize that implementing translation or localization functionality within a Flutter application involves operations on both the front-end and the back-end. Translated text presentation is a core front-end feature, focusing on the UI of the application. In contrast, the text retrieval process heavily relies on back-end methods for managing string fetching, sorting, and translation processes. As such, the text retrieval process is primarily considered a back-end operation.

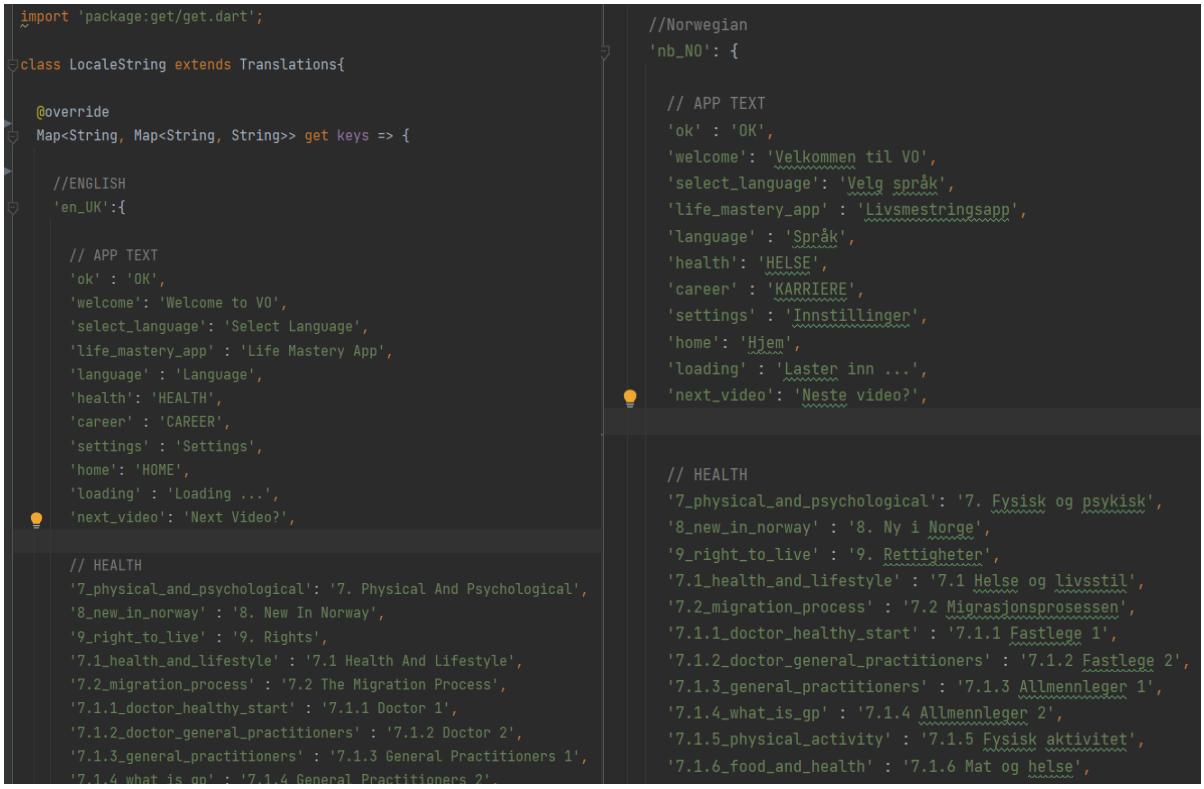
However, other operations, such as localization implementation and enabling language selection features, can be seen as primarily within the domain of the front-end. The following section provides a detailed explanation of the steps necessary to enable translation and language selection features within the application.

3.2.3.4.1 Localization Implementation

To facilitate text translation within the project, a method or getter named *getKeys* must be implemented in the *LocaleString.dart* file. This method accepts two parameters: a String variable serving as a key for the map, and an inner map assigned as the value. The inner map also takes two parameters: String variables serving as keys, and a String variable as the value.

The keys returned by the *getKeys* method are Locale values represented as strings. These Locale values are crucial for detecting the language set for the app. For instance, ‘en_UK’ represents the English language used in the United Kingdom, while ‘nb_NO’ represents the Bokmål form of the Norwegian language.

The app employs 12 maps, each corresponding to one of the 12 supported languages: English, Norwegian, Spanish, Swahili, Somali, Ukrainian, Arabic, Pashto, Kurmanci, Turkish, Tigrinya, and Urdu. These maps store text data in various languages, with each language associated with a String-formatted Locale. To retrieve text in the desired language, the app utilizes a two-step process. First, it retrieves the Locale key from the map using the *getKeys* method (getter), indicating the language. Then, it uses this key along with identical keys within each inner map to identify and display the appropriate text corresponding to the set language in the app (see *Figure 86*). The task of translating the text for each of the languages was facilitated using both ChatGPT 3.5 & Google Translate, except those that were provided by the P.O.



```dart
import 'package:get/get.dart';

class LocaleString extends Translations{

  @override
  Map<String, Map<String, String>> get keys => {

    //ENGLISH
    'en_UK':{

      // APP TEXT
      'ok' : 'OK',
      'welcome': 'Welcome to VO',
      'select_language': 'Select Language',
      'life_mastery_app' : 'Life Mastery App',
      'language' : 'Language',
      'health': 'HEALTH',
      'career' : 'CAREER',
      'settings' : 'Settings',
      'home': 'HOME',
      'loading' : 'Loading ...',
      'next_video': 'Next Video?',

      // HEALTH
      '7_physical_and_psychological': '7. Physical And Psychological',
      '8_new_in_norway' : '8. New In Norway',
      '9_right_to_live' : '9. Rights',
      '7.1_health_and_lifestyle' : '7.1 Health And Lifestyle',
      '7.2_migration_process' : '7.2 The Migration Process',
      '7.1.1_doctor_healthy_start' : '7.1.1 Doctor 1',
      '7.1.2_doctor_general_practitioners' : '7.1.2 Doctor 2',
      '7.1.3_general_practitioners' : '7.1.3 General Practitioners 1',
      '7.1.4_what_is_gp' : '7.1.4 General Practitioners 2',
```

```dart
    //Norwegian
    'nb_NO': {

      // APP TEXT
      'ok' : 'OK',
      'welcome': 'Velkommen til VO',
      'select_language': 'Velg språk',
      'life_mastery_app' : 'Livsmestringsapp',
      'language' : 'Språk',
      'health': 'HELSE',
      'career' : 'KARRIERE',
      'settings' : 'Innstillinger',
      'home': 'Hjem',
      'loading' : 'Laster inn ...',
      'next_video': 'Neste video?',

      // HEALTH
      '7_physical_and_psychological': '7. Fysisk og psykisk',
      '8_new_in_norway' : '8. Ny i Norge',
      '9_right_to_live' : '9. Rettigheter',
      '7.1_health_and_lifestyle' : '7.1 Helse og livsstil',
      '7.2_migration_process' : '7.2 Migrasjonsprosessen',
      '7.1.1_doctor_healthy_start' : '7.1.1 Fastlege 1',
      '7.1.2_doctor_general_practitioners' : '7.1.2 Fastlege 2',
      '7.1.3_general_practitioners' : '7.1.3 Allmennleger 1',
      '7.1.4_what_is_gp' : '7.1.4 Allmennleger 2',
      '7.1.5_physical_activity' : '7.1.5 Fysisk aktivitet',
      '7.1.6_food_and_health' : '7.1.6 Mat og helse',
```

Figure 28.- *LocaleString.dart file: getKeys getter. English and Norwegian Locale & map of strings.*

3.2.3.4.2 Text Presentation

To present translated text within the Flutter application, certain steps need to be followed. Moreover, careful considerations are essential to *enhancing* the UX. Below are some of the challenges that may arise in achieving a satisfactory text presentation.

Implementing localization or translation functionality in a Flutter application involves creating a *LocaleString.dart* file *and* extending the 'Translations' class or mixin from the 'get' package. Text strings marked with a '.tr' extension signify their readiness for translation. The *getKeys* method in *LocaleString.dart* manages automatic text retrieval for translation, organized within maps by Locale. The essence is partially explained, but *it* consists of the.tr string being passed to the correct maps, set by the Locale, for retrieving the correct string for the current view.

This streamlined approach ensures efficient integration of localization features within the application. Text *responsiveness* refers to the system or interface's capability to adapt text presentation according to screen size dimensions and user preferences configured in mobile settings. Responsive text dynamically adjusts within specified parameters, as demonstrated in *Figure 29*. Utilizing the *AutoSizeText* widget, text sizes can be made responsive, ensuring optimal readability across various device sizes.

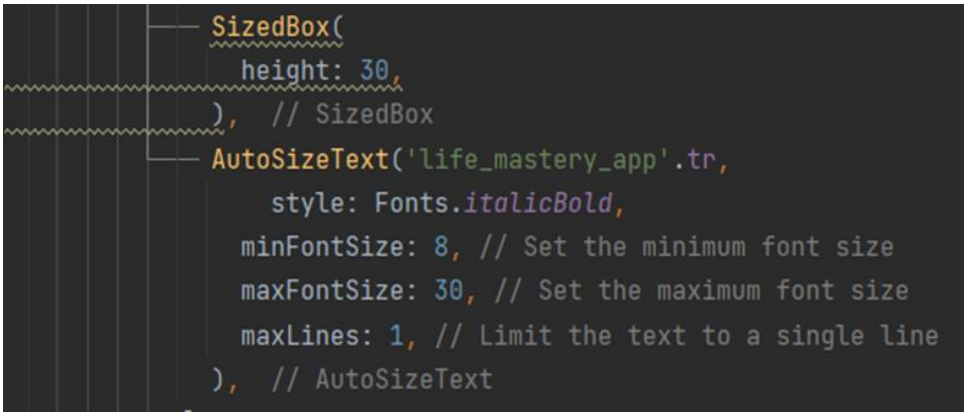


Figure 29.- *AutoSizeText widget*

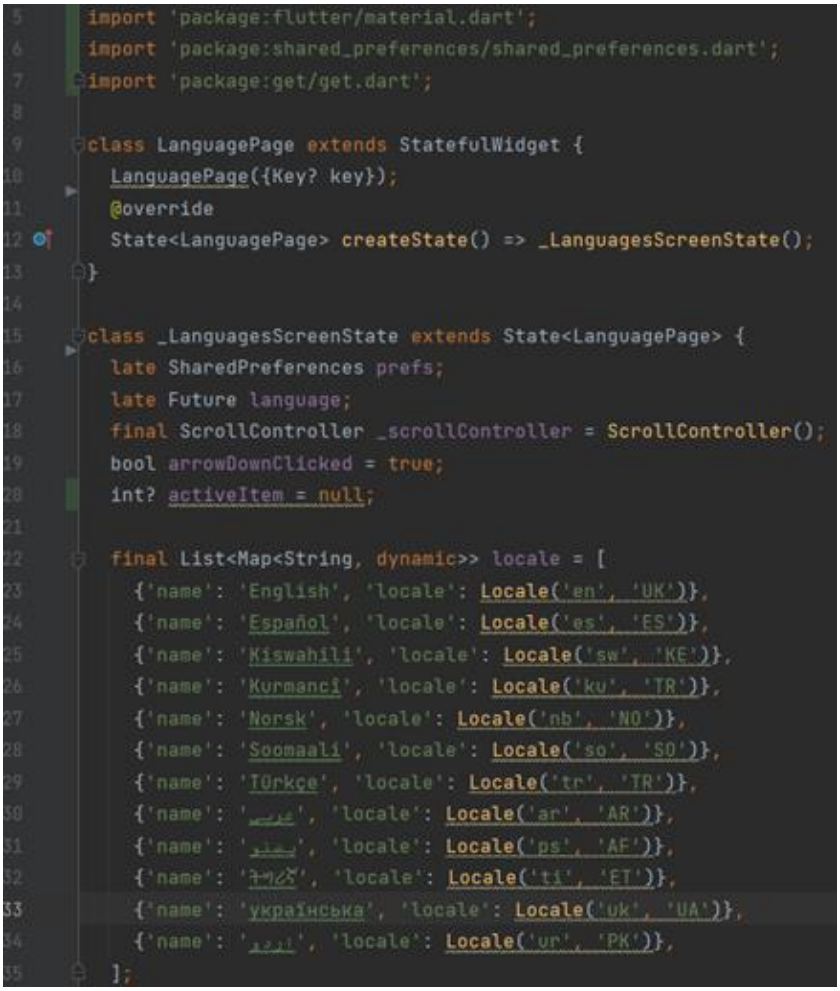
The *AutoSizeText* widget utilizes three essential parameters: 'minFontSize' sets the minimum font size for the text, 'maxFontSize' defines the maximum font size, and 'maxLines' determines the maximum number of lines for text display. Consequently, the text dimensions will dynamically adjust within the specified limits, accounting for factors such as the length of the string, mobile screen dimensions, and user preferences configured in the device settings.

3.2.3.4.3 Enabling Localization and Language Selection

Enabling localization, which includes language selection, can be achieved through the implementation of specific pages and methods. While there are various approaches to accomplish this, the following section elucidates the method through which the localization and language (Locale) selection feature is facilitated.

3.2.3.4.3.1 Language_page.dart file

The Language page was introduced in *Chapter 3.2.2. First High-Fidelity Prototype*, but the logic for implementing Locale was made in this iteration. In *Figure 30*, the list 'Locale' is depicted, which serves as a repository for mappings between language names and corresponding 'Locale' objects, encompassing all languages accessible within the application. Notably, the name 'Locale' consistently denotes a list of maps, where each map employs a String as its key, allowing for dynamic assignment of values of any type.



```dart
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
import 'package:get/get.dart';

class LanguagePage extends StatefulWidget {
  LanguagePage({Key? key});
  @override
  State<LanguagePage> createState() => _LanguagesScreenState();
}

class _LanguagesScreenState extends State<LanguagePage> {
  late SharedPreferences prefs;
  late Future language;
  final ScrollController _scrollController = ScrollController();
  bool arrowDownClicked = true;
  int? activeItem = null;

  final List<Map<String, dynamic>> locale = [
    {'name': 'English', 'locale': Locale('en', 'UK')},
    {'name': 'Español', 'locale': Locale('es', 'ES')},
    {'name': 'Kiswahili', 'locale': Locale('sw', 'KE')},
    {'name': 'Kurmancî', 'locale': Locale('ku', 'TR')},
    {'name': 'Norsk', 'locale': Locale('nb', 'NO')},
    {'name': 'Soomaali', 'locale': Locale('so', 'SO')},
    {'name': 'Türkçe', 'locale': Locale('tr', 'TR')},
    {'name': 'عربي', 'locale': Locale('ar', 'AR')},
    {'name': 'پښتو', 'locale': Locale('ps', 'AF')},
    {'name': 'ትግርኛ', 'locale': Locale('ti', 'ET')},
    {'name': 'українська', 'locale': Locale('uk', 'UA')},
    {'name': 'اردو', 'locale': Locale('ur', 'PK')},
  ];
```

Figure 30.- *Locale list in language_page.dart file*

Figure 31 illustrates the presentation of language names written in their native alphabets under the ‘name’ key. When users click or tap on a button from the list, featuring text corresponding to the ‘name’ value that is both understandable and readable, the application can identify the desired language. This is facilitated by the ‘Locale’ variable included in the same map, alongside the user-friendly name within the ‘Locale’ list of maps.

Figure 31.- *Language page displayed when the app runs for the first time*

The code snippet depicted in *Figure 32* demonstrates three key methods employed to manage the selected language in SharedPreferences and the app Locale. Firstly, the method *getSelectedLanguage()* initializes the SharedPreferences object 'prefs' and retrieves the stored index of the selected language. Next, within the *initState()* method, *getSelectedLanguage()* is asynchronously called to retrieve the selected language.

Further, *updateLanguage()* method takes two parameters: a 'Locale' and an integer. These parameters utilize the 'Locale' list, as shown in *Figure 32*, to obtain the selected Locale object and its index. This method then updates the selected language index in SharedPreferences, adjusts the app's Locale using *Get.updateLocale*, and navigates to the 'NavigationPage'.

```dart
Future<int?> getSelectedLanguage() async {
  prefs = await SharedPreferences.getInstance();
  activeItem = prefs.getInt('selectedLanguage');
}

@override
void initState() {
  super.initState();
  language = getSelectedLanguage();
}

void updateLanguage(Locale locale, int? index) {
  Get.updateLocale(locale);
  prefs.setInt('selectedLanguage', index!);
  Navigator.of(context).pushReplacement(MaterialPageRoute(
    builder: (context) => NavigationPage(updateLocale: updateLanguage),
  ));  // MaterialPageRoute
}
```

Figure 32.- *Functions in language_page.dart file*

3.2.3.4.3.2 Language_page_nav.dart file

The *language_page_nav.dart* file exhibits many similarities to the 3.2.3.4.3.1 Language_page.dart file discussed in detail above. However, once users select their preferred language upon the initial launch of the app, subsequent language changes can only be made through the *language_page_nav.dart* file.

Figure 33 illustrates a screenshot of the dialog box that appears when users click (or tap) the 'Select Language' button in the *language_page_nav.dart* file. The code snippet displayed on the right side of ***Error! Reference source not found.*** demonstrates the i mplementation of the asynchronous method *buildDialogLanguage()*, which returns a Locale representing the language selected through the dialog box shown in the screenshot on the left side of *Figure 33.*

The language list presented in the dialog box corresponds to the *localeSet* list shown in *Figure 30*, as explained in detail earlier. Specifically, the ListView within the dialog box displays items with values corresponding to the 'name' key in the mappings within the *localeSet* list. When a user clicks (or taps) on an item in the ListView to select a language, the *updateLanguage()* method is triggered to update the language. Subsequently, the dialog box is dismissed using the built-in Flutter method *Navigator.pop*, which closes the current dialog box screen and navigates the user back to the previous screen displaying either *language_page_nav.dart* or the 'Language' page.

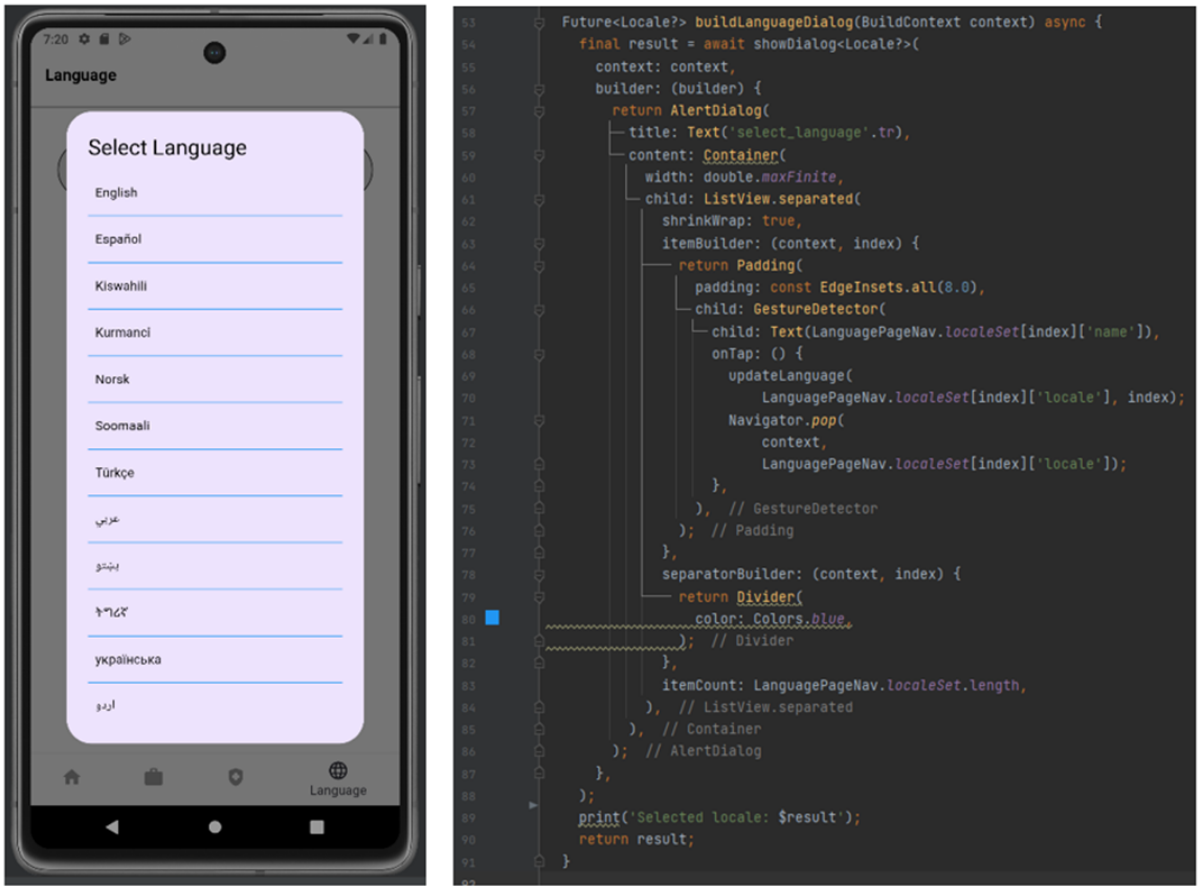
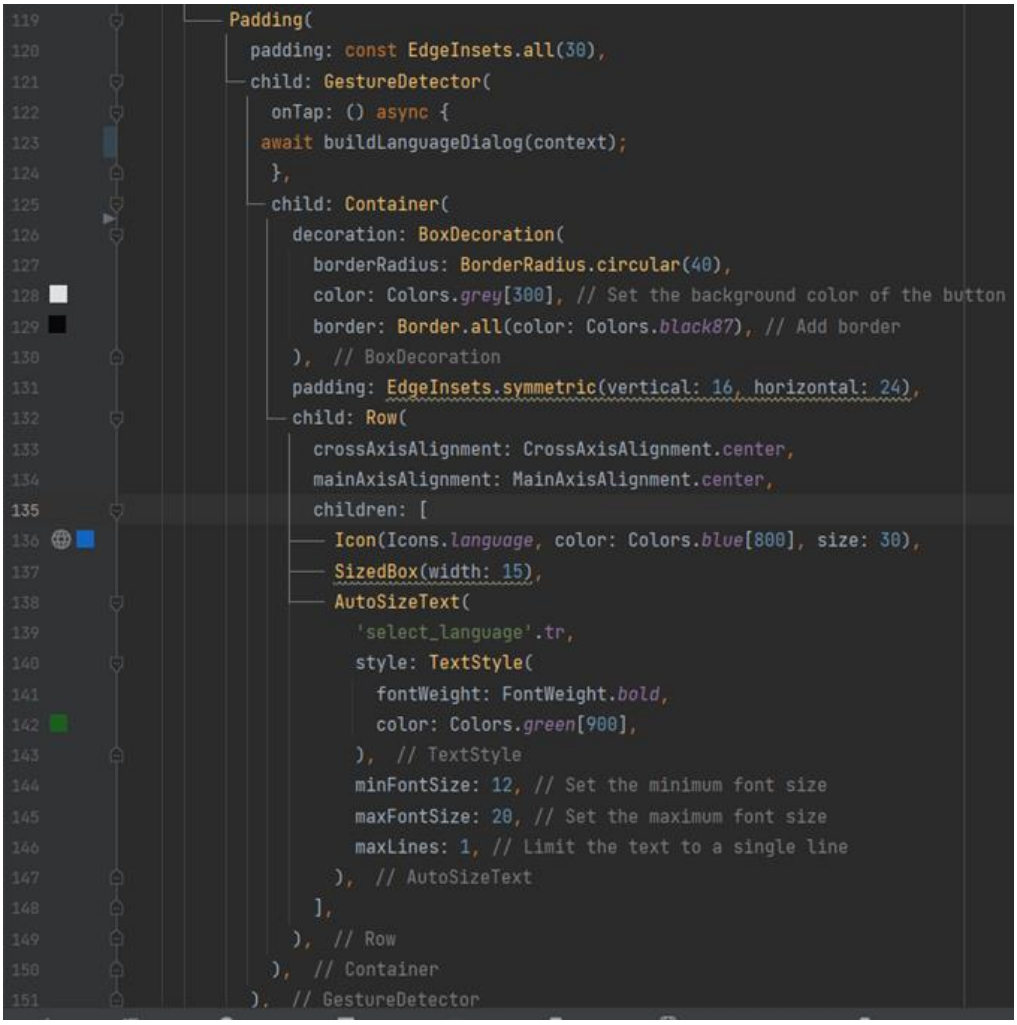


Figure 33.- *AlertDialog widget in language_page_nav.dart file*

Figure 34 illustrates a code snippet showcasing the implementation of a *GestureDetector* widget. This widget is utilized to invoke the *buildLanguageDialog()* method, as shown in *Figure 33* and elaborated upon earlier, utilizing the 'onTap' property. The *buildLanguageDialog()* method asynchronously updates the language (Locale) for the application when the user selects a language (Locale) from the ListView, as displayed in *Figure 33*. This ListView appears when the user taps on the row within the *language_page_nav.dart* file, featuring the text 'select_language'.tr.



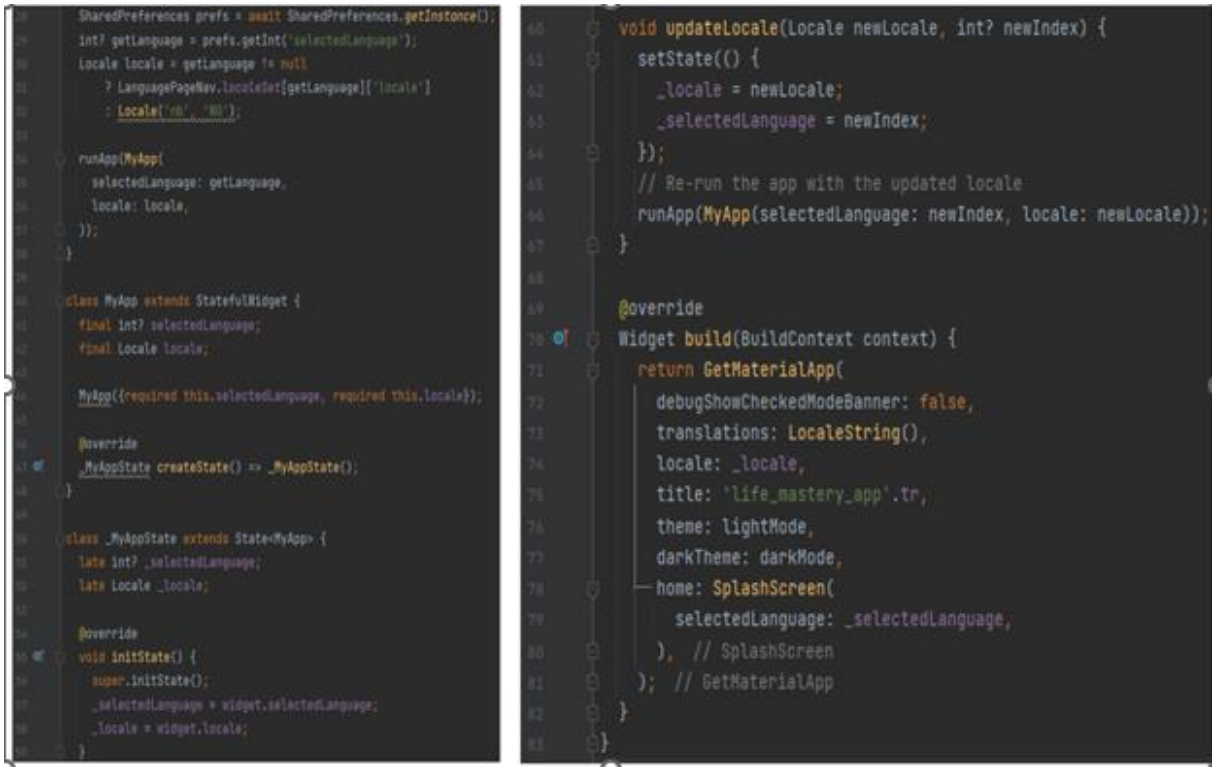
```
Padding(
  padding: const EdgeInsets.all(30),
  child: GestureDetector(
    onTap: () async {
    await buildLanguageDialog(context);
    },
    child: Container(
      decoration: BoxDecoration(
        borderRadius: BorderRadius.circular(40),
        color: Colors.grey[300], // Set the background color of the button
        border: Border.all(color: Colors.black87), // Add border
      ),  // BoxDecoration
      padding: EdgeInsets.symmetric(vertical: 16, horizontal: 24),
      child: Row(
        crossAxisAlignment: CrossAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Icon(Icons.language, color: Colors.blue[800], size: 30),
          SizedBox(width: 15),
          AutoSizeText(
            'select_language'.tr,
            style: TextStyle(
              fontWeight: FontWeight.bold,
              color: Colors.green[900],
            ),  // TextStyle
            minFontSize: 12, // Set the minimum font size
            maxFontSize: 20, // Set the maximum font size
            maxLines: 1, // Limit the text to a single line
          ),  // AutoSizeText
        ],
      ),  // Row
    ),  // Container
  ),  // GestureDetector
```

Figure 34.- *A GestureDetector widget in language_page_nav.dart file*

3.2.3.4.3 Main.dart file

Parts of this file have been covered in *Chapter*

3.2. First Iteration, and in the sub-chapter of the current chapter. ***Error! Reference source not found.*** illustrates the implementation of *MyApp*, a StatefulWidget responsible for managing language selection and Locale-related app states. The *MyApp* widget accepts two parameters: an integer representing the index of the selected language (Locale) within the 'Locale' list from *language_page.dart* and the *localeSet* list from *language_page_nav.dart*. Both lists contain identical indices corresponding to language (Locale) mappings. The second parameter of type Locale facilitates language detection, primarily through *language_page.dart* and *language_page_nav.dart*, providing users with language (Locale) selection options.



```
SharedPreferences prefs = await SharedPreferences.getInstance();
int? getLanguage = prefs.getInt('selectedLanguage');
Locale locale = getLanguage != null
    ? LanguagePageNav.localeSet[getLanguage]['locale']
    : Locale('en', 'US');

runApp(MyApp(
  selectedLanguage: getLanguage,
  locale: locale,
));
}

class MyApp extends StatefulWidget {
  final int? selectedLanguage;
  final Locale locale;

  MyApp({required this.selectedLanguage, required this.locale});

  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late int? _selectedLanguage;
  late Locale _locale;

  @override
  void initState() {
    super.initState();
    _selectedLanguage = widget.selectedLanguage;
    _locale = widget.locale;
  }
```

```
void updateLocale(Locale newLocale, int? newIndex) {
  setState(() {
    _locale = newLocale;
    _selectedLanguage = newIndex;
  });
  // Re-run the app with the updated locale
  runApp(MyApp(selectedLanguage: newIndex, locale: newLocale));
}

@override
Widget build(BuildContext context) {
  return GetMaterialApp(
    debugShowCheckedModeBanner: false,
    translations: LocaleString(),
    locale: _locale,
    title: 'life_mastery_app'.tr,
    theme: lightMode,
    darkTheme: darkMode,
    home: SplashScreen(
      selectedLanguage: _selectedLanguage,
    ), // SplashScreen
  ); // GetMaterialApp
}
}
```

Figure 35.- *Main.dart file*

Within the *MyAppState* class, two distinct methods handle the initialization and updating of the app's language (Locale). Initially, the selected language and Locale are set during the initialization phase, utilizing values passed from *MyApp* via the *initState()* method. Subsequently, the u*pdateLocale()* function manages the modification of the app's Locale, along with the index within the 'Locale' list in the 3.2.3.4.3.1 Language_page.dart file and the *localeSet* list in 3.2.3.4.3.2 *Language_page_nav.dart file*. This function, u*pdateLocale()*, employs the *setState()* method to update the integer index representing the Locale within both lists, as well as the Locale itself. Finally, the app is reloaded using the *runApp()* method to reflect the updated settings.

3.2.3.4.3.4 Splash_screen.dart file

Upon app launch, the *main.dart* file passes the latest selected language (Locale) to the *splash_screen.dart* file as a constructor parameter. The *splash_screen.dart* file (see *Figure 36*) is responsible for determining whether the app has been launched previously by checking the presence of a value in an integer variable representing the index for the language (Locale). This variable is sourced from both the 'Locale' list in the *language_page.dart* file and the *localeSet* list in the *language_page_nav.dart* file.

The check is performed to ascertain whether the user should be directed to the *language_page.dart* file for language selection during the app's initial launch. This index value is obtained from the *main.dart* file each time the app is launched. If it is not the initial launch, the app transitions to display the *navigation_page.dart* after the predetermined duration for displaying the splash screen elapses.

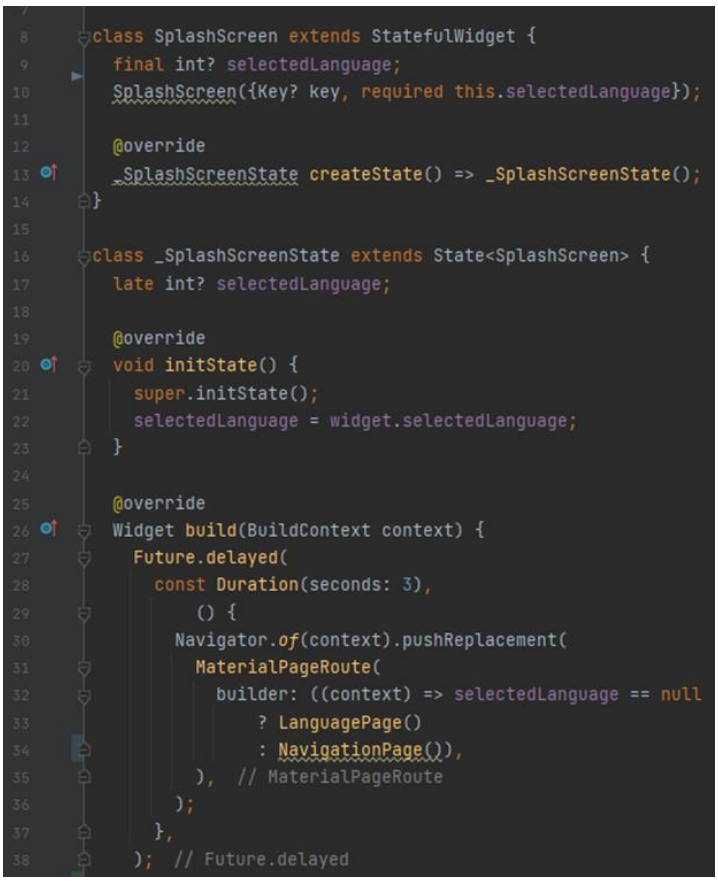


Figure 36.- *Splash_screen.dart file*

3.2.3.4.4 Text Retrieval Algorithm

The numbered translated strings referenced in the *LocaleString.dart* file are categorized into two groups, each intended for display on the career and health pages respectively. These strings are identified by numerical labels, corresponding to module and sub-module numbers. As detailed below, numerous data retrieval and sorting methods are implemented within the *career_data.dart* and *health_data.dart* files, primarily tasked with managing text retrieval algorithms across the career and health pages. For more detailed explanations of the described methods below, refer to *Appendix F: Code Explanations*.

3.2.3.4.4.1 Career_data.dart file

The *career_data.dart* file serves as the primary page responsible for managing retrieval and sorting algorithms for titles (strings/text) related to career pages. The CareerItems list illustrated in *Figure 37*, is employed to store strings that are also keys within

the inner maps managed by the *getKeys* method (getter) in the *LocaleString.dart* file. These strings, which simultaneously act as keys in the inner map of the *LocaleString.dart* file, are utilized to retrieve the correct titles on various career pages, translated into the selected language using the '.tr' extension.

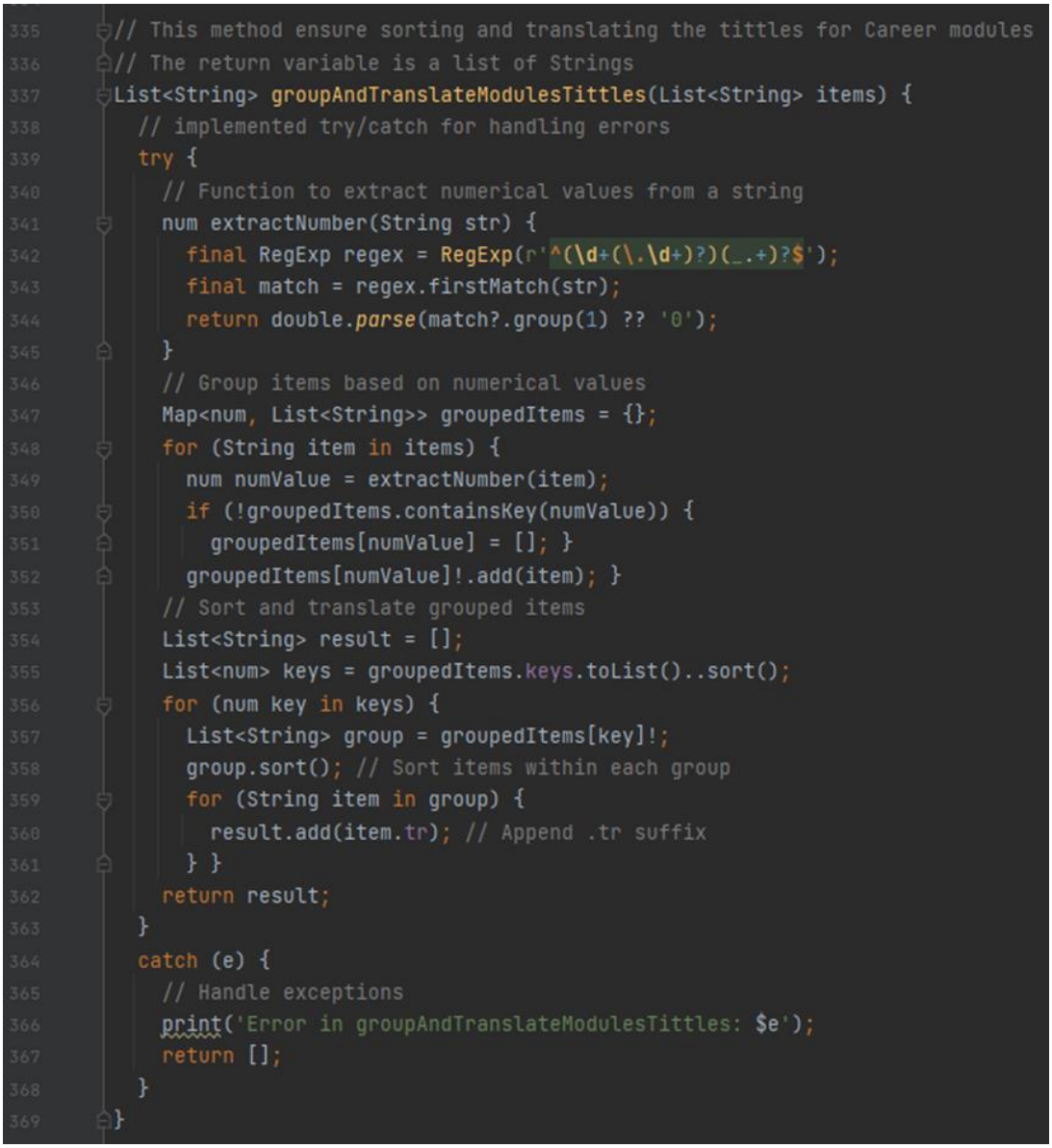
```dart
import 'package:get/get.dart';

List<String> CareerItems = [
  '1_me_in_context',
  '1.9_summary',
  '1.8_case_and_reflective',
  '1.7_roles',
  '1.6_reflective_task',
  '1.5_values',
  '1.4.1_task',
  '1.3_personal_qualities',
  '1.2_new_words',
  '1.4_interests',
  '1.3.1_reflective_task_personal_qualities',
  '1.1_introduction',
  '2_possibilties_and_limitations',
  '2.6_summary',
  '2.4.1_case_and_reflective',
  '2.2_new_words',
```

Figure 37.- *CareerItems list in career_data.dart file*

```dart
// This method ensure retrieving the tittles for Career modules
// The return variable is a list of Strings
List<String> getCareerModulesTittles(String state) {
  // implemented try/catch for handling errors
  try {
    //initiating list that will be returned
    List<String> result = [];
    // going through the CareerItems list
    for (String item in CareerItems) {
      //filtering CareerItems list to fetch Strings having only one digit
      if (item.length > 1 && item[1] == '_') {
        //appending the target string/tittle to the initiated list
        result.add(item);
      }
    }
    if(state == "Not_Translate"){
      //sending the list as a parameter to the sorting method
      return groupModulesTittles(result);
    }
    else{
      //sending the list as a parameter to the sorting and translating method
      return groupAndTranslateModulesTittles(result);
    }
  }
  catch (e) {
    // Handle exceptions
    print('Error in getCareerModulesTittles: $e');
    return [];
  }
}
```

Figure 38.- *getCareerModulesTittles method in career_data.dart file*

The *getCareerModulesTittles()* method, as depicted in *Figure 38*, primarily serves to fetch a list of strings representing the names/titles of modules on the 'Career Page.' This method accepts a string parameter to determine whether the returned list of strings should be translated.



```dart
// This method ensure sorting and translating the tittles for Career modules
// The return variable is a list of Strings
List<String> groupAndTranslateModulesTittles(List<String> items) {
  // implemented try/catch for handling errors
  try {
    // Function to extract numerical values from a string
    num extractNumber(String str) {
      final RegExp regex = RegExp(r'^(\d+(\.\d+)?)(_.+)?$');
      final match = regex.firstMatch(str);
      return double.parse(match?.group(1) ?? '0');
    }
    // Group items based on numerical values
    Map<num, List<String>> groupedItems = {};
    for (String item in items) {
      num numValue = extractNumber(item);
      if (!groupedItems.containsKey(numValue)) {
        groupedItems[numValue] = []; }
      groupedItems[numValue]!.add(item); }
    // Sort and translate grouped items
    List<String> result = [];
    List<num> keys = groupedItems.keys.toList()..sort();
    for (num key in keys) {
      List<String> group = groupedItems[key]!;
      group.sort(); // Sort items within each group
      for (String item in group) {
        result.add(item.tr); // Append .tr suffix
      } }
    return result;
  }
  catch (e) {
    // Handle exceptions
    print('Error in groupAndTranslateModulesTittles: $e');
    return [];
  }
}
```

Figure 39.- *groupAndTranslateModulesTittles method in career_data.dart file*

Figure 39 shows the *groupAndTranslateModulesTittles()* method, which accepts a list of strings as input. As its name implies, this method primarily organizes and translates the strings retrieved from the *getCareerModulesTittles()* and *getHealthModulesTittles()* methods, serving as keys for the titles of career and health modules. Consequently, it is employed to sort and translate module titles on both career and health pages. This functionality is integral to the *career_data.dart* and *health_data.dart* files, responsible for managing data associated with their respective pages.

```dart
// This method ensure sorting the tittles for the modules
// The return variable is a list of Strings
List<String> groupModulesTittles(List<String> items) {
  // implemented try/catch for handling errors
  try {
    // Function to extract numerical values from a string
    num extractNumber(String str) {
      final RegExp regex = RegExp(r'^(\d+(\.\d+)?)(_.+)?$');
      final match = regex.firstMatch(str);
      return double.parse(match?.group(1) ?? '0');
    }
    // Group items based on numerical values
    Map<num, List<String>> groupedItems = {};
    for (String item in items) {
      num numValue = extractNumber(item);
      if (!groupedItems.containsKey(numValue)) {
        groupedItems[numValue] = []; }
      groupedItems[numValue]!.add(item); }
    // Sort the grouped items
    List<String> result = [];
    List<num> keys = groupedItems.keys.toList()..sort();
    for (num key in keys) {
      List<String> group = groupedItems[key]!;
      group.sort(); // Sort items within each group
      for (String item in group) {
        result.add(item);
      } }
    return result;
  }
  catch (e) {
    // Handle exceptions
    print('Error in groupModulesTittles: $e');
    return [];
  }
}
```

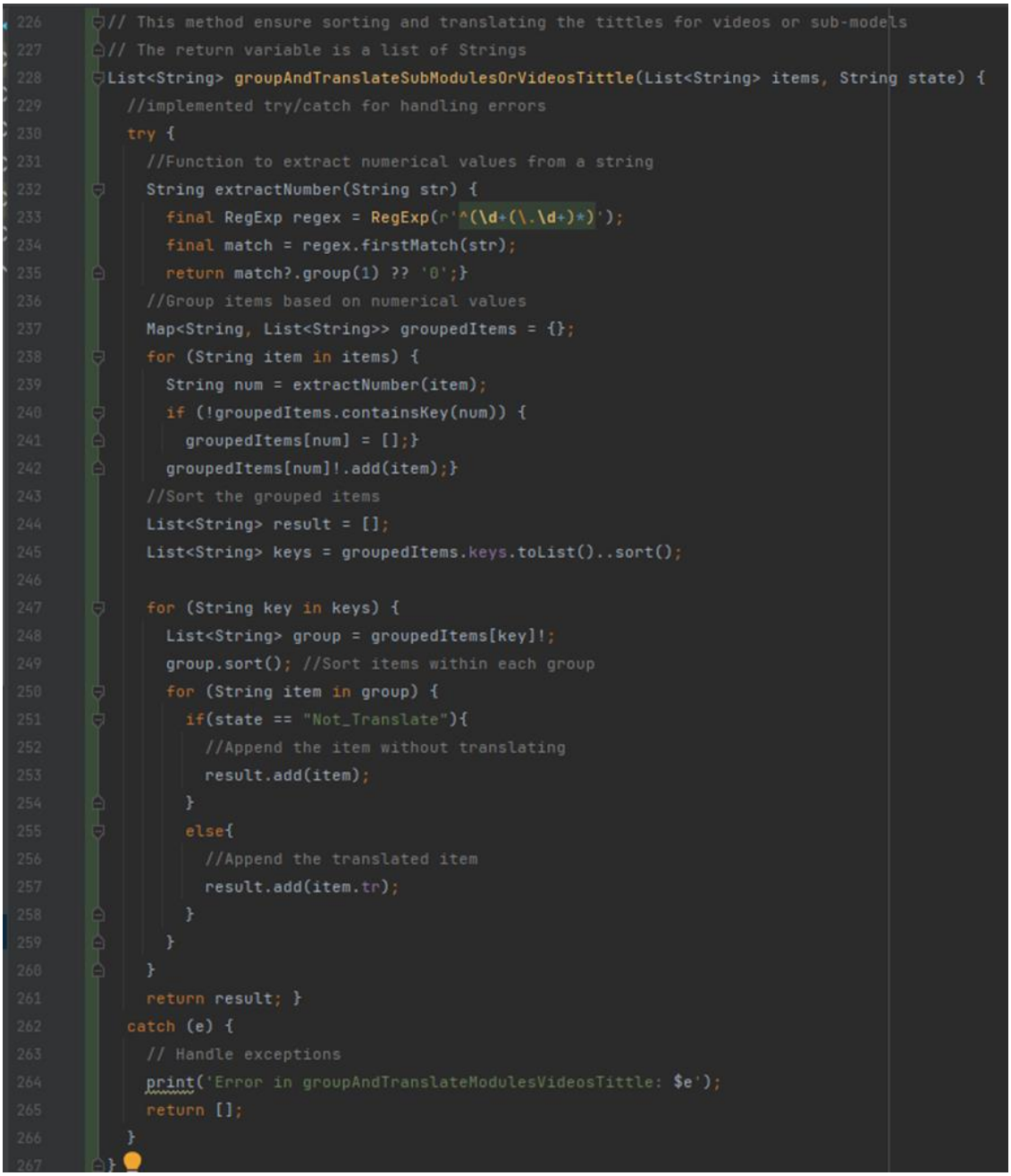
Figure 40.- *groupModuleTittles() method in career_data.dart file.*

In *Figure 40*, the method *groupModuleTittles()* is depicted, and is responsible for sorting a list of strings primarily containing module titles. This method operates similarly to the previously discussed *groupAndTranslateModulesTittles()* method. However, it is specifically designed to sort a list of strings representing module titles without performing any additional translation.

```dart
// This method ensure retrieving the tittles for career videos
// The return variable is a list of Strings
List<String> getCareerModulesVideosTittle(int index, String state){

  // implemented try/catch for handling errors
  try {
    //initiating list that will be returned
    List<String> result = [];
    //incrementing the input index by 1
    String searchIndex = '${index + 1}';
    // going through the CareerItems list
    for (String item in CareerItems) {
      //filtering CareerItems list to fetch Strings with decimal numberings
      if (item.startsWith(searchIndex) && item[1] == ".") {
        //appending the target string/tittle to the initiated list
        result.add(item);
      }
    }
    // sending the list as a parameter to the sorting and translating method
    return groupAndTranslateSubModulesOrVideosTittle(result, state);
  }

  catch (e) {
    // Handle exceptions
    print('Error in getCareerModulesVideosTittle: $e');
    return [];
  }
}
```

Figure 41.- *getCareerModulesVideosTittles method in career_data.dart file*

The *getCareerModulesVideosTittles()* method, as depicted in *Figure 41*, is utilized within the career pages to enumerate the contents of each career module, showcasing the titles of videos associated with each module. Essentially, this method generates a list of strings corresponding to the titles of the videos within each module. It accepts two parameters: an integer index representing the module title, and a string used to determine whether the returned list of strings should be translated.



```dart
// This method ensure sorting and translating the tittles for videos or sub-models
// The return variable is a list of Strings
List<String> groupAndTranslateSubModulesOrVideosTittle(List<String> items, String state) {
  //implemented try/catch for handling errors
  try {
    //Function to extract numerical values from a string
    String extractNumber(String str) {
      final RegExp regex = RegExp(r'^(\d+(\.\d+)*)');
      final match = regex.firstMatch(str);
      return match?.group(1) ?? '0';}
    //Group items based on numerical values
    Map<String, List<String>> groupedItems = {};
    for (String item in items) {
      String num = extractNumber(item);
      if (!groupedItems.containsKey(num)) {
        groupedItems[num] = [];}
      groupedItems[num]!.add(item);}
    //Sort the grouped items
    List<String> result = [];
    List<String> keys = groupedItems.keys.toList()..sort();

    for (String key in keys) {
      List<String> group = groupedItems[key]!;
      group.sort(); //Sort items within each group
      for (String item in group) {
        if(state == "Not_Translate"){
          //Append the item without translating
          result.add(item);
        }
        else{
          //Append the translated item
          result.add(item.tr);
        }
      }
    }
    return result; }
  catch (e) {
    // Handle exceptions
    print('Error in groupAndTranslateModulesVideosTittle: $e');
    return [];
  }
}
```

Figure 42.- *groupAndTranslateSubModulesOrVideosTittle method in career_data.dart file*

Figure 42 showcases the *groupAndTranslateSubModulesOrVideosTittle()* method, primarily tasked with sorting and translating the titles of each module's videos for the career pages. This method is employed to organize and translate video titles across both career and health pages. It is implemented within the '*career_data.dart*' and '*health_data.dart*' files, responsible for managing data pertinent to these pages. The method accepts two parameters: a list of strings and a string. The first parameter, a list of strings, denotes the titles to be sorted and potentially translated based on the value of the second parameter. The second parameter, a string, determines whether the returned list of strings should undergo translation.

```dart
// This method ensure retrieving the correct tittles for career modules
// The return variable is of type string which represent the translated module tittle
String getVideoPlayerCareerAppBarTittle(String tittle) {

  // implemented 'try/catch' block for handling errors
  try {
    //initiating a string variable that will be returned
    String result = "";
    // extracting the first digit of the input string
    String searchIndex = tittle[0];

    // going through the CareerItems list
    for (String item in CareerItems) {
      //filtering CareerItems list to fetch Strings having only one digit
      //the translated digit for the filtered string should also match the
      // first digit of the input string
      if (item[0].tr == searchIndex && item[1] == "_") {
        result = item; }
    }
    // returning the translated targeted module tittle
    return result.tr;
  }

  catch (e) {
    // Handle exceptions
    print('Error in getCareerAppBarTittle: $e');
    return "";
  }
}
```

Figure 43.- *getVideoPlayerCareerAppBarTittle method in career_data.dart file*

In *Figure 43*, the method *getVideoPlayerCareerAppBarTittle()* is demonstrated. This method takes a string as input and is primarily responsible for retrieving the corresponding module title for each video title provided as a parameter. If the corresponding title is found, then it is translated and returned.

3.2.3.4.4.2 health_data.dart file

The *health_data.dart* file serves as the central component responsible for managing retrieval and sorting algorithms pertinent to titles (Strings/text) associated with the 'Health Page'. Like in the beginning of the previous section about *career_data.dart*, *Figure 44* showcases important packages and a predefined list of strings that are crucial for the logic of translations.

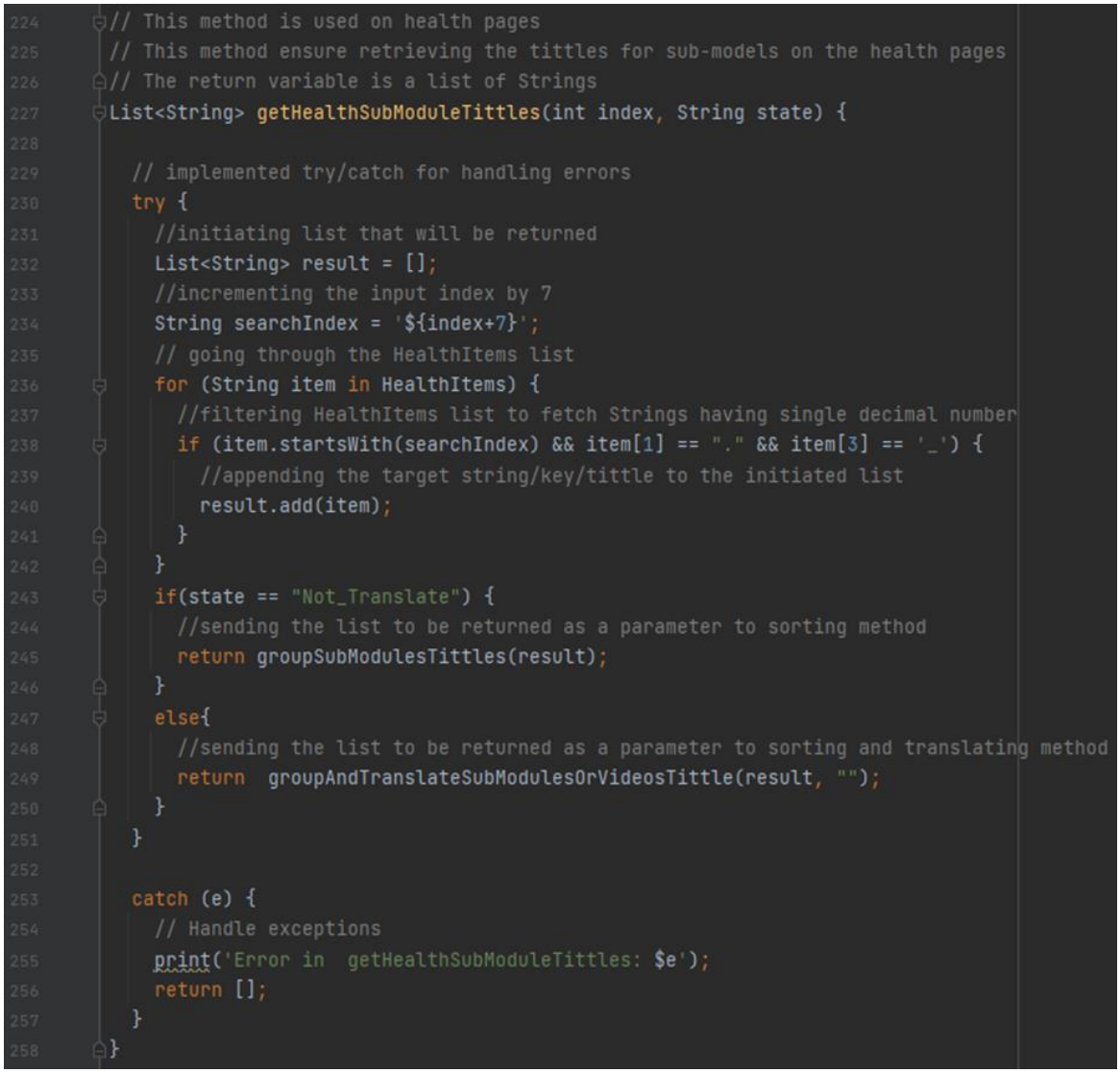
```dart
import 'package:livsmestringsapp/services/career_data.dart';
import 'package:get/get.dart';

List<String> HealthItems = [
  '7_physical_and_psychological',
  '8_new_in_norway',
  '9_right_to_live',
  '7.1_health_and_lifestyle',
  '7.2_migration_process',
  '7.1.1_doctor_healthy_start',
  '7.1.2_doctor_general_practitioners',
  '7.1.3_general_practitioners',
  '7.1.4_what_is_gp',
  '7.1.5_physical_activity',
  '7.1.6_food_and_health',
```

Figure 44.- *HealthItems list in health_data.dart file*

```dart
// This method ensure retrieving the tittles for Health modules
// The return variable is a list of Strings
List<String> getHealthModulesTittles() {

  // implemented try/catch for handling errors
  try {
    // initiating list that will be returned
    List<String> result = [];
    // going through the HealthItems list
    for (String item in HealthItems) {
      // filtering HealthItems list to fetch Strings having only one digit
      if (item.length > 1 && item[1] == '_') {
        // appending the target string/key/tittle to the initiated list
        result.add(item);
      }
    }
    // sending the list as a parameter to the sorting and translating method
    return groupAndTranslateModulesTittles(result);
  }

  catch (e) {
// Handle exceptions
    print('Error in getHealthModulesTittles: $e');
    return [];
  }
}
```

Figure 45.- *getHealthModulesTittles method in health_data.dart file*

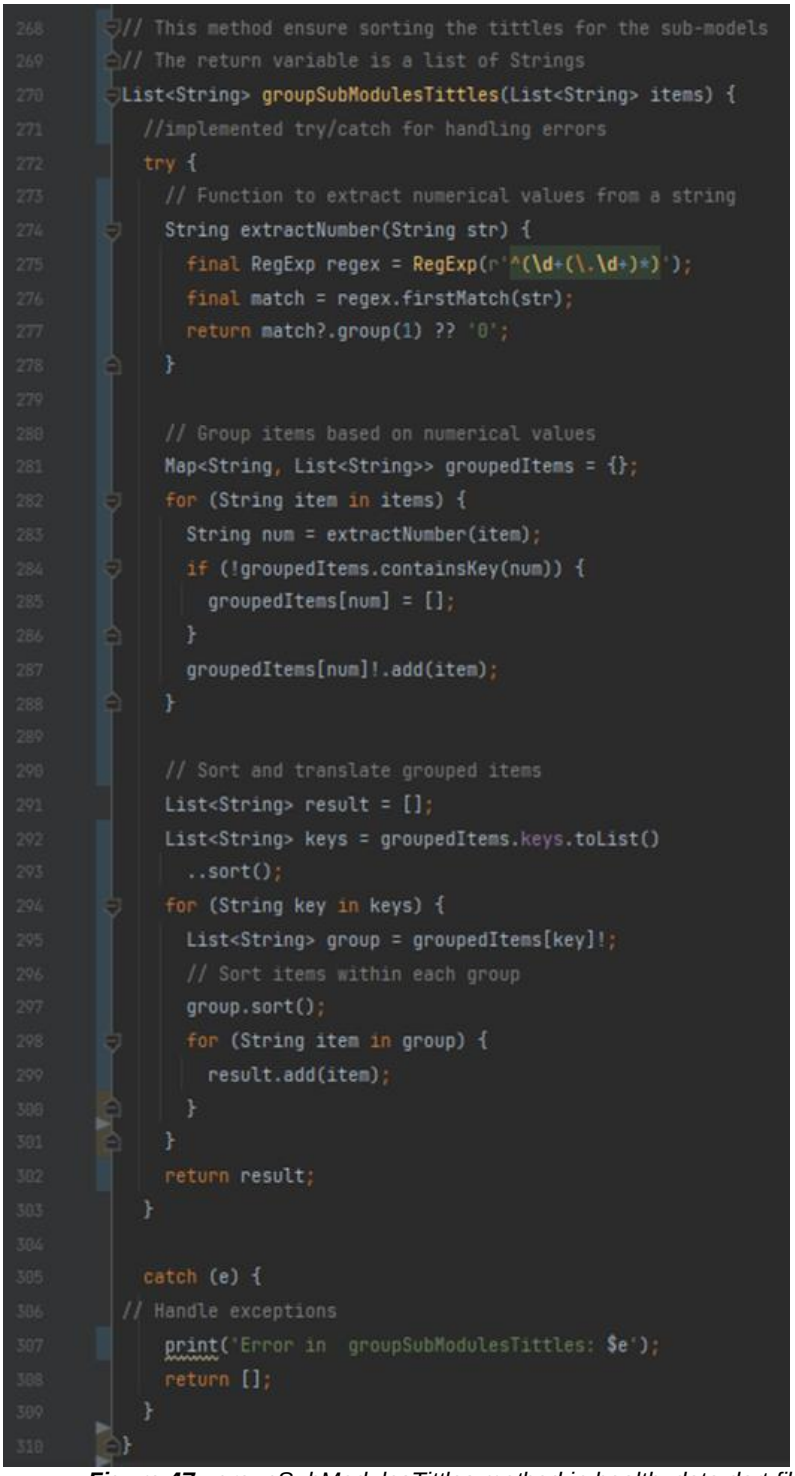
Figure 45 illustrates a code snippet for the method *getHealthModulesTittles()*, which is primarily responsible for fetching titles for modules on the health pages. Initially, a list of strings is initialized to store the titles of the modules, which will ultimately be returned by the method if no errors occur. Subsequently, a ‘for-loop’ iterates through the HealthItems list to retrieve the relevant strings.



```dart
// This method is used on health pages
// This method ensure retrieving the tittles for sub-models on the health pages
// The return variable is a list of Strings
List<String> getHealthSubModuleTittles(int index, String state) {

  // implemented try/catch for handling errors
  try {
    //initiating list that will be returned
    List<String> result = [];
    //incrementing the input index by 7
    String searchIndex = '${index+7}';
    // going through the HealthItems list
    for (String item in HealthItems) {
      //filtering HealthItems list to fetch Strings having single decimal number
      if (item.startsWith(searchIndex) && item[1] == "." && item[3] == '_') {
        //appending the target string/key/tittle to the initiated list
        result.add(item);
      }
    }
    if(state == "Not_Translate") {
      //sending the list to be returned as a parameter to sorting method
      return groupSubModulesTittles(result);
    }
    else{
      //sending the list to be returned as a parameter to sorting and translating method
      return  groupAndTranslateSubModulesOrVideosTittle(result, "");
    }
  }

  catch (e) {
    // Handle exceptions
    print('Error in  getHealthSubModuleTittles: $e');
    return [];
  }
}
```

Figure 46.- *getHealthSubModuleTittles method in health_data.dart file*

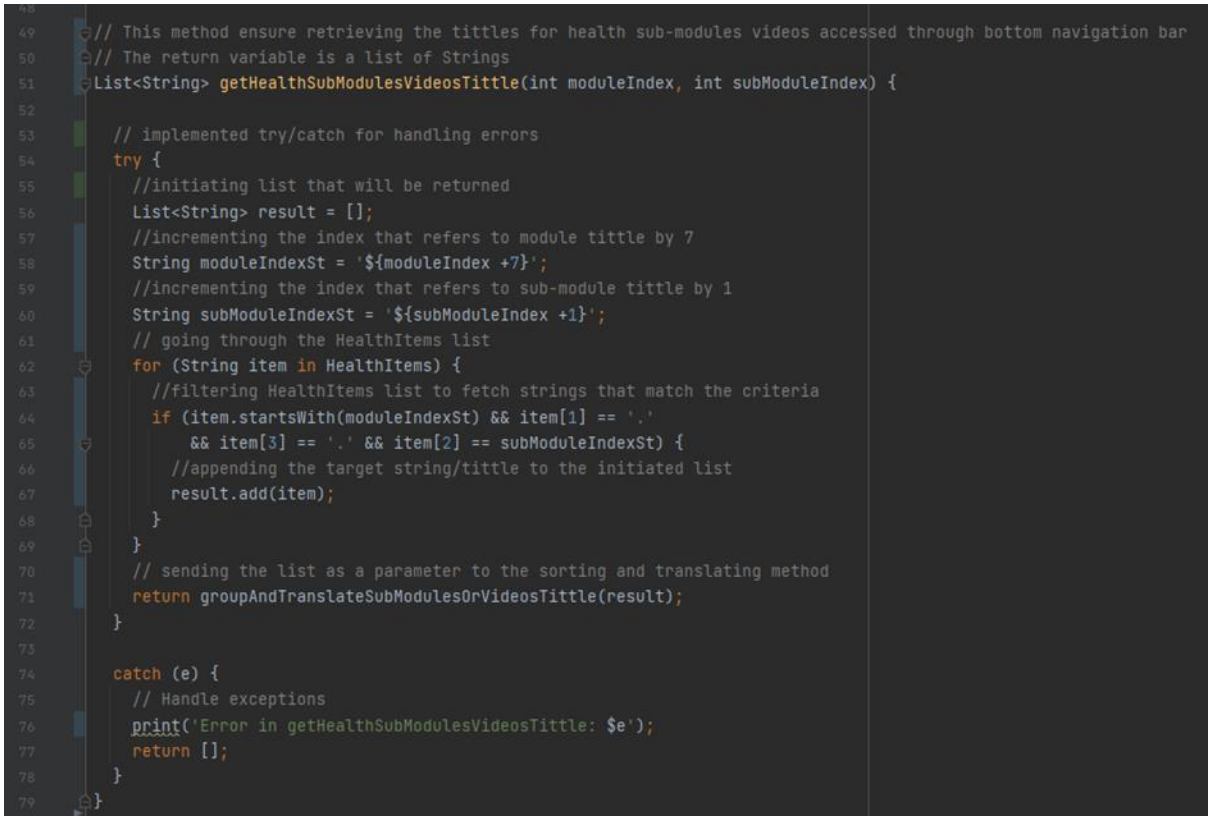
Figure 46 showcases the method *getHealthSubModuleTittles()*, employed to retrieve and display titles for sub-modules on health pages. The method accepts two parameters: the first, an integer representing the indices of the titles within the list that will be displayed using the method, and the second, a string indicating whether the strings in the list should be translated. Based on the state, the result is passed along to a new method before being returned.



```dart
// This method ensure sorting the tittles for the sub-models
// The return variable is a list of Strings
List<String> groupSubModulesTittles(List<String> items) {
  //implemented try/catch for handling errors
  try {
    // Function to extract numerical values from a string
    String extractNumber(String str) {
      final RegExp regex = RegExp(r'^(\d+(\.\d+)*)');
      final match = regex.firstMatch(str);
      return match?.group(1) ?? '0';
    }

    // Group items based on numerical values
    Map<String, List<String>> groupedItems = {};
    for (String item in items) {
      String num = extractNumber(item);
      if (!groupedItems.containsKey(num)) {
        groupedItems[num] = [];
      }
      groupedItems[num]!.add(item);
    }

    // Sort and translate grouped items
    List<String> result = [];
    List<String> keys = groupedItems.keys.toList()
      ..sort();
    for (String key in keys) {
      List<String> group = groupedItems[key]!;
      // Sort items within each group
      group.sort();
      for (String item in group) {
        result.add(item);
      }
    }
    return result;
  }

  catch (e) {
// Handle exceptions
    print('Error in  groupSubModulesTittles: $e');
    return [];
  }
}
```

Figure 47.- *groupSubModulesTittles method in health_data.dart file*

The method *groupSubModulesTittles()*, primarily serves to organize strings referring to the titles of sub-modules within health pages (see *Figure 47*). This method closely resembles the *groupAndTranslateSubModulesOrVideosTittle()* method found in the *career_data.dart* file, as discussed earlier. The key distinction lies in this method's omission of the translation step, specifically bypassing the appending of the '.tr' extension to the strings within the list being sorted and returned, provided there are no execution failures.



```dart
// This method ensure retrieving the tittles for health sub-modules videos accessed through bottom navigation bar
// The return variable is a list of Strings
List<String> getHealthSubModulesVideosTittle(int moduleIndex, int subModuleIndex) {

  // implemented try/catch for handling errors
  try {
    //initiating list that will be returned
    List<String> result = [];
    //incrementing the index that refers to module tittle by 7
    String moduleIndexSt = '${moduleIndex +7}';
    //incrementing the index that refers to sub-module tittle by 1
    String subModuleIndexSt = '${subModuleIndex +1}';
    // going through the HealthItems list
    for (String item in HealthItems) {
      //filtering HealthItems list to fetch strings that match the criteria
      if (item.startsWith(moduleIndexSt) && item[1] == '.'
          && item[3] == '.' && item[2] == subModuleIndexSt) {
        //appending the target string/tittle to the initiated list
        result.add(item);
      }
    }
    // sending the list as a parameter to the sorting and translating method
    return groupAndTranslateSubModulesOrVideosTittle(result);
  }

  catch (e) {
    // Handle exceptions
    print('Error in getHealthSubModulesVideosTittle: $e');
    return [];
  }
}
```

Figure 48.- *getHealthSubModulesVideosTittle() method in health_data.dart file*

Figure 48 displays the method *getHealthSubModulesVideosTittle()*, which retrieves strings representing the titles of videos for each sub-module on the health page. This method is employed to fetch video titles for the health page accessed through the bottom navigation bar. It takes two integer parameters, representing the indices for the module title and the sub-module title related to various lists on different health pages.

```dart
// This method ensure retrieving the tittles for health sub-modules videos accessed through home page
// The return variable is a list of Strings
List<String> getSubModuleIndexAndVideosTittle(List<String> subModulesTittles, String subModuleTittle, int moduleIndex) {
  // implemented try/catch for handling errors
  try {
    //initiating list that will be returned
    List<String> result = [];
    //integer variable that will identify sub-module index
    int? subModuleIndex = null;

    //filtering the sub-modules to fetch the index for the sub-module tittle clicked on
    for (int i = 0; i < subModulesTittles.length; i++) {
      if (subModuleTittle == subModulesTittles[i]) {
        //assigning the value of the sub-module's index to the already initiated integer
        subModuleIndex = i;
      }
    }
    //incrementing the index that refers to module tittle by 7
    String moduleIndexStr = '${moduleIndex + 7}';
    //incrementing the index that refers to sub-module tittle by 1
    String subModuleIndexStr = '${subModuleIndex! + 1}';
    // going through the HealthItems list
    for (String item in HealthItems) {
      //filtering HealthItems list to fetch strings that match the criteria
      if (item.startsWith(moduleIndexStr) && item[2] == subModuleIndexStr &&
          item[3] == '.') {
        //appending the target string/tittle to the initiated list
        result.add(item);
      }
    }
    //sending the list as a parameter to the sorting and translating method
    return groupAndTranslateSubModulesOrVideosTittle(result);
  }

  catch (e) {
  // Handle exceptions
    print('Error in getSubModuleIndexAndVideosTittle: $e');
    return [];
  }
}
```

Figure 49.- *getSubModuleIndexAndVideosTittle() method in health_data.dart file*

By displaying a code snippet for the method *getSubModuleIndexAndVideosTittle()*, (see *Figure 49*) it shows that it primarily utilizes to list video titles for sub-modules on the health page, accessed from the home page. This method, returning a list of strings, takes three parameters. The first parameter is a list of strings representing the titles of sub-modules for the associated module. The second parameter, a string type, denotes the title of the clicked sub-module by the user. The third parameter indicates the index of the module on the health page's list. These parameters are essential for identifying the video titles to be listed on the health page.

```dart
//This method ensure retrieving the correct tittles for health sub-modules
//The return variable is of type string which represent the translated sub-module tittle
String getVideoPlayerHealthAppBarTittle(String tittle) {
  // implemented 'try/catch' block for handling errors
  try {
    //initiating a string variable that will be returned
    String result = "";
    //extracting the first digit of the string that refers to the module tittle
    String searchIndex1 = tittle[0];
    //extracting the second digit of the string that refers to the sub-module tittle
    String searchIndex2 = tittle[2];

    // going through the HealthItems list
    for (String item in HealthItems) {
      //filtering HealthItems list to fetch Strings having only two digits
      //the translated digits for the filtered string should also match the
      //digits within the input string
      if (item[0].tr == searchIndex1 && item[2].tr == searchIndex2 && item[3] == "_") {
        result = item;
      }
    }
    // returning the translated targeted sub-module tittle
    return result.tr;
  }

  catch (e) {
    // Handle exceptions
    print('Error in getHealthAppBarTittle: $e');
    return "";
  }
}
```

Figure 50.- *getVideoPlayerHealthAppBarTittle method in health_data.dart file*

In *Figure 50*, the method *getVideoPlayerHealthAppBarTittle()* is illustrated. This method accepts a string as input and is primarily tasked with retrieving the corresponding submodule title for each video title provided as a parameter. When the targeted sub-module title is found, it is translated and returned.

3.2.4. User Testing (Round 1)

Participants in the first test included individuals from diverse nationalities, ages (25–45), genders, education levels, duration of residency in Norway, and information and communication technology (ICT) skills—with this batch of users rating between 5 and 9 out of 10 (see Table 6). These diverse backgrounds contributed to a varied UX during testing.

The majority of participants demonstrated proficiency in navigating the system, executing tasks such as language selection, video playback, and chapter navigation with ease. Despite this, specific tasks (for example, tasks 2, 4.2 and 5) proved challenging for

half of the participants, indicating that some areas needed improvement to provide a more seamless experience.

Table 6.- Demographics of participants (Round 1)

	Age	Gender	Education	Norwegian residence period	ICT skills (10 is max)	Mother tongue
Participant 1	30	Female	High school	2	7	Arabian
Participant 2	25	Male	Secondary school	2	5	Arabian
Participant 3	41	Female	Master	2	9	Ukrainian
Participant 4	45	Female	Bachelor	6	5	Turkish

For example, two of the participants ignored the *'*next video*'* button located underneath each video and would instead take a less efficient path through the top-left arrow. This suggests that there is a real need for multiple navigation options, as it could significantly reduce user frustration and improve interaction flow. However, it might also suggest that the button is not visible enough or does not convey its purpose with sufficient clarity.

Based on the SUS questionnaire that we provided, the system was generally found to be straightforward and well-integrated. Nevertheless, despite the success of the participants while navigating the tasks and their positive feedback, most participants exhibited a reluctance to use the system frequently, and one tester expressed that the system was somewhat cumbersome to use.

Table 7.- SUS Questionnaire Results (Round 1)

A statement / Opinion *(Scale*: 1 - strongly disagree; 2 - disagree; 3 - neutral; 4 - agree; 5 - strongly agree)	*Participant 1*	*Participant 2*	*Participant 3*	*Participant 4*
1. I think that I would like to use this system frequently.	4	3	2	2
2. I found the system unnecessarily complex.	2	4	2	2

3. I thought the system was easy to use.	5	5	5	4
4. I think that I would need the support of a technical person to be able to use this system.	1	3	1	1
5. I found the various functions in this system were well integrated.	5	4	4	4
6. I thought there was too much inconsistency in this system.	2	4	2	2
7. I would imagine that most people would learn to use this system very quickly.	5	3	4	5
8. I found the system very cumbersome to use.	1	4	2	2
9. I felt very confident using the system.	5	3	4	4
10. I needed to learn a lot of things before I could get going with this system.	1	3	2	2

Some of the insights that the participants provided us with were:

1. Accessibility issues were underscored by the Ukrainian participant, regarding mobile operator compatibility, and highlighting a pressing need to facilitate access to the application (*e.g.,* the application not showing up in the Play Store for non-Norwegian mobile operators).
2. Translation accuracy in the Ukrainian chapter titles was criticized, suggesting some room for improvement in the linguistic precision of the app for some specific languages.
3. A few enhancements were suggested for the settings menu, such as changing the way that the language change option is integrated.
4. Changes to the design of the progress bar on the home page were also proposed for better visibility and understanding.

These insights are essential for refining the app's design and functionality, improving UX, and increasing user satisfaction.

3.3 Second Iteration

Expanding upon the changes detailed in the previous section, we delved further into the enhancements made both in the UI and the back-end functionalities of the application. Front-end improvements include reworking the loading icon's appearance to enhance its visual appeal, transitioning from the default *CircularProgressIndicator* to a customizable progress indicator, which can be provided text within. Moreover, addressing feedback on the progress bar's clarity, a text feature was integrated to state clearer the user's current progress. Further clarifying navigation by refining the settings page in the Bottom Nav Bar, relabeling the icon as 'Language' and including a corresponding icon.

Additionally, to broaden accessibility, four additional languages: Amharic, Persian, Tamil, and Thai were integrated into the application. Text optimization efforts included refining various text elements for uniformity in length without compromising informative value, along with adjustments to video titles to align with existing content standards. Numerical representation was enhanced by implementing strategies to ensure accurate depiction of numbers across diverse numerical writing systems, in this case within Arabic, Pashto, and Urdu.

Back-end development focused on representing progress bars as numerical values, enabling a more precise and informative display for users, while addressing challenges related to non-Latin numeral interpretation through refined logic to accommodate languages outside the Latin language group.

3.3.1 Front-end

Based on user testing feedback, we decided to enhance the visual appeal of the content loading icon. Transitioning from the default *CircularProgressIndicator* to a customizable progress indicator was our group's consensus. Leveraging the circular percent indicator library, we now had the capability to incorporate text within the animated spinner. The text provided in the middle in addition to enlarging the spinner itself could prove to be a more satisfying visual feature than the previous spinner.

Feedback from both the first and second versions indicated a deficiency in the progress bar's ability to effectively communicate its purpose (see *Figure 51*). Consequently, we aimed to improve the progress bar by integrating a text feature in the middle. This text will display the user's current percentage progress toward watched videos within the

selected sector. In other words, showing the same value as the progress bar itself, but in another visual way.

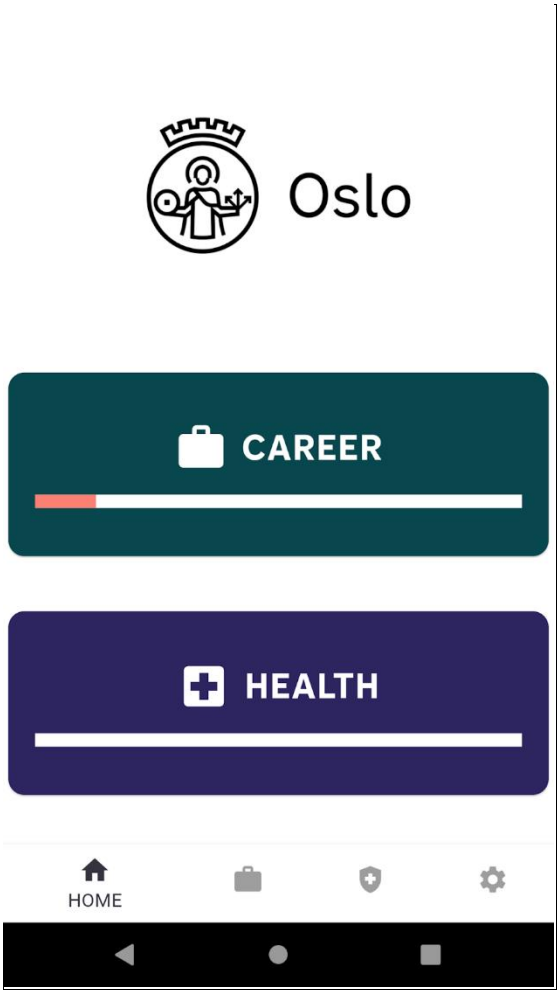


Figure 51.- *The old progress bar without text in the center.*

Some users also expressed confusion regarding the presence of a settings page in the Bottom Nav Bar, which, upon entering, only offered a language change option (see *Figure 52*). While we initially designed the settings page with future development in mind, such as implementing dark mode/light mode toggles, we recognized the need for clarity in its current state. Consequently, we decided to relabel the Bottom Nav Bar icon as ‘Language’ and include a more fitting icon to align with its singular function of language selection.

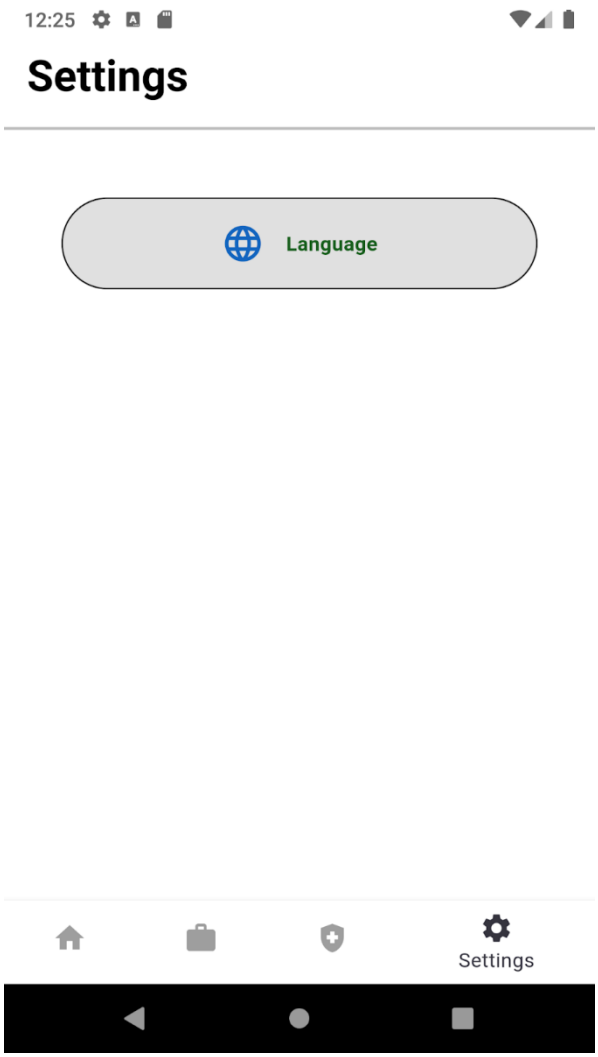


Figure 52.- *The old Bottom navigation containing 'Settings'*

In response to user feedback regarding the placement of the 'Next Video' button on the *VideoPlayerPage*, we acknowledged that its position at the bottom of the screen felt somewhat off. Also, it was not obvious to the user that the button would take the user to the next video (see *Figure 53*). To address this, we opted to relocate the button and a fitting heading to the center of the space between the video title and the bottom of the page (see *Figure 53*). This adjustment aimed to present a more balanced and satisfying view for the user.

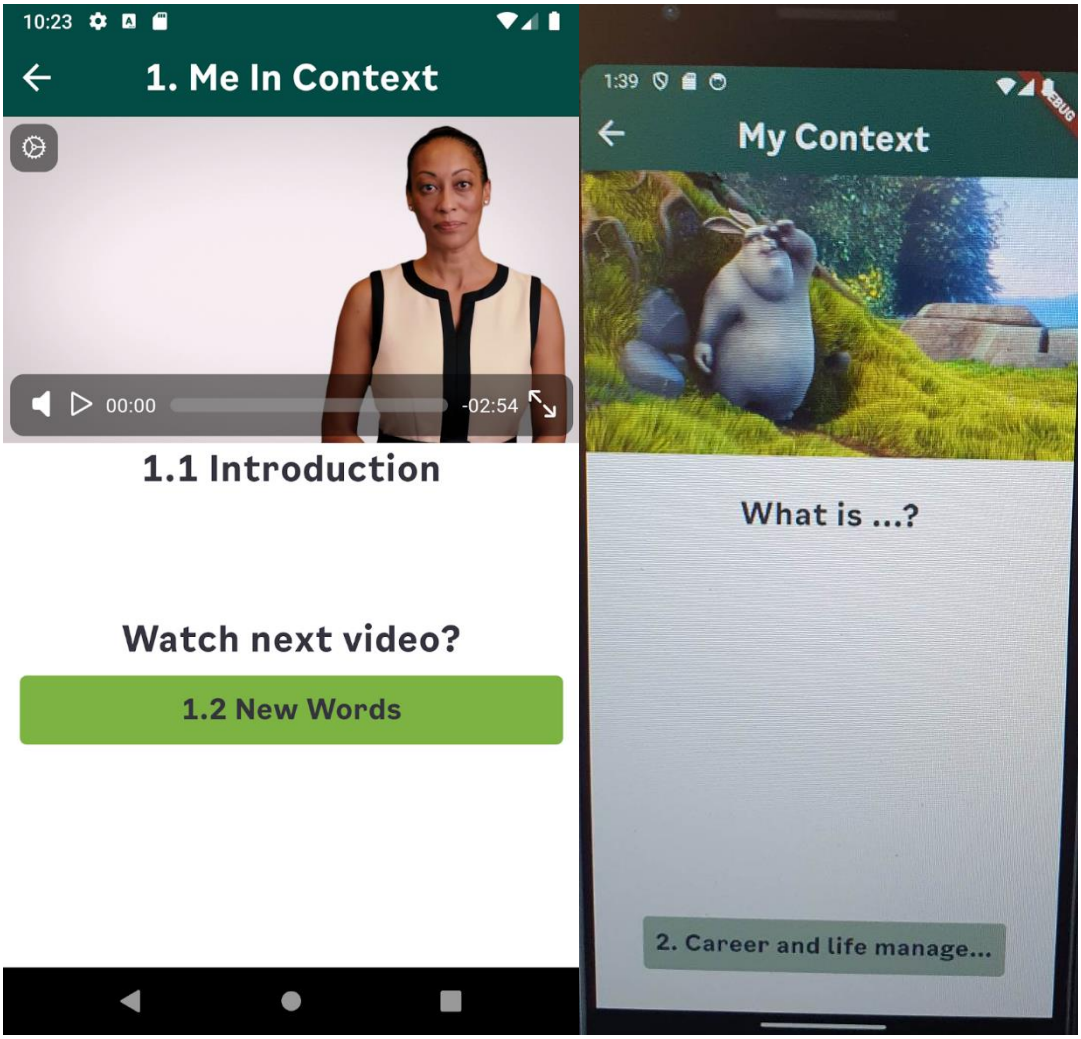


Figure 53.- *New button placement on the left, old button placement on the right*

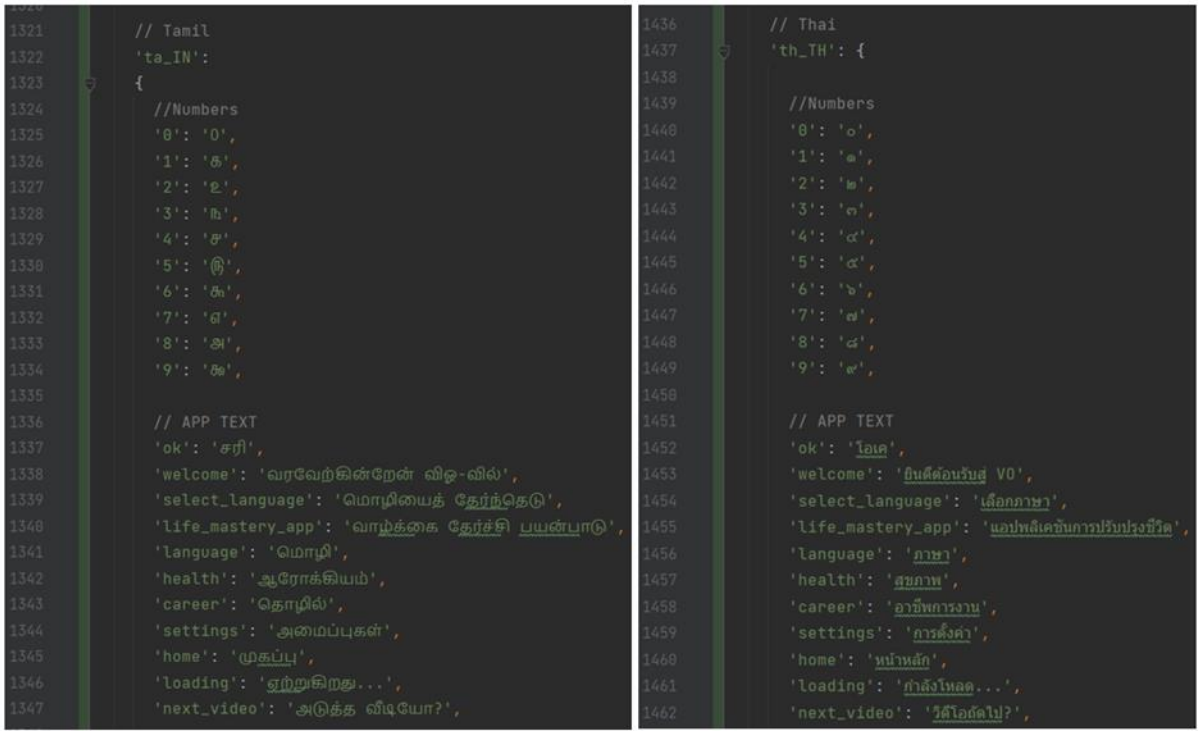
3.3.1.1 Additional languages

Four additional languages have been integrated into the application: Amharic, Persian, Tamil, and Thai. These languages were specifically requested by the client. The process of incorporating them involved translating the titles and text displayed in the app into each respective language. This translation task was facilitated using both ChatGPT 3.5 & Google Translate.

The translated text for each language is handled as values within the *getKeys* getter in the *LocaleString.dart* file. These values are assigned to keys that correspond to those previously designated for the text in each language within the application. Both the keys and values are stored within an inner map in the *getKeys* getter within the *LocaleString.dart* file.

This inner map is then assigned as a value to a key consisting of a four-letter string separated by underscores ('_'), representing the Locale for each of the five languages. The implementation of Tamil and Thai in the *LocaleString.dart* file is depicted in *Figure 54*. It's

important to note that this process mirrors that of the other languages included in the application.

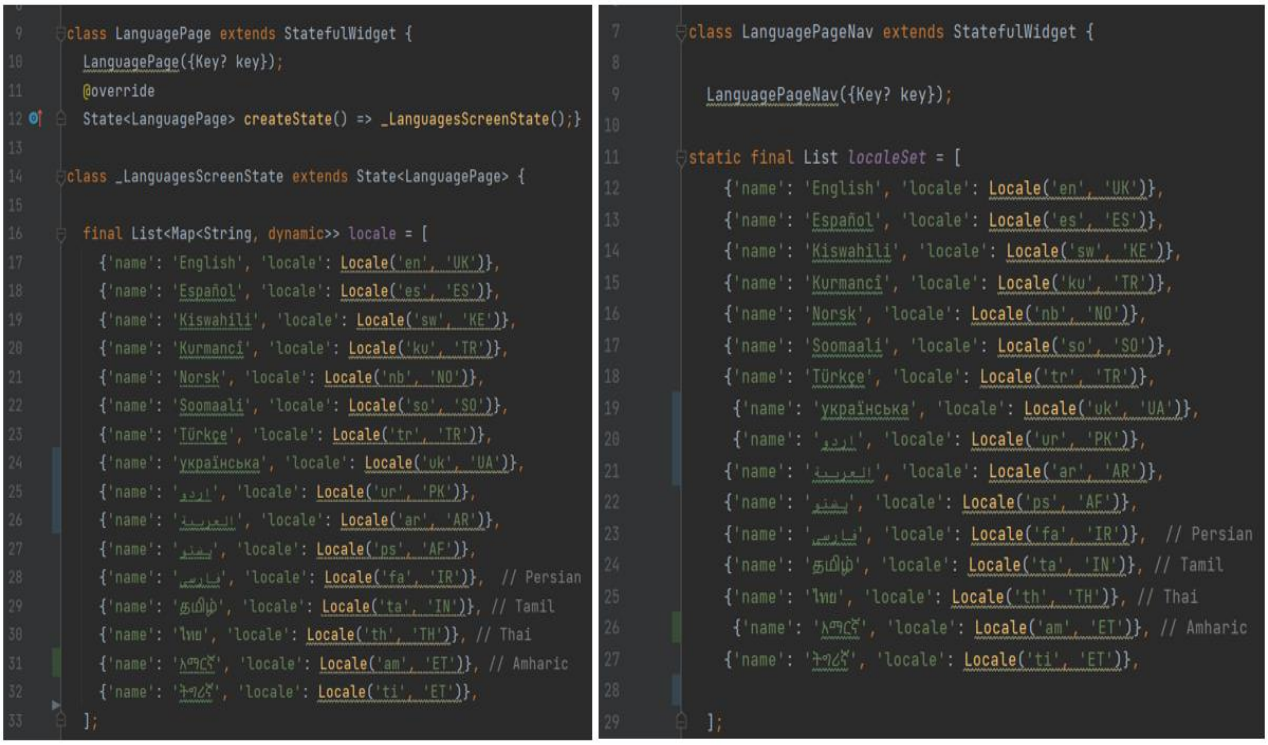


```
// Tamil
'ta_IN':
{
  //Numbers
  '0': '0',
  '1': '௧',
  '2': '௨',
  '3': '௩',
  '4': '௪',
  '5': '௫',
  '6': '௬',
  '7': '௭',
  '8': '௮',
  '9': '௯',

  // APP TEXT
  'ok': 'சரி',
  'welcome': 'வரவேற்கின்றேன் விஓ-வில்',
  'select_language': 'மொழியைத் தேர்ந்தெடு',
  'life_mastery_app': 'வாழ்க்கை தேர்ச்சி பயன்பாடு',
  'language': 'மொழி',
  'health': 'ஆரோக்கியம்',
  'career': 'தொழில்',
  'settings': 'அமைப்புகள்',
  'home': 'முகப்பு',
  'loading': 'ஏற்றுகிறது...',
  'next_video': 'அடுத்த வீடியோ?',
```

```
// Thai
'th_TH': {

  //Numbers
  '0': '๐',
  '1': '๑',
  '2': '๒',
  '3': '๓',
  '4': '๔',
  '5': '๕',
  '6': '๖',
  '7': '๗',
  '8': '๘',
  '9': '๙',

  // APP TEXT
  'ok': 'โอเค',
  'welcome': 'ยินดีต้อนรับสู่ VO',
  'select_language': 'เลือกภาษา',
  'life_mastery_app': 'แอปพลิเคชันการปรับปรุงชีวิต',
  'language': 'ภาษา',
  'health': 'สุขภาพ',
  'career': 'อาชีพการงาน',
  'settings': 'การตั้งค่า',
  'home': 'หน้าหลัก',
  'loading': 'กำลังโหลด...',
  'next_video': 'วิดีโอถัดไป?',
```

Figure 54.- *Tamil & Thai implementation in LocaleString.dart file*

As previously discussed, a Locale consists of a four-letter string, with the first two letters denoting the specific language and the last two letters indicating the country where the language is spoken or utilized. These Locale codes for each language will be appended to the lists found in the *language_page.dart* and *language_page_nav.dart* files. The lists are organized alphabetically, starting with languages that employ scripts similar to the Latin script, either entirely or partially. Subsequently, languages with non-Latin scripts are also sorted alphabetically. This arrangement is illustrated in *Figure 55*.



```dart
class LanguagePage extends StatefulWidget {
  LanguagePage({Key? key});
  @override
  State<LanguagePage> createState() => _LanguagesScreenState();}

class _LanguagesScreenState extends State<LanguagePage> {

  final List<Map<String, dynamic>> locale = [
    {'name': 'English', 'locale': Locale('en', 'UK')},
    {'name': 'Español', 'locale': Locale('es', 'ES')},
    {'name': 'Kiswahili', 'locale': Locale('sw', 'KE')},
    {'name': 'Kurmancî', 'locale': Locale('ku', 'TR')},
    {'name': 'Norsk', 'locale': Locale('nb', 'NO')},
    {'name': 'Soomaali', 'locale': Locale('so', 'SO')},
    {'name': 'Türkçe', 'locale': Locale('tr', 'TR')},
    {'name': 'українська', 'locale': Locale('uk', 'UA')},
    {'name': 'اردو', 'locale': Locale('ur', 'PK')},
    {'name': 'العربية', 'locale': Locale('ar', 'AR')},
    {'name': 'پښتو', 'locale': Locale('ps', 'AF')},
    {'name': 'فارسی', 'locale': Locale('fa', 'IR')},  // Persian
    {'name': 'தமிழ்', 'locale': Locale('ta', 'IN')}, // Tamil
    {'name': 'ไทย', 'locale': Locale('th', 'TH')}, // Thai
    {'name': 'አማርኛ', 'locale': Locale('am', 'ET')}, // Amharic
    {'name': 'ትግርኛ', 'locale': Locale('ti', 'ET')},
  ];
```

```dart
class LanguagePageNav extends StatefulWidget {

  LanguagePageNav({Key? key});

static final List localeSet = [
    {'name': 'English', 'locale': Locale('en', 'UK')},
    {'name': 'Español', 'locale': Locale('es', 'ES')},
    {'name': 'Kiswahili', 'locale': Locale('sw', 'KE')},
    {'name': 'Kurmancî', 'locale': Locale('ku', 'TR')},
    {'name': 'Norsk', 'locale': Locale('nb', 'NO')},
    {'name': 'Soomaali', 'locale': Locale('so', 'SO')},
    {'name': 'Türkçe', 'locale': Locale('tr', 'TR')},
     {'name': 'українська', 'locale': Locale('uk', 'UA')},
     {'name': 'اردو', 'locale': Locale('ur', 'PK')},
    {'name': 'العربية', 'locale': Locale('ar', 'AR')},
    {'name': 'پښتو', 'locale': Locale('ps', 'AF')},
    {'name': 'فارسی', 'locale': Locale('fa', 'IR')},  // Persian
    {'name': 'தமிழ்', 'locale': Locale('ta', 'IN')}, // Tamil
    {'name': 'ไทย', 'locale': Locale('th', 'TH')}, // Thai
     {'name': 'አማርኛ', 'locale': Locale('am', 'ET')}, // Amharic
    {'name': 'ትግርኛ', 'locale': Locale('ti', 'ET')},

  ];
```

Figure 55.- *Language lists in language_page.dart & language_page_nav.dart files.*

3.3.1.2 Text Phrasing

All text presented within the app is centralized in the *LocaleString.dart* file, which primarily governs the display of text according to the selected language (Locale). Within this file, the *getKeys* getter comprises four main categories: text pertaining to career pages, text related to health pages, numerical values, and miscellaneous text throughout the app. The first two categories, concerning content on career and health pages, were predetermined by the client. However, extensive text adjustments were made to enhance the UX and optimize text presentation.

Indeed, various languages have different word lengths, resulting in titles of varying lengths across languages. Additionally, the number of words required for titles can differ between languages. In the client's initial list of titles, many were excessively long, providing detailed information about the content, as illustrated in *Figure 56*.

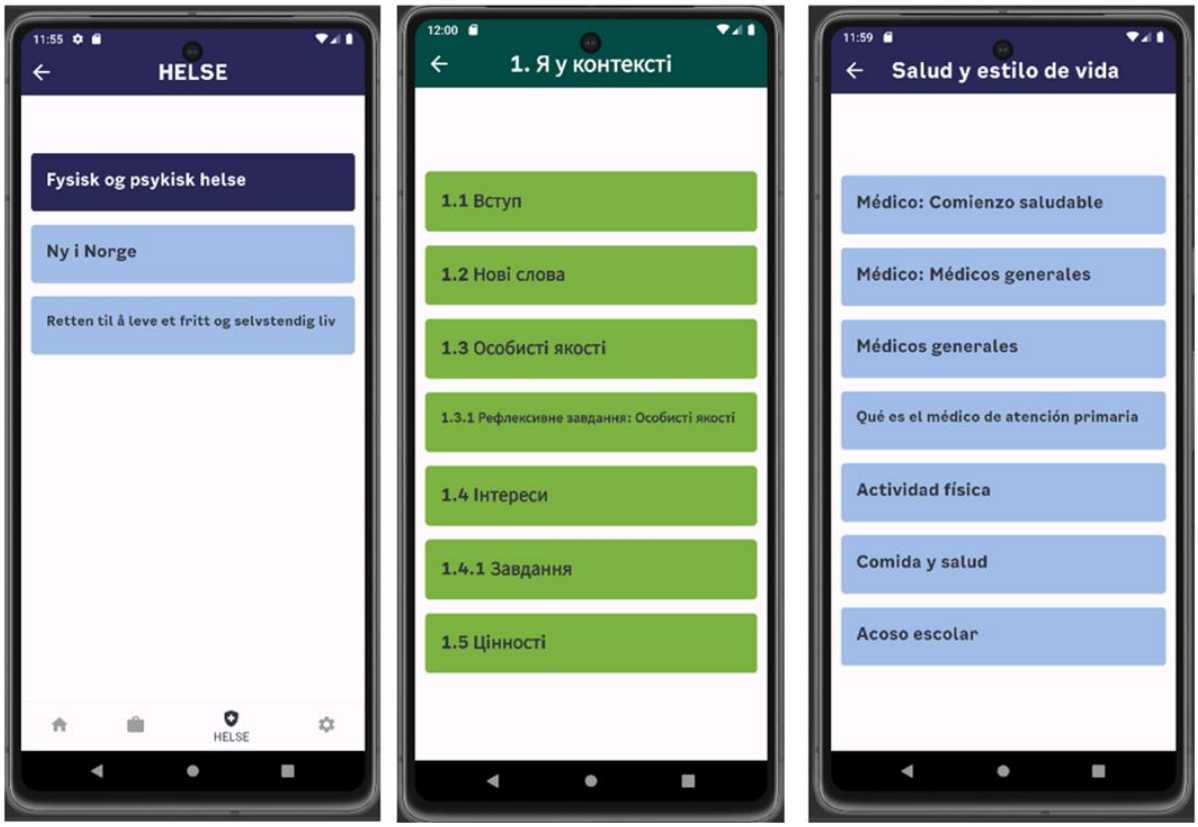


Figure 56.- *Text on career and health pages before re-phrasing.*

To promote consistency across career and health pages, various titles underwent revision to ensure uniformity in their length. However, these adjustments were made without sacrificing the informative value of the titles for users. *Figure 57* provides examples of titles that were refined to rectify inconsistencies in text length. In addition, video titles for languages such as Turkish, Ukrainian, and Arabic which were already decided by the client were revised and made consistent with the titles found on Synthesia, the website in which the client uploaded all videos used on the app.

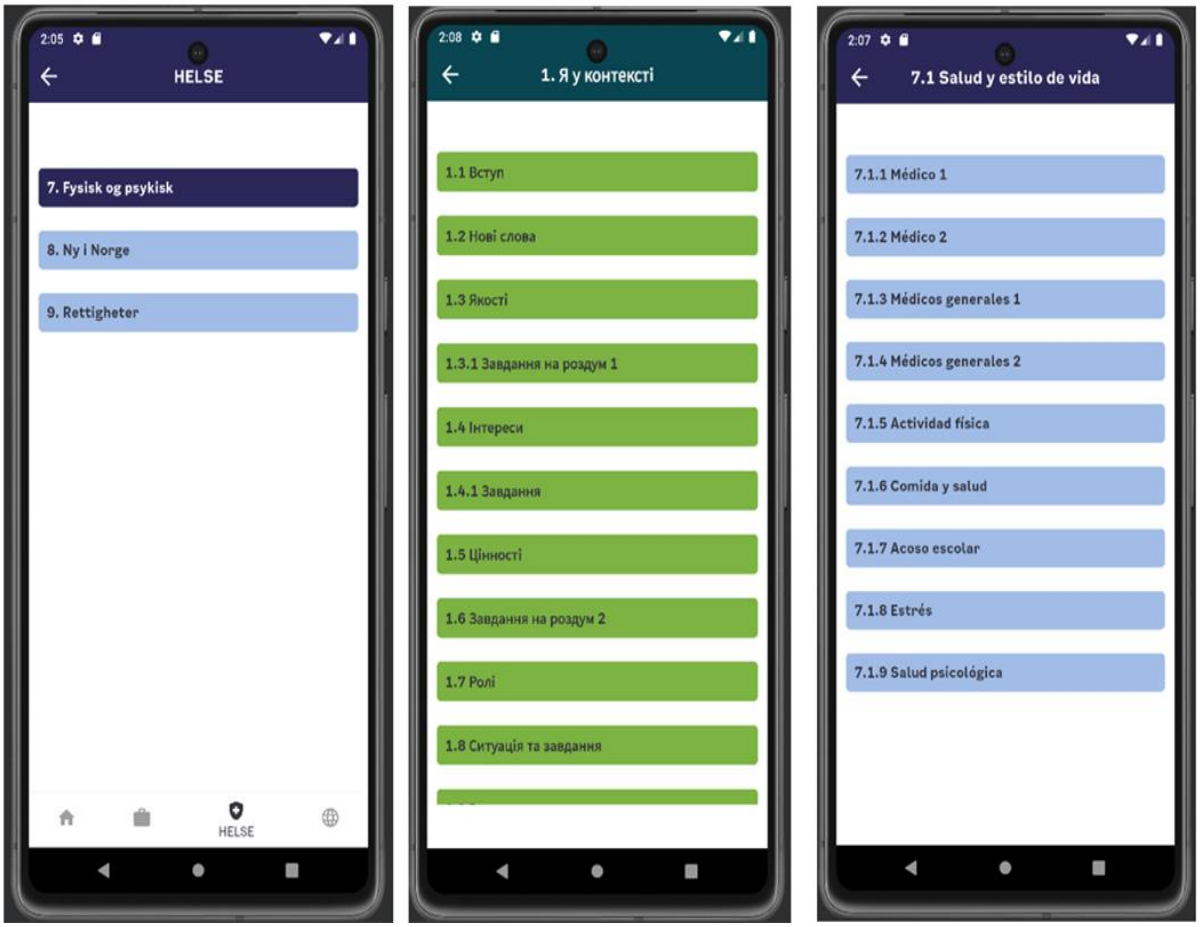


Figure 57.- *Text on career and health pages after re-phrasing*

Additionally, within the *getKeys* getter in the *LocaleString.dart* file mentioned earlier, there exists a category of strings denoting numerical values. These values range from single-digit numbers, spanning from zero ‘0’ to nine ‘9’ (0-9). Notably, Arabic, Pashto, Persian, Tamil, Thai, and Urdu each possess their own distinct numerical writing systems and characters. To ensure an accurate representation of numbers when any of these languages are selected, the translation value corresponding to Latin numerical digits was assigned to keys containing Latin digits. This process for the numerical values for Arabic, Tamil, Thai, and Persian is depicted in *Figure 58*. *Figure 59* demonstrates the utilization of Arabic numerical values in the progress indicator on the homepage.

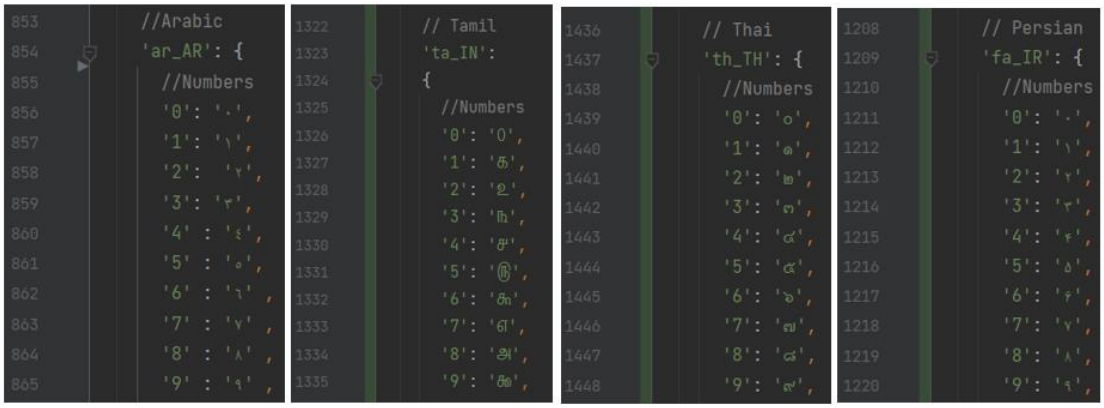


Figure 58.- *Arabic, Tamil, Thai & Persian numerical values.*

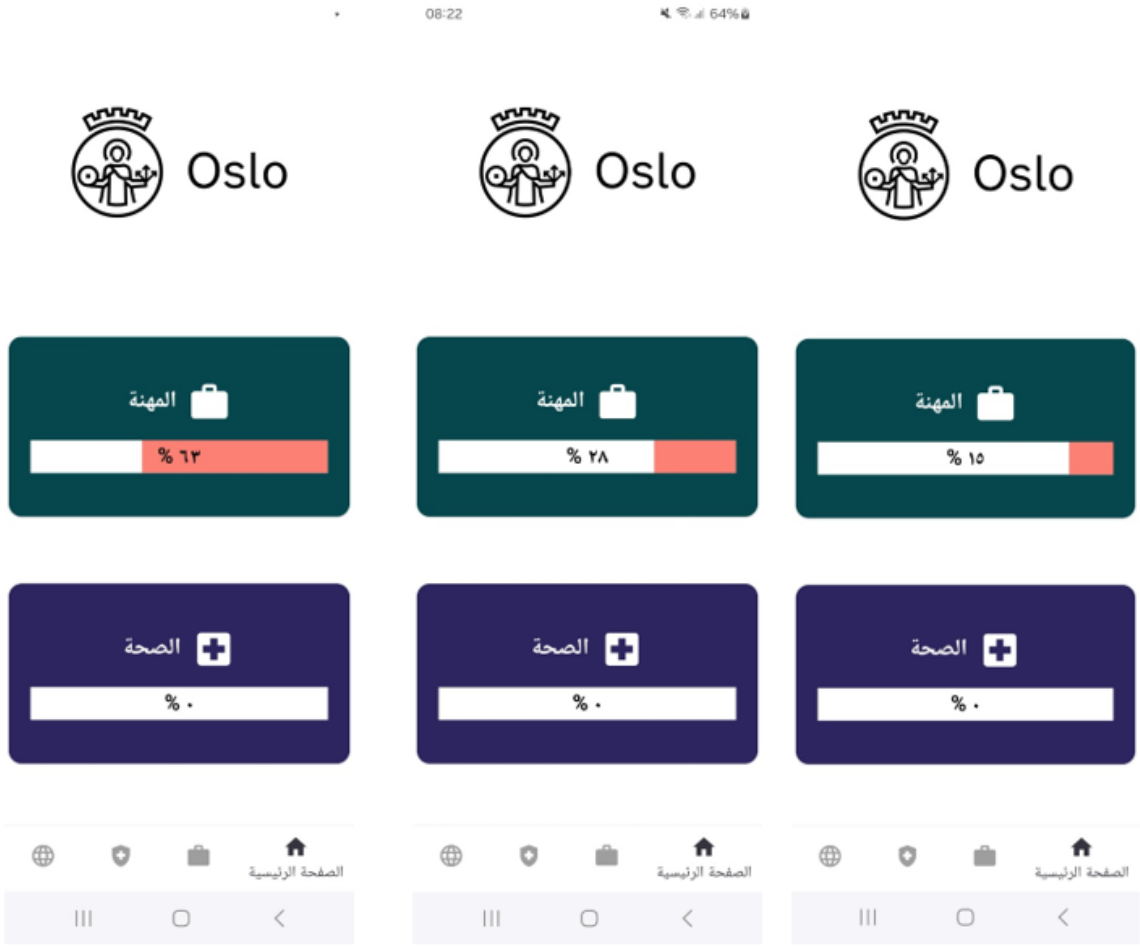


Figure 59.- *Progress indicator in Arabic language.*

3.3.2 Back-end

In order to develop the feature of representing the progress bar as numbers, and not just a scaling color bar, we had to take advantage of the previously developed methods *getCareerProgress()* and *getHealthProgress()*. Since both of these models returned a double that was the sum of seen videos, then we could turn it into a percentage value.

3.3.2.1 Non-Latin Numerals

When developing the logic for fetching the video URL strings from the database. We tried to take advantage of passing the retrieved topics for the current page to know which path to provide for the query. For example, if the user clicked into '*Chapter 1. Me in context*' inside Career, then I would like to retrieve this string and fetch the first number to know how to set the path for the database, where it would retrieve all the topics inside this chapter.

The problem was that all the strings could be translated into languages that were not part of the Latin language group. These language numerals could not be interpreted by functions trying to parse numeric values from a string. It took some time implementing methods that could take care of these exceptions and make it part of the ordinary logic. These methods sorted on headings that were not translated at the given time, which were part of the Latin language group.

3.3.3 User Testing (Round 2)

This test targeted individuals with a more limited proficiency in Norwegian or English, specifically targeting Ukrainian refugees—some of whom were proficient in ICT and held master's degrees (see Table 8). Participants included technology professionals and educators, which enriched the quality of the feedback with more specialized insights.

Table 8.- Demographics of participants (Round 2)

	Age	Gender	Education	Norwegian residence period	ICT skills (10 is max)	Mother tongue
Participant 1	23	Male	Master	1	8	Ukrainian
Participant 2	22	Male	Master	1	7	Ukrainian
Participant 3	29	Male	Master	3	9	Ukrainian
Participant 4	26	Female	Master	2	6	Ukrainian
Participant 5	42	Female	Master	4	9	Ukrainian

The participants adeptly completed the tasks, appraising the app's design elements, such as the color palette and font size. Technical issues, such as delays in loading the translations and poor responsiveness of the back arrow or the language buttons, were noted.

Table 9.- SUS Questionnaire Results (Round 2)

A statement / Opinion *(Scale*: 1 - strongly disagree; 2 - disagree; 3 - neutral; 4 - agree; 5 - strongly agree)	*Participant 1*	*Participant 2*	*Participant 3*	*Participant 4*	*Participant 5*
1. I think that I would like to use this system frequently.	4	5	4	5	4
2. I found the system unnecessarily complex.	1	2	2	1	2
3. I thought the system was easy to use.	5	4	4	5	4
4. I think that I would need the support of a technical person to be able to use this system.	1	1	1	1	2
5. I found the various functions in this system were well integrated.	5	5	3	5	4
6. I thought there was too much inconsistency in this system.	1	2	2	1	2
7. I would imagine that most people would learn to use this system very quickly.	5	5	4	5	4
8. I found the system very cumbersome to use.	1	2	2	1	2
9. I felt very confident using the system.	5	4	4	4	4
10. I needed to learn a lot of things before I could get going with this system.	1	1	1	1	2

This time around, the participants also proposed several improvements:

1. introducing more dynamic graphics or text during the loading screen transitions.
2. adding a *'*Continue*'* button on the home page for easier navigation and improving upon the design of the progress bar.
3. enhancing the video player settings, such as easier language switching and layout adjustments, will improve usability.
4. increasing the font size of video titles and adding captions or a transcript under the videos.
5. having a more consistent design throughout the application, including font, color, icons, and so on.
6. improving the contrast between some background and font colors.

Due to time constraints, not all feedback from this round could be immediately addressed. However, the insights provided are invaluable for future updates, making sure that the app will remain responsive to the users' needs and preferences above everything else. The full conclusion for the tests conducted can be located in *Appendix C: User Testing Evaluations*.

3.4 Final Applied Changes

Based on the findings from the second round of user testing, enhancements were implemented to refine the text within the app. Specifically, feedback highlighted inconsistencies in capitalization, particularly in the titles of the career and health pages. To ensure uniformity, it was recommended to maintain consistent capitalization across all text elements, including those in the bottom navigation bar.

To address this issue, additional strings (keys and corresponding values) have been integrated into the *LocaleString.dart* file, which governs all translations within the app. Three new pairs of strings (keys and values) were added for each of the 16 languages supported by the app, utilizing similar keys: 'language_up', 'career_nav', and 'health_nav'. This adjustment ensures coherence across the UI. *Figure 60* demonstrates the revised presentation of text in Norwegian on the career, health, and language pages, reflecting the implemented changes. The modification is demonstrated in *Figure 61*, where one can see in the nav-bar that the strings are consistent in terms of font with or without capitalization.

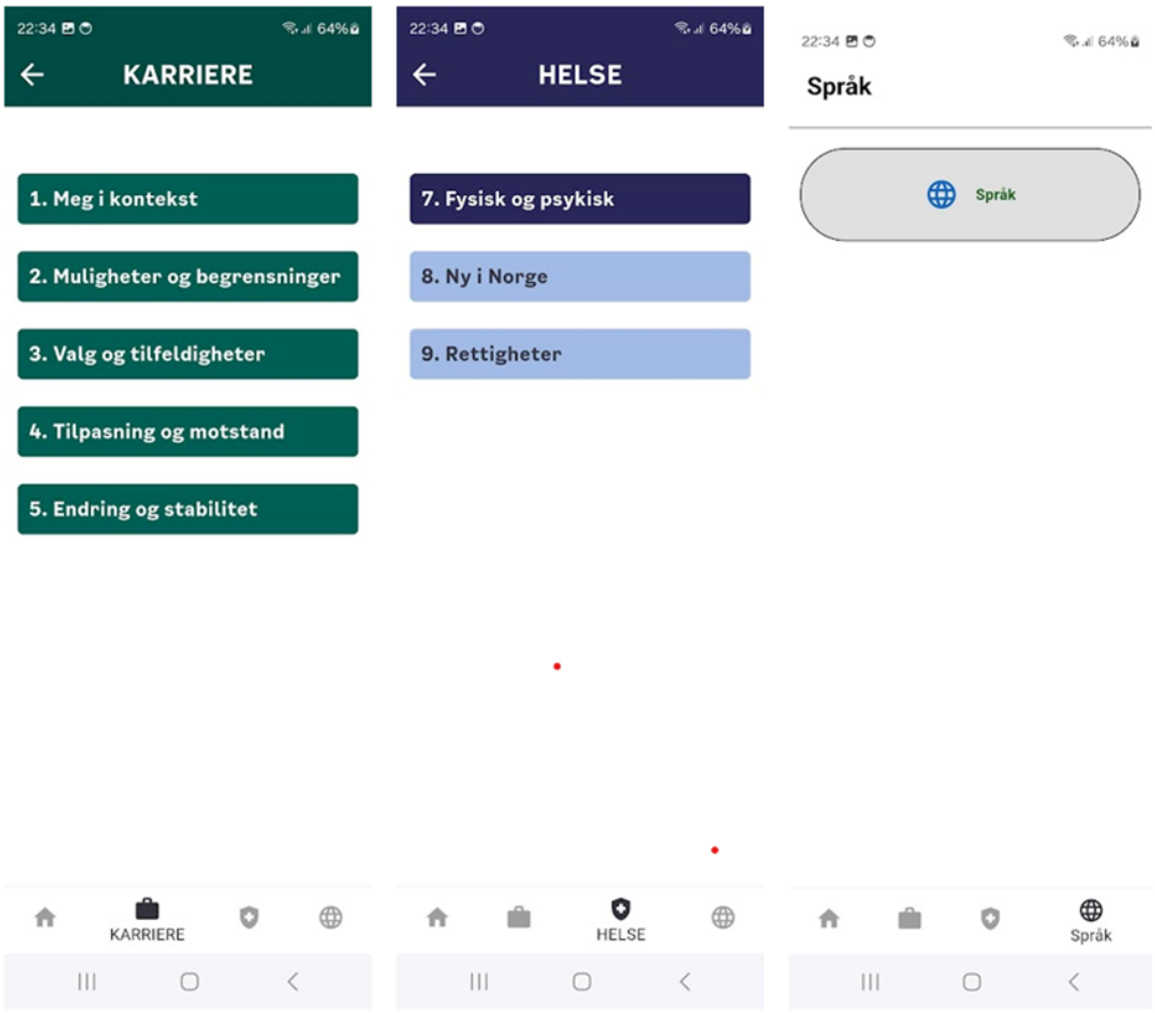


Figure 60.- *Text on career, health, and language pages before modifications displayed in Norwegian*

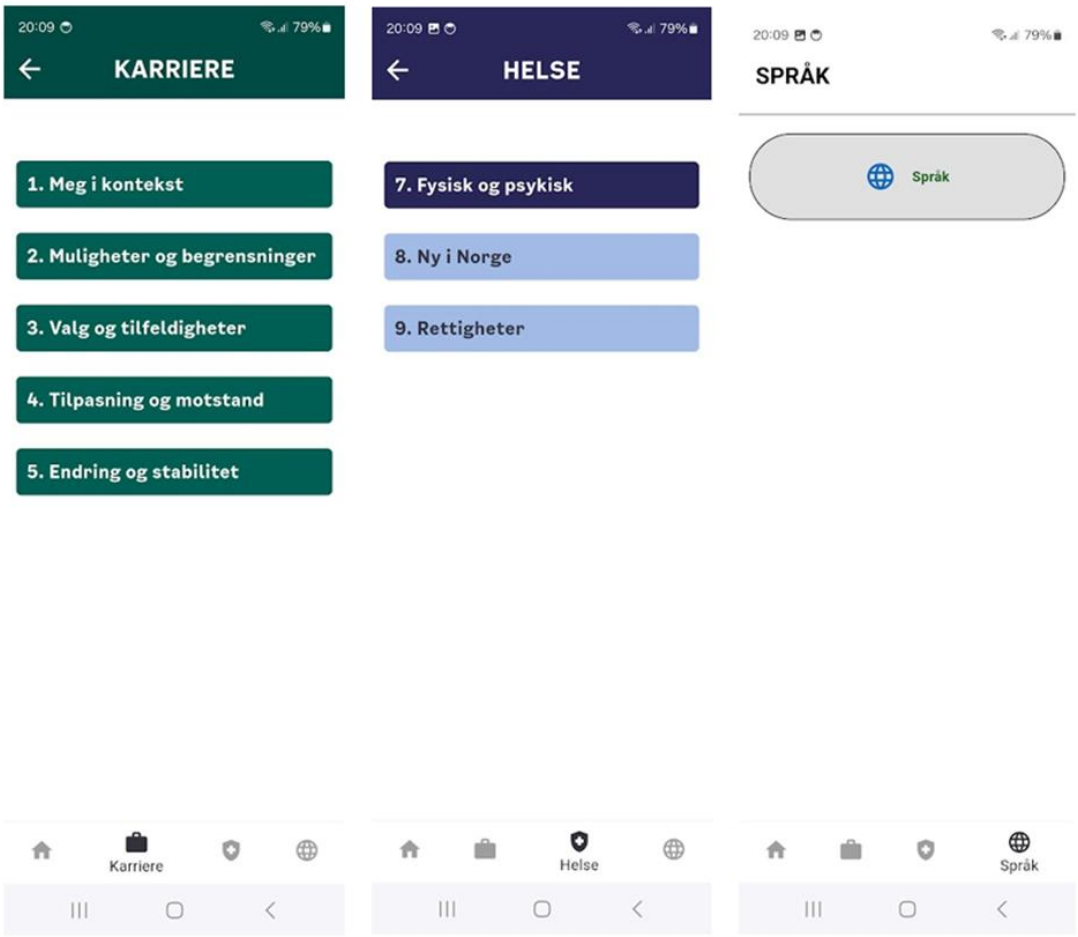


Figure 61.- *Text on career, health, and language pages after modifications displayed in Norwegian*

4. Product Documentation

The earlier sections of the report covered the development of the product and showcased how the product, or parts of the product, looked in the different stages of the development. This walkthrough will display how the final product looks from startup but will only cover a slim part of the functionality, as has been explained in earlier stages of the report.

At startup the application will render the splash screen (see left illustration in *Figure 62),* the text on this screen will by default be in Norwegian, but after a language has been set, the text on the screen will be translated thereafter. The first time the application is started the splash screen will lead to the language page (see middle illustration in *Figure 62*). On this page the user can choose a language. When a button is clicked the language is set and the user will be navigated to the home page (see right illustration in *Figure 62*). This page gives the user the possibility to navigate to Career or Health content by clicking on the cards in the middle of the screen, or by clicking on the tabs in the nav-bar. It also provides the possibility to navigate to the language page nav by clicking on the tab for language in the nav-bar.

Figure 62.- *The splash screen, language page and home page.*

If the user clicked on the suitcase or health icon on the nav-bar, then it would lead the user to a navigation similar to the middle illustration in *Figure 63*. The career or health page gives the user the possibility to choose chapter, and then choose a sub-chapter or topic. If the user where to click on the career card in the middle of the home screen (see right

illustration in *Figure 62*), then it would take the user to the right illustration in *Figure 63*. Had the user chosen the health card instead, then the navigation would have taken the user to the left picture in *Figure 64*. Each time the application accesses dynamic content, the loading indicator is displayed (see left illustration in *Figure 63*).

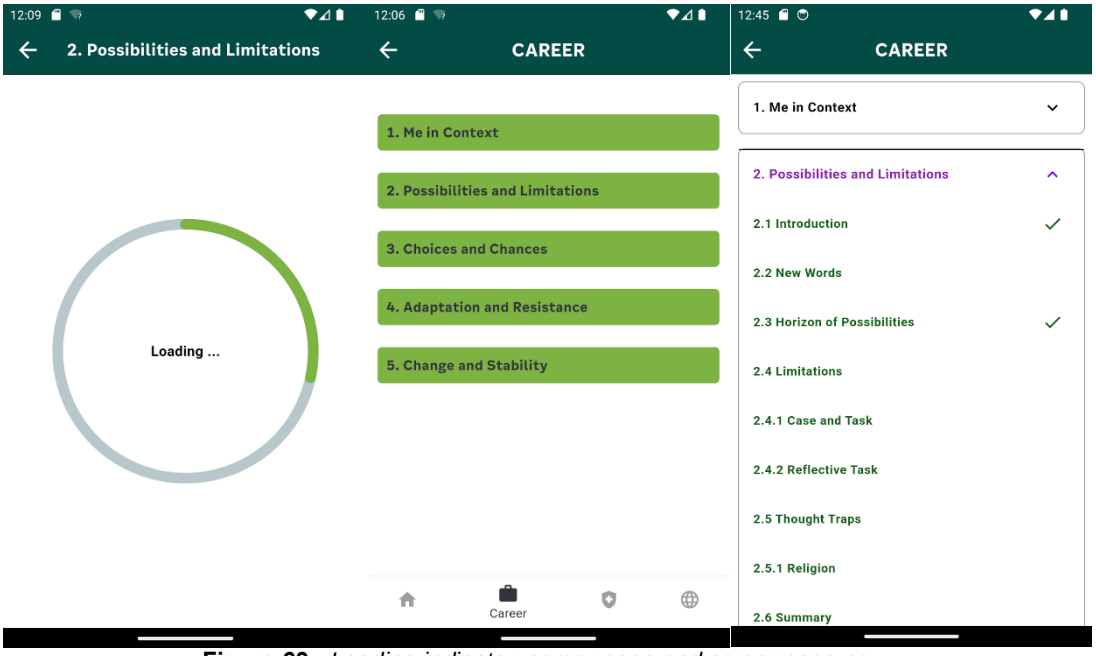


Figure 63.- *Loading indicator, career page and career page new.*

After the user either clicks on a topic in the navigation through the nav-bar, or through the overview via the home page cards, it leads to the video player page (see middle illustration in *Figure 64*). The user has the ability to watch the video of the topic. By doing this he can change the playback speed, volume and enter full screen (see right picture in *Figure 64).* If the video had not been available, then the user would have got the message instead from the left picture in *Figure 65*. After the user has seen a movie, and the state of the page is refreshed, then the user can see a green tick to the right of the movie in the overview of the chapter (see right picture in *Figure 63*). It also increases the percentage bar of the section at the home page (see right illustration in *Figure 62*).

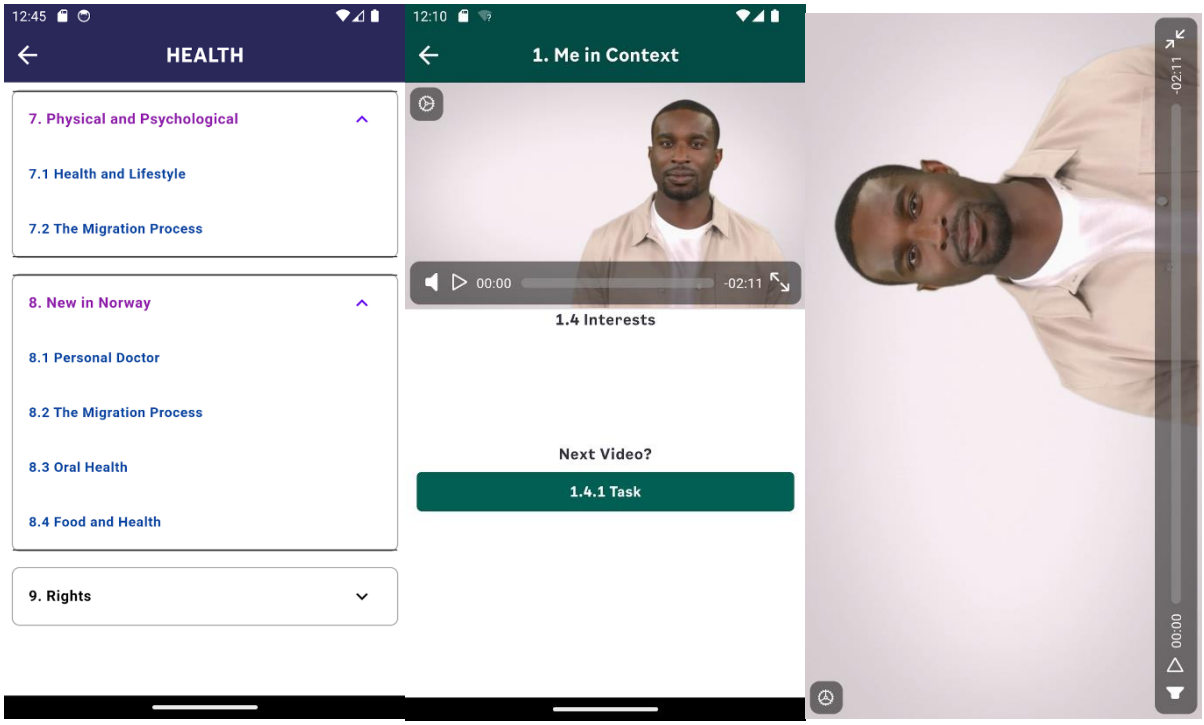


Figure 64.- *Health page new, video player page and video in full screen.*

In case that the user chose the wrong language, or just prefers to change it, then this can be done by navigating to the language page nav via the right tab ‘Language’ in the nav-bar (see middle picture in *Figure 65*). On this page the user can select the language button, this will display an alert dialog which offers a list of all the languages available in the application (see right illustration in *Figure 65*). The user then sets the preferred language by clicking on it. When this happens the application is refreshed, the newly set language is applied and the user is returned to the home screen.

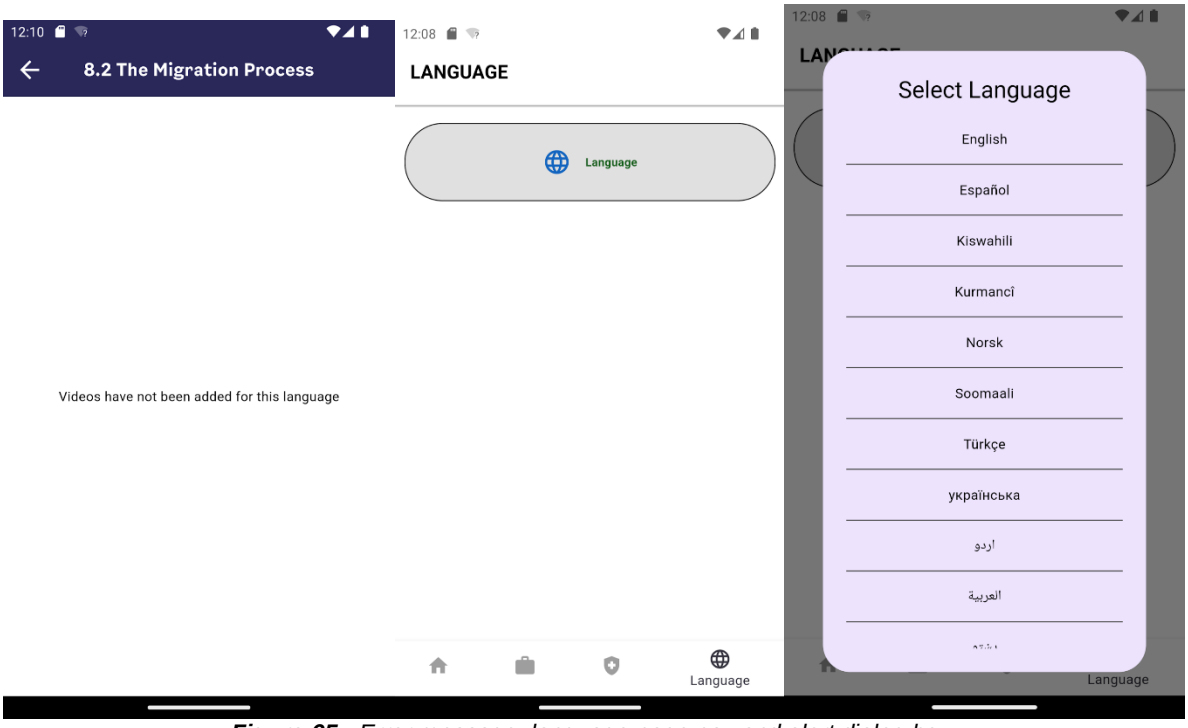


Figure 65.- *Error message, language page nav and alert dialog box.*

5. Product Maintenance

From time to time, there is a need for the P.O. to update the contents of the videos. Therefore, implementing a simple maintenance system that does not rely on anyone with a technical background was an essential step in creating a successful product for our client. The P.O. themselves can add or change videos to the application if the headlines have been provided up front. To do this, the P.O. must have access to the Firebase Storage for the Livsmestring app, which can be seen in *Figure 66*.

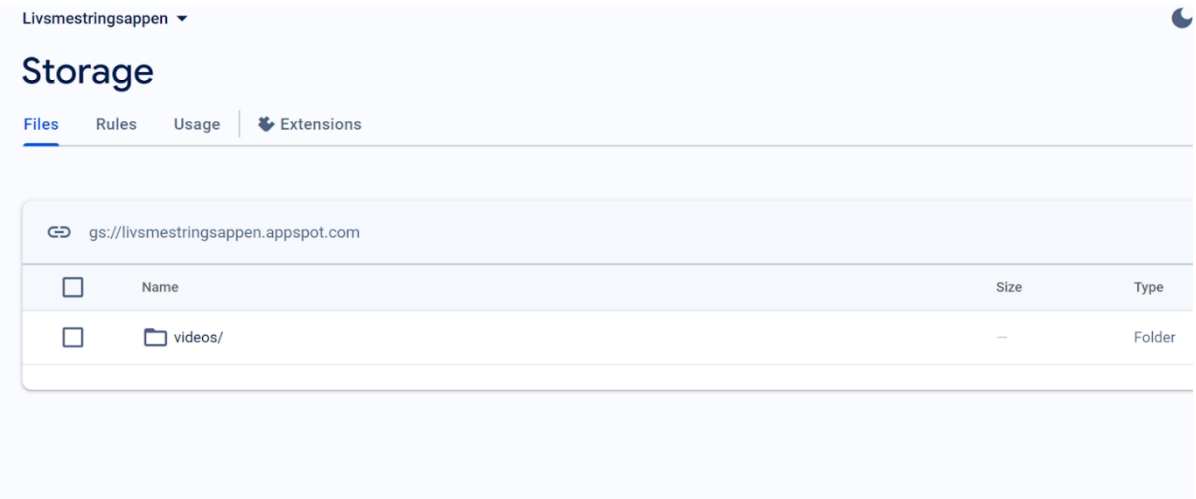


Figure 66.- *Videos folder at root of Firebase Storage*

By entering the storage, the user finds itself at root (videos). When the P.O. clicks on the folder, they can navigate through the different languages, sections, chapters, and sub-chapters currently made for the application. *Figure 67* displays that the P.O. has clicked on the Norwegian language, the health section, the chapter called 'Physical and Psychological', and at last the sub-chapter called health and lifestyle. This sub-chapter is currently empty but could easily become populated by clicking on the upload file button. The P.O. then simply provides the videos it wants to display for the current chapter, as long as they match the number of topic headlines in the application. *Figure 68* shows the importance of numerating the videos in the order one wants them displayed. If they are not numerated, then they will be matched with headlines at random.

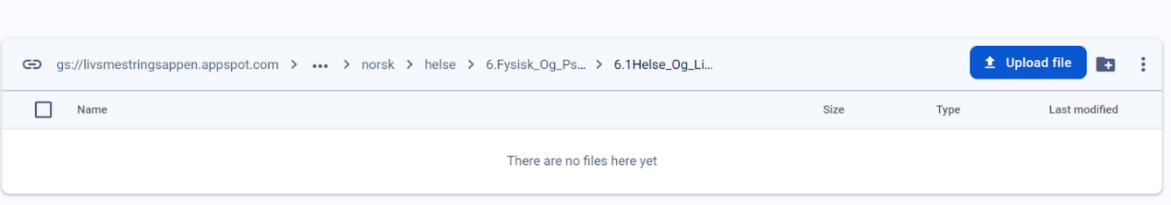


Figure 67.- *Inside the folder for topic health and lifestyle*

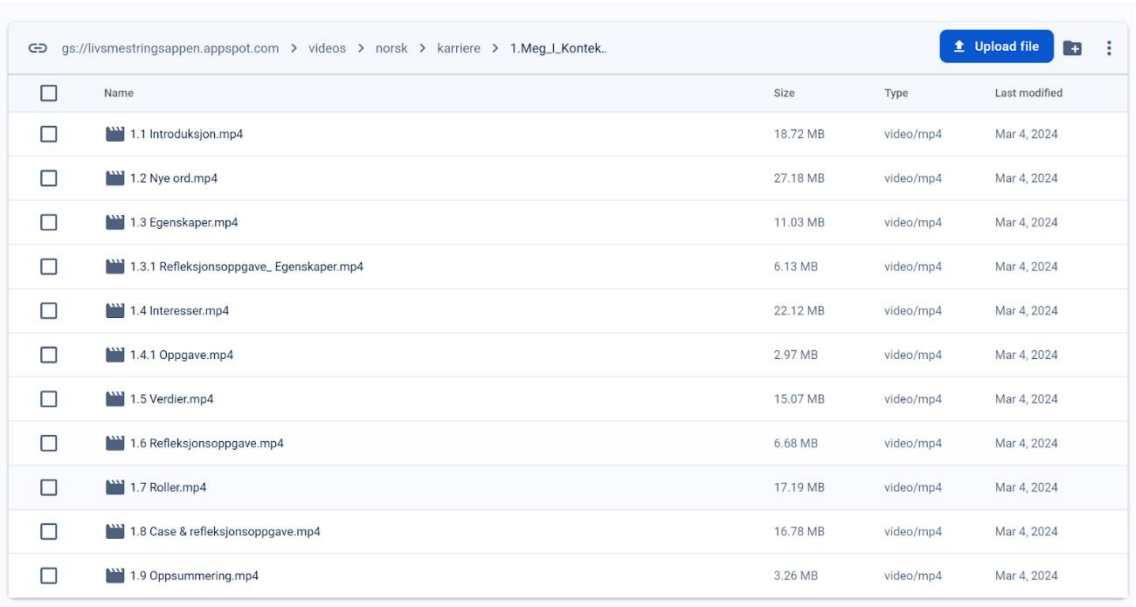


Figure 68.- *Inside the folder for topic me in context, showcasing how videos should be numerated.*

6. Reflections

In this section, our reflections showcase how things were achieved both product and process wise. It demonstrates how the product is somewhat acceptable in terms of achieving the functionalities, but also the performance standards set for the product. However, the product is not flawless and can be enhanced in multiple ways. Also, the process could have gone better, when considering our implementation of Scrum and communication channels.

6.1. Planning Vs Reality

At the beginning of the project, we wrote a pre-report in which we laid out the goals we had set for ourselves within this project, as well as the step-by-step plan we had created to accomplish those objectives (Halsne et al., 2024). Five months is a short time to create a flawless application from scratch, while simultaneously being a long enough period for plans to change, whether it be because of a new requirement or an obstacle along the way.

When comparing the actual course of events with our original planning as stated here, there were definitely some changes that took place. For instance, we had to cut down *on* our learning and prototyping phases drastically—originally, we wanted to take a week-long sprint to plan the project, then a two-week sprint to learn about Flutter, and then take another one to create two different Figma prototypes. Instead, we reduced this process from a total of five weeks down to about three weeks, as we were advised by our supervisor that we would not have enough time to spare if we didn't tighten up our schedule. Following this change, the plan was followed more or less as we predicted, with the exception of not having enough time to perform heuristic testing on the design, or unit testing on every single method of the application, which will be explained more in the next sub-chapters.

The plan to divide our team into a front-end developing pair and a back-end developing pair did somewhat work out as expected. The product was developed, but the communication and enhancement of Scrum in daily life were poor. Furthermore, the text-to-speech functional requirement had to be dropped due to it being a highly complex implementation that was not high on our priority list. On the other hand, we did find the time to take on some extra tasks, such as creating a custom launcher icon for the application, which required some extra time for learning how to use vector graphics software.

Finally, the deployment date of the application to the App Store and Play Store is set to be right after this report is turned in. This fairly matches the deployment date set in the preliminary report (Halsne et al., 2024).

.

6.1.1. Achieving the Specified Requirements

In *Chapter 2. Methodology,* we presented and referred to the outlined functional and non-functional requirements set for this project. By comparing the final product against these requirements, it is possible to evaluate the final product. The functional requirements consisted of the main and the desired requirements:

As shown above (*e.g., Figure 64),* the user has the possibility to navigate through both the Health and the Career sections and, at the culmination, watch videos about different topics. When the application starts for the first time, the user has to set the application language. It can also be changed afterwards under the tab 'Languages', as presented and discussed earlier in the report. Our solution does not support language settings based on the phone's internal language settings. The reason for this is that it is simply not that useful when the user itself can set and change it in the application. This also opens up the possibility of having different languages on the phone and in the application, which some users might favor.

The P.O. did not want us to implement the feature of creating user accounts. We adhered to that wish but still made it possible for the user to follow their own progress. Though this progress would be lost if the user chose to change phones or delete the data on their current phone. The progress would be measured by the already-explained progress bar on the home screen, as well as the green tics that appear if a user has seen a video.

We have enabled the P.O. to independently update content to a considerable extent. The ease of adding pre-made videos lies in simply uploading them to Firebase Storage. However, for videos requiring new text strings, each language variation must be manually added to the *LocaleString.dart* file. While these tasks may be manageable for a non-developer, other modifications could pose challenges. Structural changes to chapters necessitate updates in both string files and the cloud storage database architecture, where even minor typos can trigger errors.

The wished requirement of having text-to-speech has not been implemented as stated in the introduction above. One could discuss the usefulness of that feature in the current product as the app lacks complex sentence structures and rather contains isolated words or short headlines. It might have been useful to have this feature for a user who cannot read when choosing languages, where the different languages could have been read out loud. The navigation is supported by both headlines and numeration, giving two possible sources of information that leads the user to the videos, which can be watched in their native tongue.

6.1.2. Meeting the Non-Functional Requirements

During user testing and use during the implementation of code, the non-functional requirements have been tested to some extent, shedding some light on the current state of the product in terms of performance, delivery, accessibility, usability, maintainability, and documentation.

While the documentation of the application meets basic standards, there is room for improvement. The backend code includes some comments, as illustrated in *Figure 76*, but the frontend lacks sufficient commentary. However, this thesis serves as comprehensive documentation for both the product and its development process.

The codebase has been structured with an emphasis on maintainability, as highlighted multiple times throughout the thesis. However, it is important to note that we have not conducted unit tests or integration tests. The application has undergone acceptance testing through user feedback and feedback from the P.O., which we have actively incorporated into feature development and modifications. There are instances of code repetition that could have been refactored for better reuse. Overall, the codebase is moderately maintainable, enabling the P.O. to update data and facilitating seamless handover for new developers to extend the project.

Our development approach in the front-end illustrates that we have prioritized usability, ensuring the application's adaptability across various screen sizes, from phones to tablets. While the primary focus was on optimizing the UX for phones, the design may not be as visually appealing on tablet-sized screens. The focus on phones and not tablets is due to the client stating that they rarely use tablets in teaching.

As it has been shown earlier in the thesis, our solution is based on recommendations and guidelines from Oslo Municipality (Oslo Kommune, n.d.), Material Design by Android (Google, n.d.), and Figma (Figma, n.d.-a). Both colors and font size have been chosen to enhance accessibility, which ensures contrast and manipulation of text size (uutilsynet, n.d.). *AutoSizeText* functionality combined with settings on the phone for text size allows the textviews or strings in the application to scale.

The product was completed ahead of schedule, enabling timely delivery and accessibility for the client. However, it is important to note that, as of now, videos in both sections have not been fully developed for all languages implemented in the application. Additionally, there remain opportunities for enhancing features and refining the overall structure to further improve the product.

The application's performance is deemed acceptable, demonstrating an ability to efficiently handle data streams and provide responsive user interactions. During data

retrieval from the database, a loading icon is displayed on the screen, enhancing the UX by indicating ongoing activity.

6.1.3. Working Methods and Communication

Looking back, our Scrum implementation as a working method did not work out as planned. There were several factors in how we used it that went wrong. One of them was that we did not enforce a Scrum Master. In the beginning of the project, we stated that we would have one, but we never actually chose one. At the start of the project, it did not seem to matter that much. Tasks were divided, and there was not that much else that needed to be communicated between meetings. But as time went by, there were a lot of suggestions from both group members and P.O. representatives. This, combined with the fact that we all made decisions ourselves without consulting the group or waiting for a consensus, made it challenging to work together as a group.

Another thing that went wrong was our implementation of Scrum meetings and how often we had them. We did set the Scrum meetings to be bi-weekly, but that was never enough. Also, the time chosen by the group to have the bi-weekly meetings did not always fit all participants, which meant that it could take a month before they checked in with the whole group in person. We never enhanced daily stand-ups, but that was also calculated due to the fact that all participants in the group had their schedules filled up and could therefore rarely meet at the same time.

We tried to create a file for making a shared production backlog that would be updated from week to week. This was never put into use, meaning that each participant in the group only kept some sort of control over what was implemented from week to week on their end. making it challenging to know for other participants when certain features were meant to be created or not by the participant in charge of them.

Another factor that is not related to Scrum directly but had an effect was the use of communication channels and ways of communicating. Since one group member lived in Ålesund and the rest in Oslo, the group never met all together in person. Additionally, two of the members based in Oslo would occasionally have to travel on a short notice for personal circumstances, making a regular schedule all the more difficult to implement. But some of the group participants in Oslo did meet each other in person on multiple occasions.

As stated, we used Facebook Messenger as the primary communication channel and supplied Discord and Zoom. The fact that our main communication channel was chatting, and rarely video calls or in person, could have been enough to make the process efficient with this type of communication. But since we only used the chat for certain periods and it involved all types of communication, it became difficult to ensure meaningful and effective

communication between group members. Important information could easily drown in chatting, and messages could easily be interpreted in the worst way.

As a result of the lack of implementation of Scrum and communication, the group ended up in a conflict that had to be resolved by the internal supervisor. In the meeting for resolving the conflict, the supervisor was set to be the Scrum Master, ensuring weekly meetings, the distribution of work, and checking in on work.

6.1.4. Re-Prioritization of Tasks

The second high-fidelity version of the app was based on the idea of a group member to develop a new design for one of the navigation paths. At this point, we had not yet implemented the feature to measure progress for the user for each individual video watched. Instead of placing this in the settings or language page, the group member thought that a new design could combine navigation and user progress in a satisfying way. This design ensured navigation in terms of an overview site while also giving feedback on the user's progress in terms of displaying green ticks to the right of the video headline (see *Figure 63*).

These changes were presented to the P.O., the group, and the internal supervisor all at the same time. The P.O. approved of such a feature and wanted it to be implemented in the app. Although it is a great idea, the idea has not gone through the 'proper channels'. For instance, we had not read or researched if such a navigation could benefit our application. Implementing such a feature made us have to re-prioritize tasks. It consumed a lot of time that could have been spent elsewhere: earlier user testing, implementing units, and integration tests enhancing Mockito. Reworking code makes it clearer, more reusable, and more effective.

One can also discuss if the new design for the navigation path has been successfully implemented. The feedback from some users in User Testing 2 states that the two different designs are a bit confusing (see *Chapter 3.3.3 User Testing (Round 2)*) Also, the navigation through the nav bar is currently not persisting when the user clicks into another tab. This was intentionally the idea that it should be, but it is a feature that was forgotten or dropped at the expense of the new design and its features.

6.2. Coding Challenges

One challenge early on was that multiple dependencies in Flutter could cause dependency issues with one another. One version of a dependency might not accept another version of another dependency. Changing these could cause the Android Gradle file to complain, and finding a good solution was not always easy.

In the development of the new design for one of the navigation paths after the first prototype, we had to develop an overview site over chapters and topics in a section. If we were to develop a page consisting of all the chapters and topics on one page, then one way to solve it would mean implementing nested lists. It was quite challenging to develop this logic for fetching the necessary video URLs and topics in a way that they were defined in the correct nested list, but also so that they always knew the next video in the list. In the old navigation, the videos only knew about the videos in their own chapter, but this type of overview navigation that went directly into a topic demanded that the current topic knew about the next chapter topic as well.

In the progression of the project, where we had recently conducted user testing 2, we had a meeting with our internal supervisor. With the time left in the project and the amount of work left to be done, it was highly recommended that we drop unit testing and integration testing with the use of mocking on behalf of upcoming tasks. This does not imply that the code has not undergone testing during development to verify its functionality. Rather, it suggests that we did not extensively test every edge case or error handling scenario when moving forward.

Another challenge was to keep the naming convention consistent, make code more effective or reusable, and comment along the way of the development. *Figure 67* demonstrates how mixing the mother tongue and English can be a challenge and is not advised to do. Also, naming convention suffers from the fact that the names of methods or variables might not communicate their real logic or essence, making it harder for the developers and others to comprehend what actually happens in the code.

Methods like *getDataAboutUserHaveSeenVideos* and *getDataAboutUsersHaveSeenVideosOtherLanguage* also demonstrate that the codebase is not as effective as it can be (see *Figure 83* and *Figure 84* respectively). Two almost identical methods are being utilized for the same purpose, but they are not cut down to one purely due to time constraints and the delivery of the application.

6.3. Lessons Learned and Changes for Next Time

The reality shown above demonstrates that not everything has gone according to plan. Although the product is working and will be delivered on time, there are some factors that could have made the process, the product, and the documentation a lot better. Our main issue was communication and how we implemented Scrum.

We should have had weekly meetings, which were too rare. Scrum.org suggests that each event has a shorter duration than the maximum of 1 month (Scrum.org, n.d.). Having more frequent Scrum events could have allowed us to have more control all together:

presenting the workload achieved from the past week, adding content to the shared product backlog, discussing challenges, adding, changing, or removing features, and communicating with P.O. It is not certain that an increase in frequent events besides daily stand-ups would have made an impact, but Scrum.org holds on to the idea that daily stand-ups should happen daily (Scrum.org, n.d.).

The daily stand-ups would have made an improvement in presenting what we were currently working on and how we would combine it with the other logic being developed by others. This could have been done through more frequent video meetings on Discord or Zoom and more active communication on Facebook Messenger without chatting about all things related to the project, but rather through multiple channels so that important information did not get lost so easily. This could have solved interpretation errors and reduced frustration and conflict.

By setting a Scrum Master, we could have avoided situations that lead to frustration regarding work distribution, implementation of new features, communication with P.O., meeting agendas, and making final decisions. This could have taken some of the communication overload of the remaining group members, but it also made decisions and discussions more predictable in terms of the fact that the Scrum Master had the last word and facilitated the discussion itself. Additionally, ensuring that every group member did what they stated they were doing.

7. Conclusion

In this report, we have shown the development process that culminated in the final product of the Livsmestring application. It has been demonstrated that the current product covers the main functionalities presented by the P.O. (*i.e.*, watch videos, be able to choose language, and see progress without the need for a user account). Furthermore, the application shows promising results in terms of performance due to responses from P.O. and user testing. Despite having such results and feedback, there are still some changes that could be made to improve the UX based on the data we got during User Testing 2 (refer to *Chapter 3.3.3 User Testing (Round 2)*).

Although the functionalities were met, the development process of the application could have gone smoother. *Chapter 2. Methodology* states that the development method of the application is Scrum. *Chapter 6. Reflections* then displays that the implementation of Scrum could have gone better and suffered due to a lack of a Scrum Master and daily stand-ups, but also that bi-weekly meetings were too rare. Communication issues based on this and the use of technologies to communicate caused conflicts in the group. which underlines the importance of having regular meetings face-to-face (in person or on screen) that are facilitated by a chosen leader who can set the agenda and make final decisions.

That being said, we are pleased with our product and the lessons that the process has taught us in terms of developing a product and how to work as a group. We hope that the P.O. will be happy with the final product and continue to develop it so that it can become a useful tool in the teaching of adult immigrants.

7.1. Future Work

We have learned that Oslo V.O. wants to continue on developing the application and use it in their teaching, which is great news. We have some recommendations on what can be looked at in future work with the application.

As the conclusion above states, there are improvements to be made based on the information we gathered during the round 2 of our user testing. Furthermore, during *Chapter 6. Reflections*, we saw that the code base can be made more effective and clean. This can be achieved by making some code reusable and removing ‘duplicate methods’.

The navigation to videos via the health or career tab in the nav-bar is currently stateless in terms of preserving the navigation state when it is ‘destroyed’. It is our recommendation that it might be something to look into and implement it in a way that the

state is preserved so that the navigation or the video can be loaded back at the last visited position.

Another functionality to consider is the text-to-speech function. The report has shown that there is not a lot of text in the application, and it therefore might not be that useful, but on the other side, it might be useful for the user to have some components read out loud and highlighted to start a video on their preferred language.

8. Bibliography

Bird, C. (2016). Interviews. In *Perspectives on Data Science for Software Engineering* (pp. 125–131). Elsevier. https://doi.org/10.1016/B978-0-12-804206-9.00025-8

Bligård, L.-O., & Osvalder, A.-L. (2013). Enhanced Cognitive Walkthrough: Development of the Cognitive Walkthrough Method to Better Predict, Identify, and Present Usability Problems. *Advances in Human-Computer Interaction*, *2013*, 1–17. https://doi.org/10.1155/2013/931698

Brekke, J.-P., Fladmoe, A., Hesstvedt, S., & Wollebæk, D. (2024). *Holdninger til innvandring, integrering og mangfold—Integreringsbarometeret 2024* (7). INSTITUTT FOR SAMFUNNSFORSKNING. https://hdl.handle.net/11250/3128272

Burton-Jones, A., & Hubona, G. S. (2006). The mediation of external variables in the technology acceptance model. *Information & Management*, *43*(6), 706–717. https://doi.org/10.1016/j.im.2006.03.007

Cheng, M., An, S., Cheung, C. F., Leung, Z., & Chun, T. K. (2023). Gerontechnology acceptance by older adults and their satisfaction on its servitization in Hong Kong. *Behaviour & Information Technology*, *42*(16), 2932–2951. https://doi.org/10.1080/0144929X.2022.2151936

Christensen, D. A. (2024). *Immigrant inclusion in Norway: An analysis of immigrant adaption and native reaction*. https://www.samfunnsforskning.no/english/projects/aktive/innvandring-og-inkludering-reformer-politikk-og-en.html

Figma. (n.d.-a). *Color Contrast Checker*. Figma. Retrieved 22 May 2024, from https://www.figma.com/community/plugin/1218666737106149812/color-contrast-checker

Figma. (n.d.-b). *Components, styles, and shared library best practices*. Figma. Retrieved 22 May 2024, from https://www.figma.com/best-practices/components-styles-and-shared-libraries/

Figma. (n.d.-c). *Contrast*. Figma. Retrieved 22 May 2024, from https://www.figma.com/community/plugin/748533339900865323/contrast

Google. (n.d.). *Material Design*. Material Design. Retrieved 22 May 2024, from https://m3.material.io/?fbclid=IwAR2znbCIOCSPa-mdNZwk14xU0Wm_6xXysRxojy58U2op41SIVdTmPptQUc4

Google for Developers. (n.d.). *Download files with Cloud Storage on Flutter | Cloud Storage for Firebase*. Retrieved 7 April 2024, from https://firebase.google.com/docs/storage/flutter/download-files

Halsne, P., Chornous, O., Alquézar, A. B., & Ali, S. S. A. (2024). *Pre-project*. https://docs.google.com/document/d/1dAxNG5antcE_Kcmi40pBd6rIzEmHG04nL88IuTiHFX4/edit?usp=drive_web&ouid=110462731416987534876&usp=embed_facebook

Hufnagel, E. M., & Conca, C. (1994). User Response Data: The Potential for Errors and Biases. *Information Systems Research*, *5*(1), 48–73. https://doi.org/10.1287/isre.5.1.48

IMDi. (n.d.). *Livsmestring i et nytt land*. Fagressurser for Introduksjonsprogrammet. Retrieved 22 May 2024, from https://introduksjonsprogrammet.imdi.no/innhold/livsmestring-i-et-nytt-land/

Inostroza, R., Rusu, C., Roncagliolo, S., & Rusu, V. (2013). Usability heuristics for touchscreen-based mobile devices: Update. *Proceedings of the 2013 Chilean Conference on Human - Computer Interaction*, 24–29. https://doi.org/10.1145/2535597.2535602

ISO. (2019). ISO 9241-210:2019 Ergonomics of human-system interaction—Part 210: Human-centred design for interactive systems. *International Organization for Standardization*. https://www.iso.org/standard/77520.html

Lewis, C., & Wharton, C. (1997). Cognitive Walkthroughs. In *Handbook of Human-Computer Interaction* (pp. 717–732). Elsevier. https://doi.org/10.1016/B978-044481862-1.50096-0

Lovdata. (2020, November 6). *Lov om integrering gjennom opplæring, utdanning og arbeid (integreringsloven)*. https://lovdata.no/dokument/NL/lov/2020-11-06-127

Nielsen, J. (Ed.). (1994a). *Usability inspection methods*. Wiley.

Nielsen, J. (1994b, April 24). *10 Usability Heuristics for User Interface Design*. Nielsen Norman Group. https://www.nngroup.com/articles/ten-usability-heuristics/

Norman, D. A. (2013). *The design of everyday things* (Revised and expanded edition). Basic Books.

OECD. (n.d.). *Skills and Labour Market Integration of Immigrants and their Children in Norway*. OECD iLibrary. https://www.oecd-ilibrary.org/sites/ea9050b2-en/index.html?itemId=/content/component/ea9050b2-en

Oliveira, E. R., Branco, A. C., Carvalho, D., Sacramento, E. R., Tymoshchuk, O., Pedro, L., Antunes, M. J., Almeida, A. M., & Ramos, F. (2022). An Iterative Process for the Evaluation of a Mobile Application Prototype. *SN Computer Science*, *3*(4). https://doi.org/10.1007/s42979-022-01153-6

Oslo Kommune. (n.d.). *Designmanual—Oslo kommunes visuelle identitet*. Designmanual for Oslo Kommune. Retrieved 22 May 2024, from https://designmanual.oslo.kommune.no/

Scrum.org. (n.d.). *Introduction to the Scrum Events*. Scrum.Org - The Home of Scrum. Retrieved 22 May 2024, from https://www.scrum.org/resources/introduction-scrum-events

Shneiderman, B., & Plaisant, C. (2005). *Designing the user interface: Strategies for effective human-computer interaction* (4th ed.). Pearson/Addison-Wesley.

Smith, S. L., & Mosier, J. N. (1986). *Guidelines for designing user interface software* (SRI-CSL-86-2). SRI International, Computer Science Laboratory. https://hcibib.org/sam/
Sommerville, I. (2016). *Software engineering* (Tenth edition, global edition). Pearson.
Stevens, E. (2024, April 9). *The 10 Best UI Design Tools to Try in 2024*. https://www.uxdesigninstitute.com/blog/user-interface-ui-design-tools/
Trust Center—Security and Compliance. (n.d.). Google Cloud. Retrieved 8 April 2024, from https://cloud.google.com/trust-center
uutilsynet. (n.d.). *WCAG-standarden*. uutilsynet. Retrieved 22 May 2024, from https://www.uutilsynet.no/wcag-standarden/wcag-standarden/86
Venkatesh, V., & Davis, F. D. (2000). A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies. *Management Science*, *46*(2), 186–204. https://doi.org/10.1287/mnsc.46.2.186.11926
Williams, M., & Moser, T. (2019). The Art of Coding and Thematic Exploration in Qualitative Research. *International Management Review*, *15*, 45–55.
Wynn, D. C., & Eckert, C. M. (2017). Perspectives on iteration in design and development. *Research in Engineering Design*, *28*(2), 153–184. https://doi.org/10.1007/s00163-016-0226-3

Appendix

Appendix A: Sprints

A1: Infographic

A2: Sprint Information

Sprint 1

Duration: 29/01 - 04/02 (W5).

Goal: Setup the project and plan the schedule.

Activities:

- Define the minimum viable product (MVP).
- Divide the workload.

Sprint 2

Duration: 05/02 - 18/02 (W6-7).

Goal: Deepen our knowledge.

Activities:

- Learn the tools we are going to use (Figma and Flutter).
- Sketch a basic application structure.
- Create user stories.

Sprint 3

Duration: 19/02 - 03/03 (W8-9).

Goal: Begin frontend and backend development.

Activities:

- Frontend:
    - Brainstorm designs.
    - Define application themes (colors, fonts, etc.).
    - Create two Figma prototypes.
- Backend:

- Connect to Firebase database.
- Make logic for buttons, icons, and nav-bars.
- Make text-resource with easy scaling for multiple languages.

- Testing:
  - A cognitive walkthrough.
  - Heuristic testing before user testing round 1.
  - User testing (app flow, cultural affordances, user preferences, etc.).
  - Integration test of database.
  - Unit testing.

Sprint 4

Duration: 04/03 - 17/03 (W10-11).

Goal: Continue frontend and backend development.

Activities:

- Frontend:
  - Product owner review.
  - Choose the next iteration of the design.
  - Create an interactive prototype.
- Backend:
  - Make channel calls for using SharedPreferences in Android and UserDefaults in iOS for persisting progress.
  - Make logic to show videos in each chapter.
- Testing:
  - Heuristic testing before user testing round 2.

- User testing.
- Unit testing.

Sprint 5

Duration: 18/03 - 31/03 (W12-13).

Goal: Integration of the frontend with the backend, polishing phase.

Activities:

- Frontend:
  - Product owner review round 2.
  - Choose the final design.
  - Polish the final UI.
- Backend:
  - Create the remaining logic.
  - Integrate text-to-speech functionalities for illiterate users.
  - Unit testing.

Sprint 6

Duration: 01/04 - 14/04 (W14-15).

Goal: Application testing and debugging.

Activities:

- Final unit testing.
- Double check integration testing for Firebase database.
- Failure testing.
- Acceptance testing with the product owner.

Sprint 7

Duration: 15/04 - 28/04 (W16-17).

Goal: Deployment and documentation.

Activities:

- Deploy the application on the App Store and Play store.
- Document the Readme-file and add comments to the code.

Sprint 8

Duration: 29/04 - 12/05 (W18-19).

Goal: Post-launch monitoring and maintenance.

Activities:

- Monitor the application post-launch.
- Begin writing the report.

Sprint 9 (Final Sprint)

Duration: 13/05 - 24/05 (W20-21).

Goal: Deliver project report.

Activities:

- Finish writing project report.
- Quality Check.

Appendix B: User Testing Documentation

B1: Sikt Application for Data Processing

Error! Objects cannot be created from editing field codes.

B2: Consent Form (English)

Invitation and consent form for project «Livsmestringsappen»

Dear Participants,

We are inviting you to participate in a research study. Oslo voksenopplæring Helsfyr asked us to develop the Livsmestringsapp, as a companion resource in teaching the course Coping with Life for immigrants. Therefore, *the purpose of this study* is to investigate the user's experience with the Livsmestringsapp. When you participate in this study, you are expected to co-design with us, evaluate usability and accessibility aspects of the app's prototype. The obtained data could help us to identify issues with the app's design and features. As a result, we will find solutions for improving the app (if necessary).

The following information is provided to help you make an informed decision about participating in this study:

Study Procedures: The study consists of interview and user testing of the prototype during approximately 30-45 min. We will test design and technological, and functional features of the app's prototype, and not your physical/psychological possibilities. The conversation will be recorded during the testing. Received data (the recording) will be used to reference information that can improve the prototype, and no more.

At the beginning, we ask you to provide us with personal information about you: age, gender, educational background, ethnicity, your health conditions. This will help us understand your functional and cognitive abilities, and digital competency. After that, we will ask you to perform a series of testing tasks according to our instructions. During the process, you will be asked questions that are aimed at improving the design of the prototype. Thus, you will co-design the app with us.

Confidentiality: All information collected during this study will be kept confidential. Your name and personal information that will identify you will not be disclosed. Only the researcher that is in contact with you knows this information. This information will be stored in a document, in a separate area on encrypted memory sticks. Other collected data, for example the recording and notes during the testing, will also be stored in the same way. The collected personal information such as age, gender, educational background, and digital competency will be anonymized when it is published in the (student) research work and/or articles. It is possible that anonymized quotes from the study will be used in such publications. The entire project is scheduled to end in June 2025. The information will then be deleted at the end of the project.

Benefits: We hope that the information obtained from this study can improve the user experience with Livsmstringsapp.

Withdrawal: You have the right to withdraw from the study at any time without any negative consequences.

Voluntary participation: Your participation in this study is voluntary, when you make a decision about it on your own. In case of your decision to take part in this study, you will be asked to sign a consent form. After signing the consent form, you can still withdraw from participation at any time and without giving a reason. In case of your withdrawal before data collection is completed, your data will be returned to you or destroyed.

Information about you will be processed based on your consent. As long as you can be identified in the data material, you have the right to:

- access personal data, which are registered about you,
- to have personal data about you corrected or/and deleted,
- receive a copy of your personal data (data portability), and
- to send a complaint to the Data Protection Commissioner or the Norwegian Data Protection Authority about the processing of your personal data.

Lastly, we would like to inform you that

- OsloMet is the data controller and
- No personal data will be collected and/or processed by the tested prototype.

If you have questions related to the project, contact the researcher:

- Petter Halsne, bachelor student: e-mail: s320947@oslomet.no , Tel: 97526818
- Way Kiat Bong, Associate Professor: wayki@oslomet.no, Tel: 96724429
- Our data protection representative: Ingrid S. Jacobsen (ingridj@oslomet.no)
- SIKT – Kunnskapssektorens tenesteleverandør, via email (kontakt@sikt.no) or phone: 73984040.

As your ethnicity will be identified due to the user testing, consent from parents and/guardian shall be provided as well, if you are below 18 years old. By signing this consent form, you confirm that you have read and understood the information provided above and that you/children under the care of a guardian are voluntarily agreeing to participate in this study.

If you have any questions or concerns about the study, please do not hesitate to ask.

Participant's Signature: _____

Parent/ guardian's Signature (if applicable):: _____

Date: _____

Thank you for considering participating in this study.

B3: Usability Testing Protocol (Second Version)

USABILITY TESTING PROTOCOL 2

Objective:

To conduct the usability testing of Livsmestringsappen among immigrants who are taking the course "Coping with life" in Oslo voksenopplæring Helsfyr. This usability testing seeks to uncover a range of issues related to the app's performance, including its efficiency, effectiveness, ease of learning, accessibility, user satisfaction, navigational clarity, and the quality of its content and interface design. Ultimately, the goal is to evaluate the app's overall quality and user experience.

Below are the tasks that participants will carry out within the app, along with the corresponding questions.

Note: In case the participant cannot complete the task, the Interviewer will offer assistance as necessary.

During the task execution, it's important to record the time taken by each participant to finish it, along with specifying whether they completed it independently or with interviewer assistance. After the session, it's essential to inquire about the participant's opinion on the overall aesthetic of the design, their comprehension of the features, and their attitude towards using the app for learning the course about integration into Norwegian society. Additionally, it's crucial to gather the participant's perspective on how the goal setting corresponds with their experience with tracking progress of studying (if there's time for that during the session).

Participants are encouraged to provide any suggestions they have for improving the design.

Before testing

- ➢ Collect information on age, gender, and education level.
- ➢ Ensure that the app is installed correctly.
- ➢ Make an appointment (time and place of the interview, should not be noisy).

Briefing

- Background of the study:
    - Application designed for individuals aged 18 and older who have relocated to Norway and are enrolled, as pupils, in adult education centers.
    - 1 main functionality of the App, but also other functionalities..
    - The app is undergoing testing, not individuals.
    - Let us know what you think or wonder about while performing the tasks.

- Make sure that participant has given consent.
- Check the screen and the audio recorder (they should be charged, function properly and the memory card should be clean).
- Start recording.
- Ask the participant about his/her ICT skills from a scale of 1-10 (1 - is for "very bad", 10 - is for "very good").

Usability testing

Subject ID:

Date of Evaluation:

Task Name	Task completed independently y (Y/N)	Time Set (In minutes)	Time Taken by Subject	Notes (Observation, comments)
Choose a language and open the chapter “Career”.		1		
Go back to the Home page, and open the “Career” chapter using a different path than last time.		2		
Choose any chapter and open the topic, which you are interested in.		2		
1) Expand video to full screen, Start the video. 2) Change the playback speed. Stop the video. 3) Reduce video screen to the original size		3		
Go to the next section with the video. Come back to the previous video page (which you visited recently).		3		
Go to the Home page, and then go back to the video you were recently watching.		3		
Find the chapter “Me in context” in the Career section, and start the video “1.2 New Words”.		3		
Find the chapter “Adaptations and Resistance” in the Career section. Start the video “4.3 Cases”.		3		
Go to Settings. Change to a different language.		2		

Questions after tasks:

- What do you think about the application?
- Was there anything you missed in the application?
- Can you describe any difficulties or challenges you encountered while completing the tasks?
- Were there any features that you found particularly intuitive or easy to use?
- Were there any specific elements of the user interface that you found confusing or unclear?
- Did you feel confident in your ability to navigate through the application? If not, what aspects contributed to not feeling confident?
- Did you encounter any bugs, glitches, or technical issues while using the application? If yes, please describe.
- Were there any features that you expected to find in the application but didn't? If yes, what were they?
    - Did you find the loading icon satisfying before the content appeared on the screen?
    - Do you like how topics in chapters are listed out (displayed) on the screen? Did you realize that they can be scrolled?
    - How do you understand the bar in the middle of the Career card and the Health card?
    - How did you find the size of the text in the application?

System Usability Scale Questionnaire

	A statement / Opinion	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
		1	2	3	4	5
1.	I think that I would like to use this system frequently.					
2.	I found the system unnecessarily complex.					
3.	I thought the system was easy to use.					

4.	I think that I would need the support of a technical person to be able to use this system.					
5.	I found the various functions in this system were well integrated.					
6.	I thought there was too much inconsistency in this system.					
7.	I would imagine that most people would learn to use this system very quickly.					
8.	I found the system very cumbersome to use.					
9.	I felt very confident using the system.					
10.	I needed to learn a lot of things before I could get going with this system.					

Appendix C: User Testing Evaluations

C1: Technical Considerations

During our study, the ability to extract and understand the intended user base conceptual model was essential to crafting an effective user testing phase and product. This methodology involved presenting the users with our prototype and requesting that they explain its components and functionalities. For example, participants were asked to explain their understanding of features such as the progress bar, or to give insight regarding what a certain icon might mean to them. This provided us with valuable data on how they perceive and understand the system and its elements.

This approach is grounded in cognitive psychology, which stresses the importance of aligning UI design with the users' mental models to reduce frustration and improve the overall UX (Norman, 2013). Historically, methods like conceptual model extraction have proven effective in revealing users' understandings of complex systems, as demonstrated by Smith and Mosier (1986), who employed a similar approach and highlighted its effectiveness in revealing users' understanding of different interface elements and their relationships.

This approach ultimately helped us enhance our prototype and final product, and aligns with user-centred design principle (ISO, 2019), ensuring that user feedback directly influences the development process.

The technology acceptance model (TAM) also provided a framework to evaluate how user expectations align with their actual experiences, particularly in terms of the app's usability and effectiveness. By assessing factors such as UI design and functionality, we aimed to improve user acceptance and satisfaction. Burton-Jones & Hubona (2006) underscored the significance of individual differences, including age, education, and system experience, which directly influence usage patterns as seen below in *Figure 69***.**

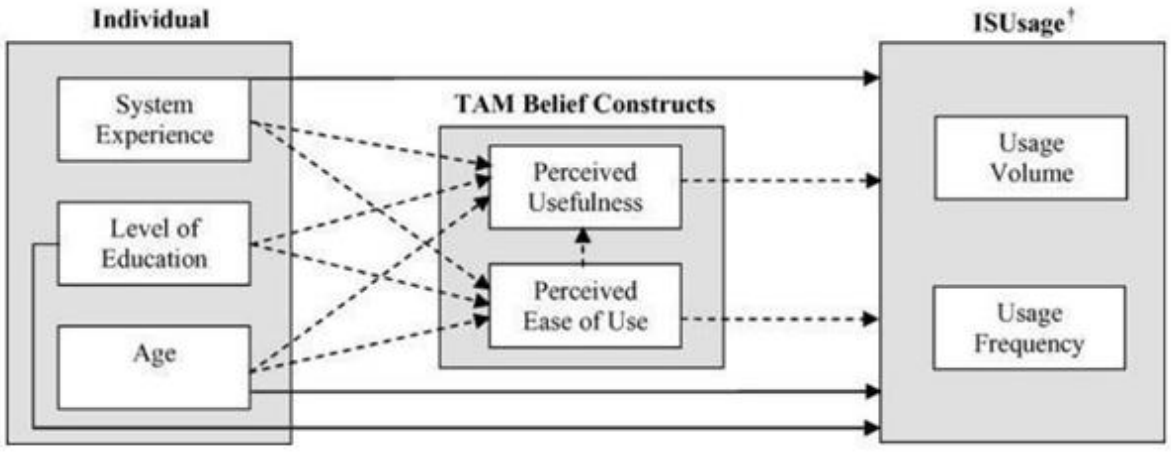


Figure 69.- *The direct effects of individual differences on usage, ISUsage - information systems usage (Burton-Jones & Hubona, 2006, p. 708).*

Moreover, the comprehensive study by M. Cheng et al.(2023) highlighted key factors within the TAM model, including value, usability, affordability, and social support, among others. Our findings indicated that while some TAM variables like societal image were less relevant, other aspects such as subjective norms and output quality significantly influenced user perceptions. This model can be observed in *Figure 70.*

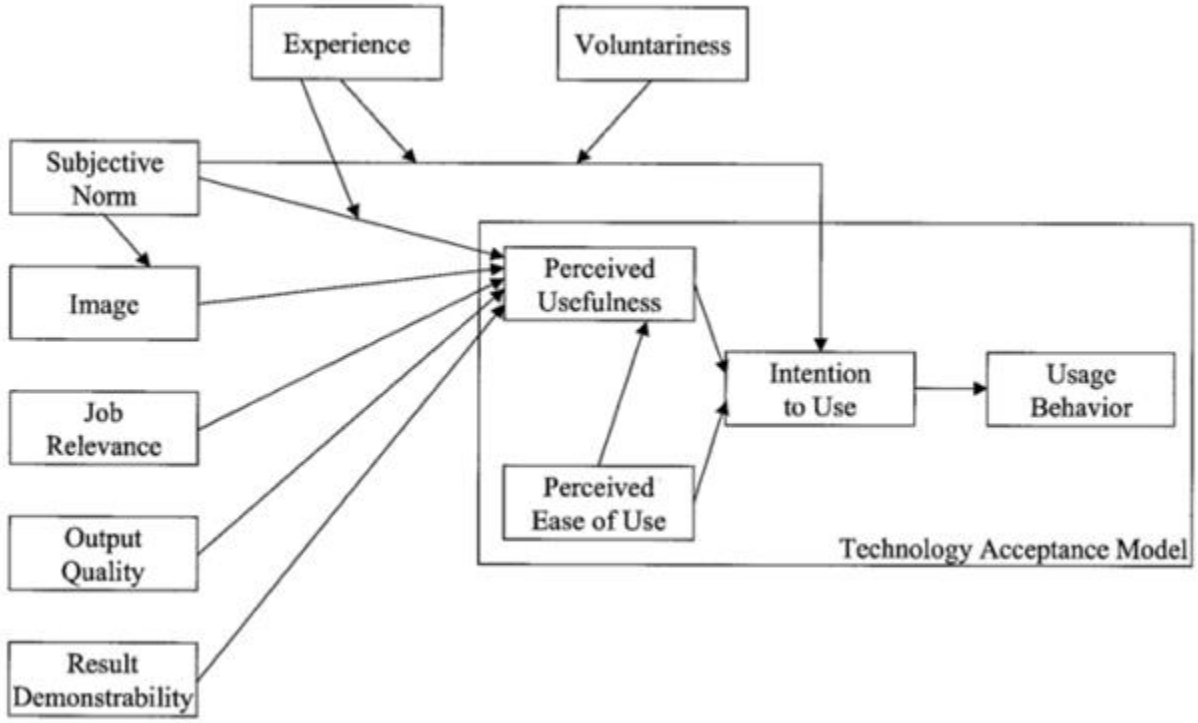


Figure 70.- *The technology acceptance model TAM (Venkatesh & Davis, 2000).*

Finally, after each round of user testing, we analyzed the results by using open coding, axial coding, and the affinity diagram method to meticulously parse the data we collected. These structure coding procedures ensured the validity and reliability of our findings (Williams & Moser, 2019, p. 53).

During the open coding process, we highlighted key phrases and dissected the interview text line by line, categorizing the data into positives, negatives, and neutral observations. This segmentation of data facilitated a comprehensive comparison of similar tasks across different interviews, allowing us to identify patterns and unique insights. Subsequently, we used axial coding to help establish connections between the identified categories. As described by Williams and Moser (2019, p. 50), axial coding refines and categorizes emergent themes, allowing for a deeper understanding of the data.

C2: Conclusion

Overall, both user tests offered crucial data on the app's usability, design, and functionality. It was very enlightening to test participants with vastly different backgrounds and observe how those differences impact the experiences that the users have with the app, and the type of feedback that they provide.

For instance, users with self-proclaimed lower ICT skills were seemingly more preoccupied with being able to navigate the application properly. They have less interest in

additional features that the app might provide, the thought process behind the design, and a lower threshold of patience. As a result, their feedback was more focused on the quality of the navigation and whether their expectations were met or betrayed when exploring some functionality.

On the other hand, participants with a higher self-proclaimed level of ICT skills demonstrated a higher level of curiosity and patience and were more focused on providing feedback on the design (*e.g.*, color contrast, tactile feedback). This kind of feedback can be very valuable to refine and polish the visual appeal of the UI, but it might not reflect the average user's experience and could distract from more important quality of life changes that the app might need.

It also became evident that an effective system evaluation is essential for meeting user needs, a principle evident throughout the Livsmestring app design process. Employing triangulation, we utilized various methods such as interviews, questionnaires, observation, constructive interaction, and the Think Aloud technique. This multifaceted approach ensured a comprehensive understanding of UX, guiding iterative improvements to enhance the app's usability and utility.

Appendix D: Figma Prototype

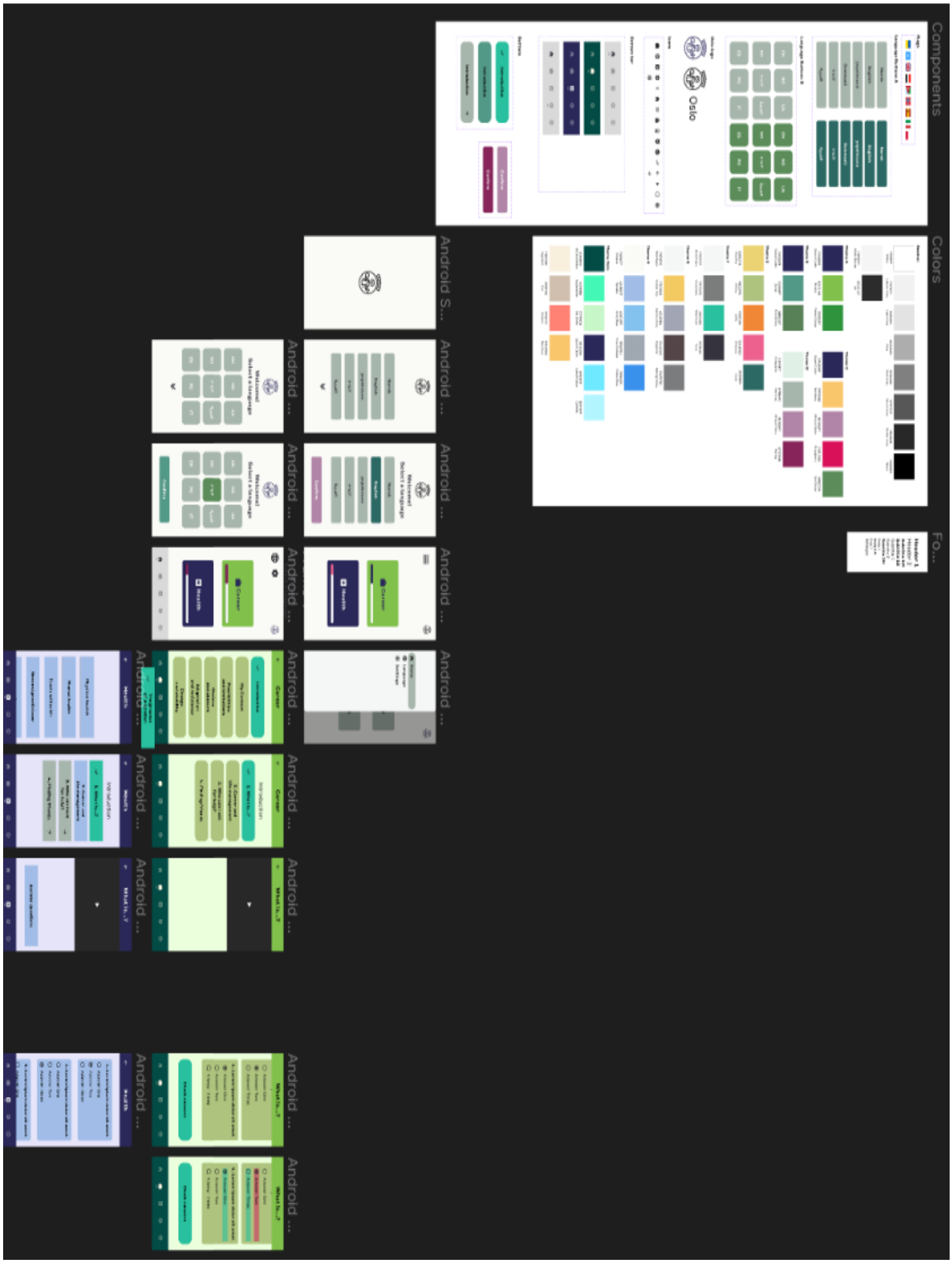


Figure 71.- *Figma design components and screens*

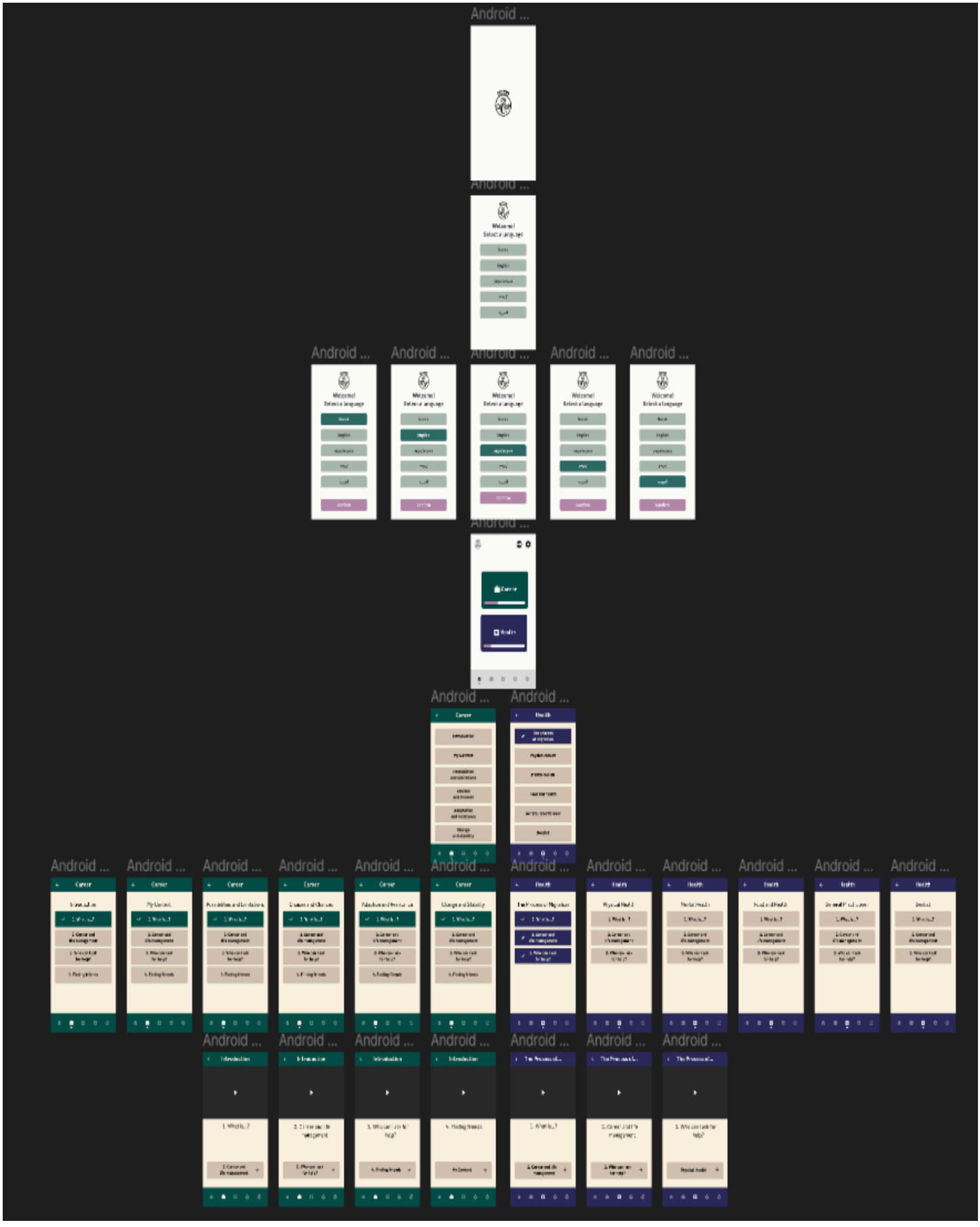


Figure 72.- *Figma prototype in flow mode*

Appendix E: Technical Specifications

E1: First Hi-Fi Prototype

Architecture

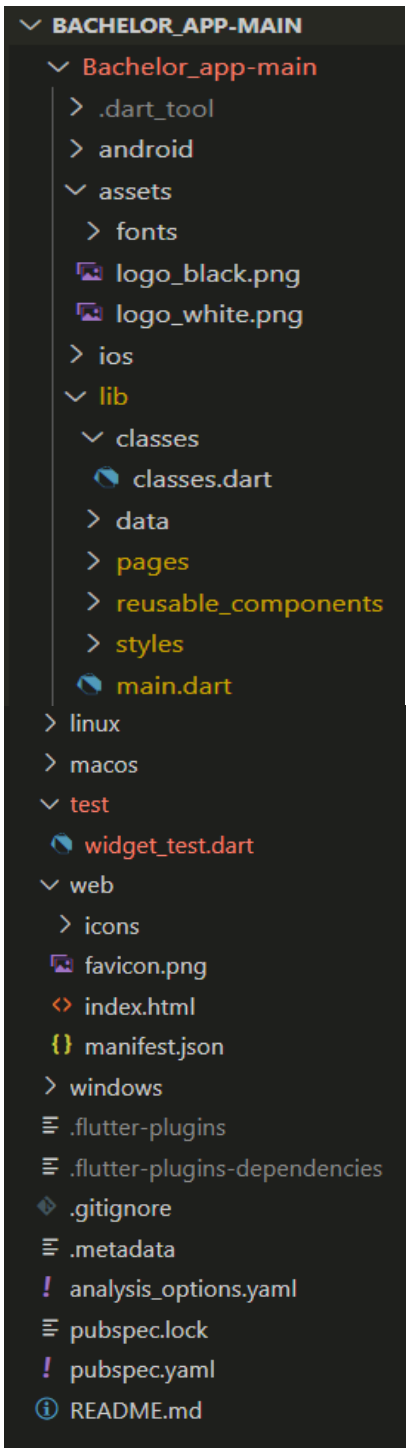


Figure 73.- *Main overview of the project's structure*

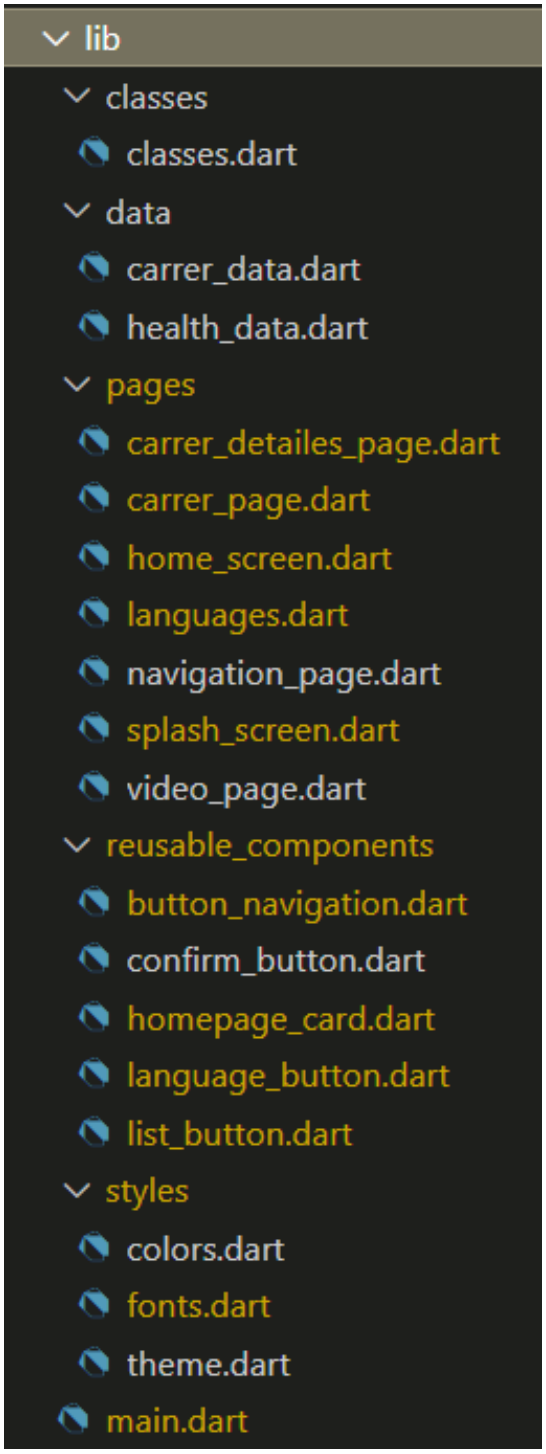


Figure 74.- *Contents of lib directory*

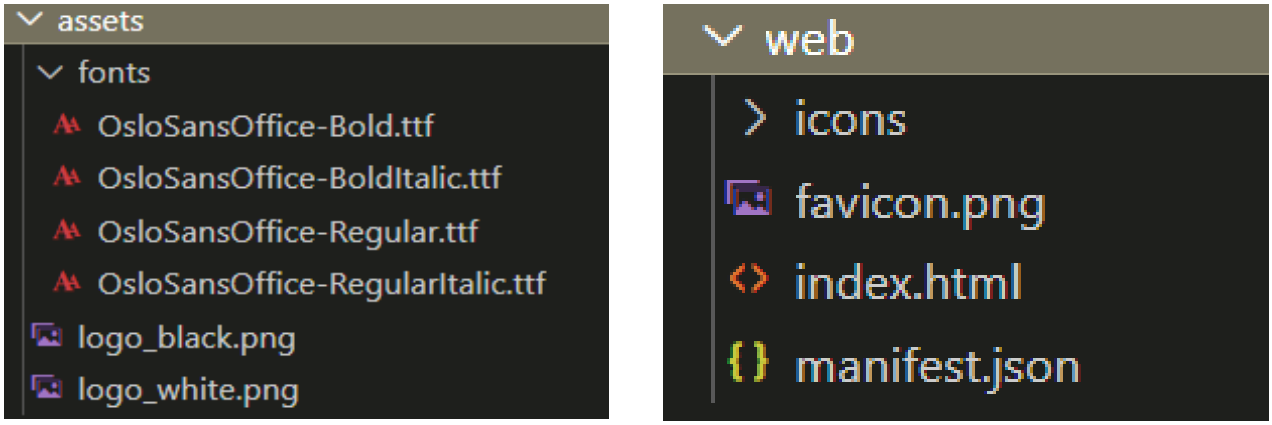


Figure 75.- *Contents of assets directory and web directory, respectively*

Reusable Components

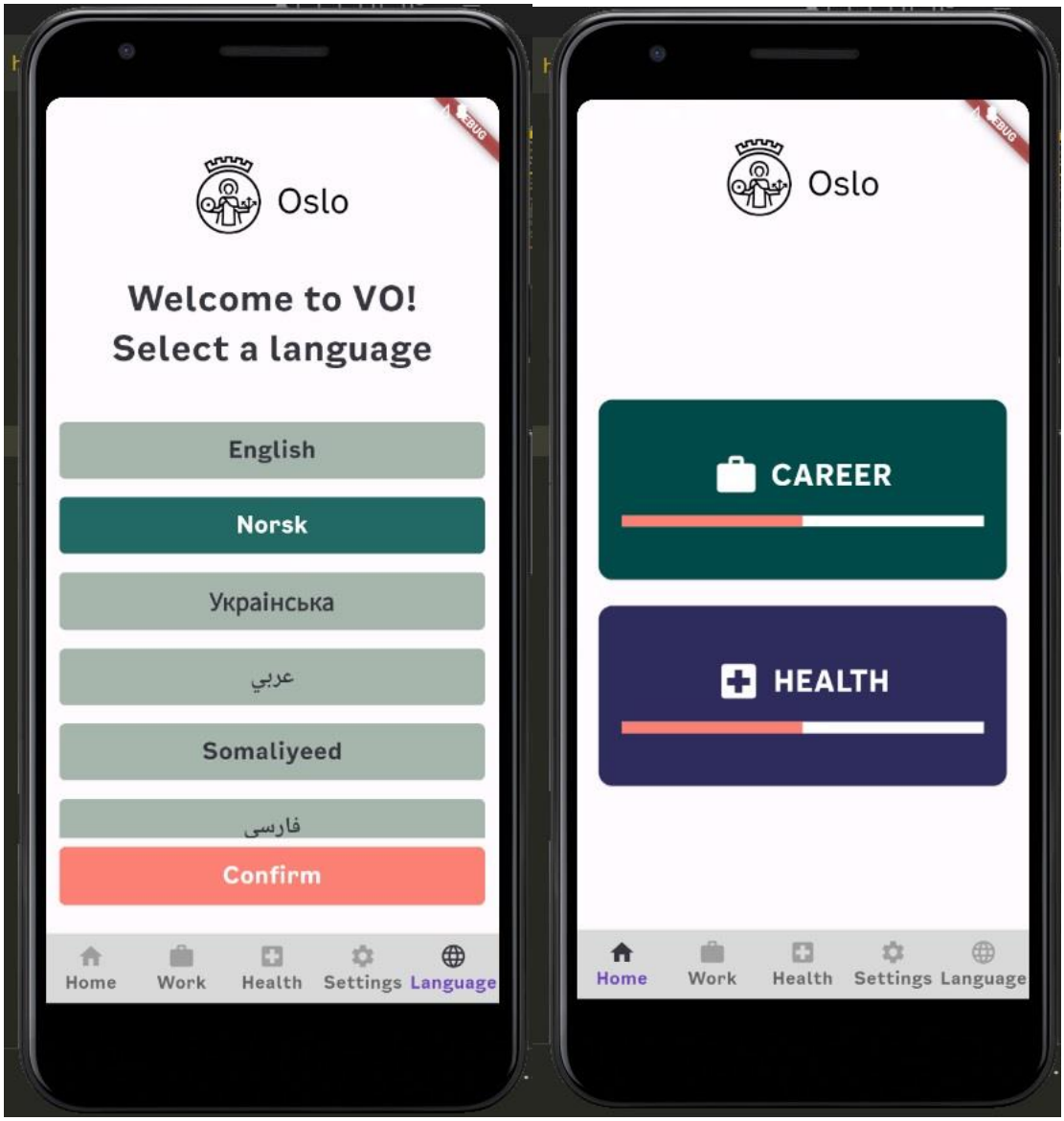


Figure 76.- *On the left, LanguageButton and ConfirmButton components, on the right HomePageCard components*

Implementation Details

To commence engagement with the Livsmestring app, the following procedural steps are recommended:

1. Clone the pertinent repository for your local computing environment.
2. If not already installed, procure and set up the Flutter framework.
3. Proceed to the designated project directory and execute the command ‘flutter pub get’ to initialize the installation of the requisite dependencies.
4. Launch the application by executing the command ‘flutter run’.

Usage of the reusable components and styling files within the app can be exemplified through various scenarios:

1. Integration of Reusable Components: Upon instantiation of new interface elements or features within the Livsmestring app, developers leverage the pre-existing reusable components stored within designated files. For instance, components such as buttons, input fields, or navigation bars are consistently employed across diverse sections of the application.
2. Consistent Styling Across Screens: The styling files, including *colors.dart* and *fonts.dart*, ensure uniform visual aesthetics throughout the application. As different screens are developed, these files are referenced to maintain coherence in color palettes, font choices, and text formatting standards, as prescribed by the Oslo Common Design Guidelines.
3. Adherence to Design Guidelines: Usage of the reusable components and styling files facilitates adherence to established design guidelines. Developers consistently reference these files to ensure that the implemented UI elements align with the prescribed visual standards, thereby fostering a cohesive UX.
4. Efficient Maintenance and Updates: By centralizing reusable components and styling definitions, maintenance and updates become more efficient. Changes to the visual appearance of the application can be easily implemented by modifying these files, thereby minimizing the need for extensive refactoring across multiple sections of the codebase.

In summary, the utilization of reusable components and styling files within the Livsmestring app serves to promote consistency, adherence to design guidelines, and efficiency in development and maintenance processes.

E2: Second Hi-Fi Prototype

Architecture

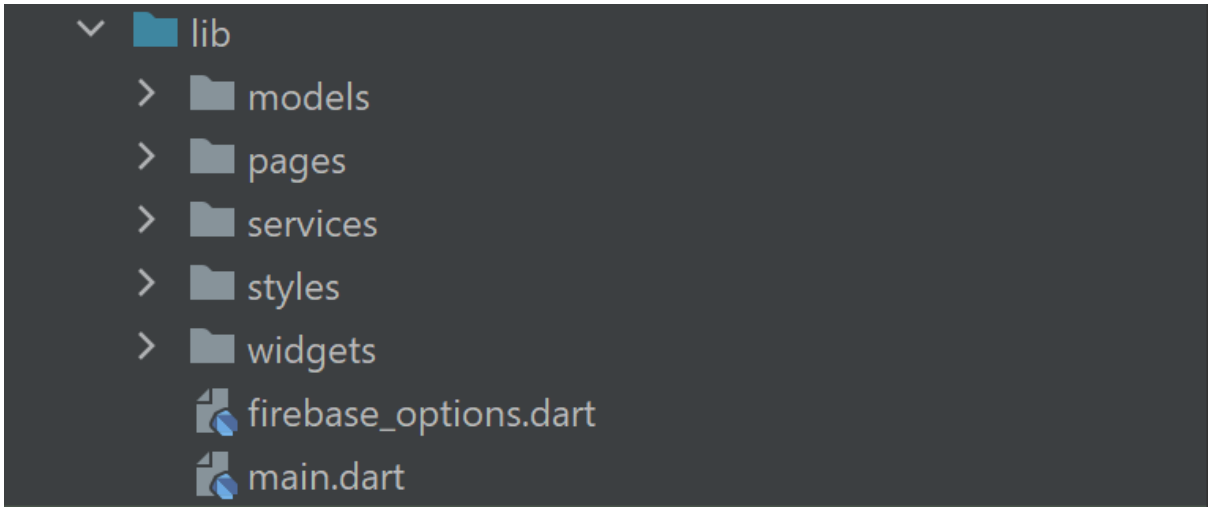


Figure 77.- *File-structure inside the livsmestrings-project at start of second version*

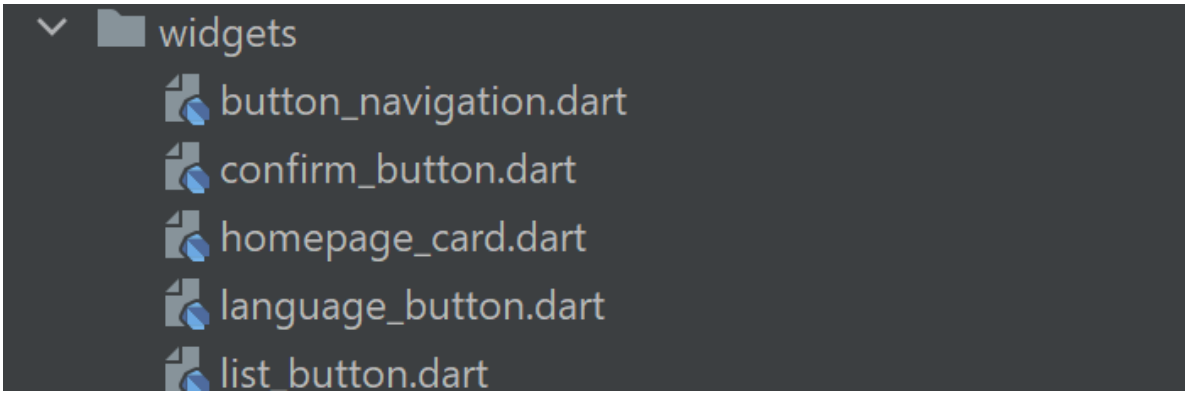


Figure 78.- *New widgets folder*

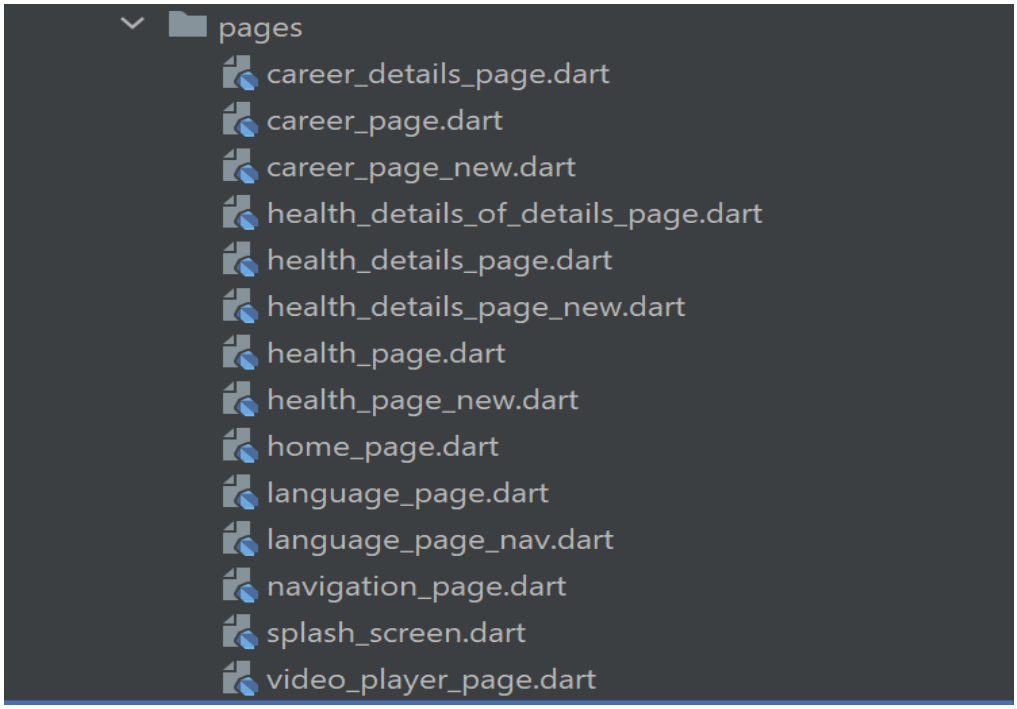


Figure 79.- *New pages folder*

Code Snippets for Methods

```dart
//This method takes in a list of strings in the parameter, which is the videoTitles fetched for the current view.
//The method then retreives the first numeral value from the first item in the list.
// This is for determing which chapter we are in career.
//Then it fetches all the videoUrls for the given path based on the numeral value, this
// videolist of urls is returned back to the view.
Future<List<String>> getVideoUrlsCareer(List<String> careerModuleVideoTitles) async {
  try {

    num extractNumber(String str) {
      final RegExp regex = RegExp(r'^(\d+(\.\d+)?)(_.+)?$');
      final match = regex.firstMatch(str);
      return double.parse(match?.group(1) ?? '0');
    }

    Database database = Database();
    List<String> videoUrls = [];

    num firstItem = extractNumber(careerModuleVideoTitles.first);
    print("Detter er items før: $careerModuleVideoTitles");

    String currentLanguage = await getCurrentLanguage();
    String videoPath = "videos/" + currentLanguage + "/";

    print("This is firstItem: $firstItem");

    if (firstItem == 1.1) {
      videoPath = videoPath + "karriere/1.Meg_I_Kontekst/";
      videoUrls = await database.fetchAllVideos(videoPath);
    }
    else if (firstItem == 2.1) {
      videoPath = videoPath + "karriere/2.Muligheter_Og_Begrensninger/";
      videoUrls = await database.fetchAllVideos(videoPath);
    }
```

```dart
    else if (firstItem == 3.1) {
      videoPath = videoPath + "karriere/3.Valg_Og_Tilfeldigheter/";
      videoUrls = await database.fetchAllVideos(videoPath);
    }
    else if (firstItem == 4.1) {
      videoPath = videoPath + "karriere/4.Tilpasning_Og_Motstand/";
      videoUrls = await database.fetchAllVideos(videoPath);
    }
    else if (firstItem == 5.1) {
      videoPath = videoPath + "karriere/5.Endring_Og_Stabilitet/";
      videoUrls = await database.fetchAllVideos(videoPath);
    }
    else {
      print("getVideoUrls: Den er ikke en del av denne listen");
    }
    print("Returning Videos");
    return videoUrls;
  }
  catch (e) {
    print('Error in getVideoUrlsCareer: $e');
    return [];
  }
}
```

Figure 80.- *Illustration of getVideoUrlsCareer() shows the logic for itself and getVideoUrlsHealth()*

```
598  Future<List<VideoItem>> nextVideoItems(List<String> url, List<String> title) async {
599    try {
600      List<VideoItem> nextVideoReversed = [];
601      List<VideoItem> nextVideo = [];
602
603      // Check if both lists are not empty and have the same length
604      if (url.isNotEmpty && title.isNotEmpty && url.length == title.length) {
605        // Iterate through the list of titles and URLs
606        VideoItem videoItem;
607        int counter = -1;
608        for (int i = url.length; i != 0; i--) {
609          // Create a new VideoItem
610          if (i == url.length) {
611            //The last item should be emptey
612            videoItem = VideoItem(title: "", url: "", nextItem: null);
613          }
614          else {
615            videoItem = VideoItem(title: title[i],
616                url: url[i],
617                nextItem: nextVideoReversed.elementAt(counter));
618          }
619          // Add the VideoItem to the list
620          nextVideoReversed.add(videoItem);
621          counter++;
622        }
```

```
623
624          for (int i = nextVideoReversed.length - 1; i > -1; i--) {
625            nextVideo.add(nextVideoReversed[i]);
626          }
627
628        }
629        else {
630          int urlSize = url.length;
631          int titleSize = title.length;
632        }
633
634        return nextVideo;
635      }
636      catch (e){
637        print('Error in nextVideoItems(): $e');
638        return [];
639      }
640  }
```

Figure 81.- *nextVideoItems() method*

```
Future<List<List<VideoItem>>> nextVideoItemsNested(List<VideoItem> videoItems, List<String> careerItems) async{
  try {
    String language = await getCurrentLanguage();

    if(language == "arabisk" || language == "pashto" || language == "urdu" || language == "farsi" || language =
      return videosMappedInOtherLanguages(videoItems, careerItems);
    }
    else {
      List<List<VideoItem>> result = [];
      result.add([]);
      String currentVideoChapter = videoItems[0].title;
      int postion = 0;

      for (VideoItem video in videoItems) {
```

```
737           if (compareFirstDigits(currentVideoChapter, video.title)) {
738             result[postion].add(video);
739             result.add([]);
740             currentVideoChapter = video.title;
741             postion++;
742           }
743           else {
744             String title = video.title;
745             print("Adding this video: $title, this is position: $postion");
746             result[postion].add(video);
747           }
748         }
749
750         return result;
751       }
752     }
753     catch (e){
754       print('Error in nextVideoItemsNested: $e');
755       return [];
756     }
757   }
```

Figure 82.- *nextVideoItemsNested() illustrated*

```
Future<List<List<bool>>> getDataAboutUserHaveSeenVideos(List<String> careerItems) async {
  try {
    String language = await getCurrentLanguage();

    SharedPreferences prefs = await SharedPreferences.getInstance();

    if(language == "arabisk" || language == "pashto" || language == "urdu" || language == '
      return getDataAboutUserHaveSeenVideosOtherLanguages(prefs, careerItems);
    }
    else{
      List<String> itemResult = await groupAndTranslateWithoutHeadlinesCareer();
      List<List<bool>> result = [];
      result.add([]);
      String currentChapter = itemResult[0];
      int postion = 0;

      for (String item in itemResult) {
        // Update the mapping to the list according to chapters
        if (compareFirstDigits(item, currentChapter)) {
          result.add([]);
          currentChapter = item;
          postion++;
        }

        bool? checkValue = prefs.getBool(item);
        if (checkValue != null) {
          if (checkValue) {
            print("Video has been seen");
            result[postion].add(true);
          }
          else {
            print("Video has NOT been seen");
            result[postion].add(false);
          }
        }
        else {
          print("Video has NOT been seen");
          result[postion].add(false);
        }
      }

      print("This is the result: $result");
      return result;
    }
  }
  catch (e) {
    print('Error in getDataAboutUserHaveSeenVideos: $e');
    return [];
  }
}
```

Figure 83.- *getDataAboutUserHaveSeenVideos() method*

```
List<List<bool>> getDataAboutUserHaveSeenVideosOtherLanguages(SharedPreferences prefs, List<String> careerItems){
  try{
    List<List<bool>> returnList = [];
    int i = 0;

    for(String heading in careerItems){
      returnList.add([]);
      List<String> anIteration = getCareerModulesVideosTittle(i, "");

      for(String item in anIteration){
        bool? checkValue = prefs.getBool(item);
        if (checkValue != null) {
          if (checkValue) {
            print("Video has been seen");
            returnList[i].add(true);
          }
          else {
            print("Video has NOT been seen");
            returnList[i].add(false);
          }
        }
        else {
          print("Video has NOT been seen");
          returnList[i].add(false);
        }
      }

      i++;
    }

    return returnList;
  }
  catch(e){
    print('Error in getDataAboutUserHaveSeenVideosOtherLanguages: $e');
```

Figure 84.- *getDataAboutUserHaveSeenVideosOtherLanguages() method*

```dart
Future<List<bool>> getDataAboutUserHaveSeenVideosHealth(List<String> healthItemIndexList) async {
  try {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    List<bool> result = [];
    for (String item in healthItemIndexList) {
      bool? checkValue = prefs.getBool(item);
      if (checkValue != null) {
        if (checkValue) {
          print("Video has been seen");
          result.add(true);
        }
        else {
          print("Video has NOT been seen");
          result.add(false);
        }
      }
      else {
        print("Video has NOT been seen");
        result.add(false);
      }
    }
    return result;
  }
  catch (e) {
    print('Error in getDataAboutUserHaveSeenVideosHealth: $e');
    return [];
  }
}
```

Figure 85.- *getDataAboutUserHaveSeenVideosHealth() method*

Appendix F: Code Explanations

F1: Packages and Dependencies for Translations

To enable translation functionality, add the 'get' package dependency to the *pubspec.yaml* file, as demonstrated in *Figure 86*.

Figure 86.- *Adding 'get' dependency in pubspec.yaml file*

Moreover, to utilize the translation functionality, all files must include the *get.dart* file by importing the 'get' package, as depicted in *Figure 87*. This ensures that each Dart file requiring text translation incorporates the necessary code at the beginning.

Figure 87.- *'get.dart' package import code*

To integrate this widget, start by adding the necessary dependency in the *pubspec.yaml* file, as depicted in *Figure 88*. Subsequently, as illustrated in *Figure 89,* import the package for the *AutoSizeText* widget into all Dart files where the text responsivity feature will be utilized.

Figure 88.- *AutoSizeText widget dependency in pubspec.yaml file*

Figure 89.- *AutoSizeText package import code*

F2: Localization Implementations

Implementing localization or translation functionality in a Flutter application involves several steps. These steps include implementing the *getKeys* getter, which are elaborated below. However, enabling localization is a comprehensive process that requires additional steps, such as integrating the feature into the *main.dart* file, as will be explained later in the next chapter 'Enabling localization and language selection'. Also, adding the necessary

dependency and importing the package are essential part of the steps, but is quite intuitive and will therefore not be covered.

Enabling translation functionality begins with the creation of the *LocaleString.dart* class file, which extends the *Translations* class or mixin. The *Translations* class or mixin usually offers methods or properties for accessing translated strings according to a specified locale. This approach aids in organizing and managing translations effectively within the application.

In this context, *LocaleString* is a class that extends the *Translations* class from the 'get' package, commonly utilized for internationalization (i18n) and localization (l10n) purposes. The *Translations* class necessitates implementations for the 'keys' getter, returning a nested 'Map<String, Map<String, String>>'. In Dart, a getter is a specialized method granting access to an object's properties. It is defined using the 'get' keyword, followed by the property name, in this case 'keys', found within the maps. This getter is responsible for supplying the translation keys and their corresponding translations for each supported language.

In addition, the *getKeys* method (getter) contains maps that encompass all text elements to be displayed within the app when utilizing translation functionality. These internal maps consist of String keys, which share similar values across all maps returned by the *getKeys* method. However, the values within these internal maps differ from one another, each representing translations of the text into selected languages. Essentially, the values within the internal maps carry the same meaning but are translated into various languages, identified by the Locale in String format. This facilitates displaying text within the app in the desired language. Moreover, all values within the internal maps are written in their respective native alphabets, ensuring easy comprehension for app users.

F3: Enabling Localization and Language Selection

Language_page.dart File

Within the *language_page.dart* file, essential packages such as 'flutter/material.dart', 'shared_preferences/shared_preferences.dart', and 'get/get.dart' are imported, as demonstrated in *Figure 30*. These packages serve various purposes, including managing UI elements, storing preferences, and handling state.

The keys within the mappings of the 'locale' list, namely 'name' and 'locale', exclusively utilize the String variable type. The first values within these mappings pertain to the language names visibly displayed to users upon app launch, aligned with the value set for the 'name' key. The second value, an object of type Locale, facilitates language detection by the app. Consequently, the inner maps of the 'locale' list encompass two String keys:

'name', representing the displayed language name on the Language Page, and 'locale', expecting a value of type Locale to specify the app's language preference.

In addition, instances of the Locale type are designed to store two String values: the first representing a language abbreviation, and the second representing the associated country. These two strings within the Locale variables serve as keys for the maps returned by the *getKeys* method (getter) in the *LocaleString.dart* file. This mechanism allows the application to identify which map the *getKeys* method (getter) returns, thus enabling it to display text in the desired language.

The code snippet depicted in *Figure 90* illustrates the declaration of two variables: 'prefs' of type 'SharedPreferences' and 'language' of type 'Future'. The 'late' keyword signifies that these variables will be initialized later in the program's execution, ensuring they are not accessed before initialization.

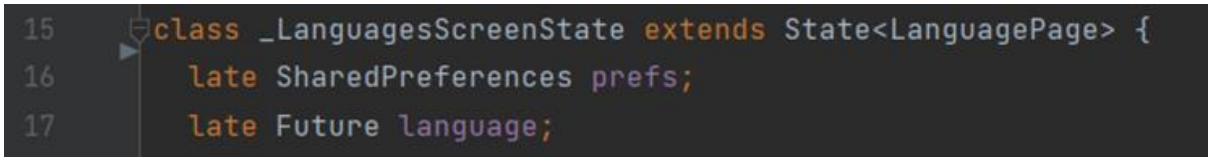
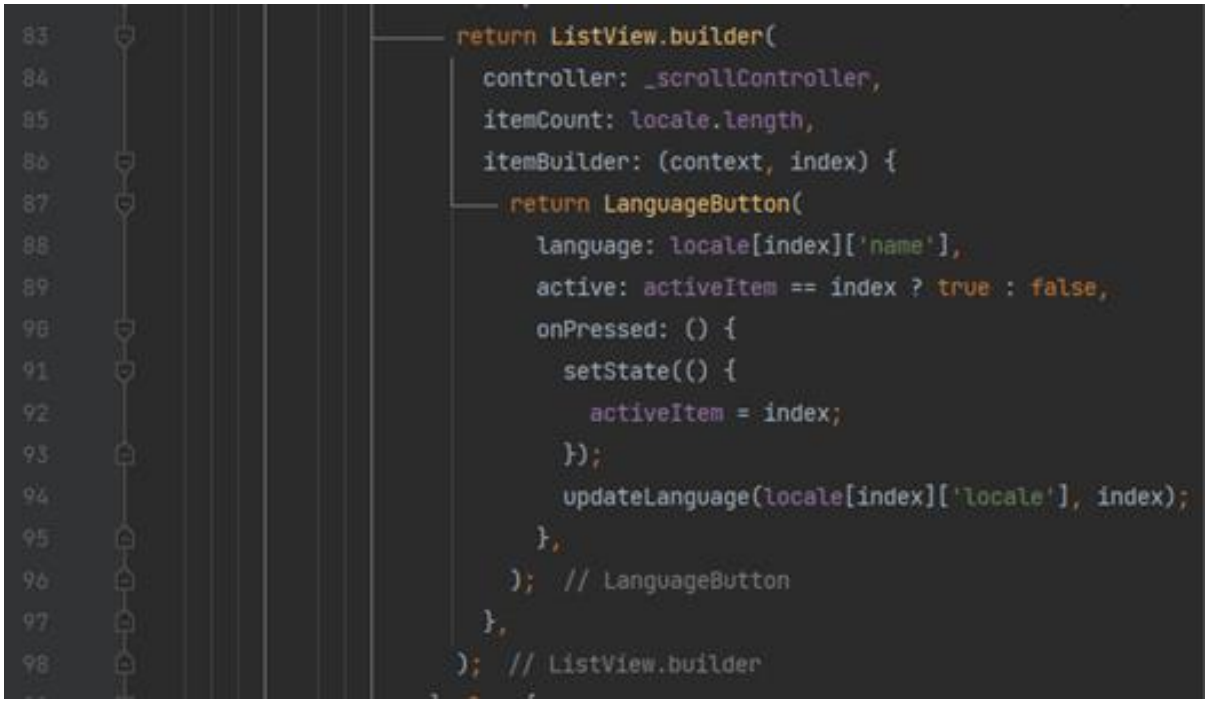


Figure 90.- *Shared Preferences declaration*

In Flutter, 'SharedPreferences' is a class provided by the 'shared_preferences' package, facilitating persistent storage and retrieval of primitive data types (such as bool, int, double, String, etc.) on the device. Meanwhile, a 'Future' in Dart and Flutter is employed for asynchronous programming, representing a value that will be available at some point in the future. The variable 'prefs' will primarily serve to retrieve the index for the selected language from the aforementioned 'locale' list, as detailed earlier. Conversely, the 'language' variable, upon initialization, will be assigned a reference to the asynchronous method *getSelectedLanguage()*, elaborated in *Figure 32* and further explained below.



```
return ListView.builder(
  controller: _scrollController,
  itemCount: locale.length,
  itemBuilder: (context, index) {
    return LanguageButton(
      language: locale[index]['name'],
      active: activeItem == index ? true : false,
      onPressed: () {
        setState(() {
          activeItem = index;
        });
        updateLanguage(locale[index]['locale'], index);
      },
    ); // LanguageButton
  },
); // ListView.builder
```

Figure 91.- *ListView builder in language_page.dart*

In *Figure* 91, a code snippet is presented, demonstrating a ListView of LanguageButtons as depicted in *Figure 31*. Each LanguageButton corresponds to a language in the 'locale' list, as illustrated in *Figure 93*. When an item is selected from the ListView of LanguageButtons, the chosen language is updated, triggering the *updateLanguage()* method, as demonstrated in *Figure* 91 and explained previously.

Language_page_nav.dart File

```dart
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:shared_preferences/shared_preferences.dart';

class LanguagePageNav extends StatefulWidget {

  LanguagePageNav({Key? key});

  static final List localeSet = [
    //   {'name': 'دری', 'locale': Locale('prs', 'AF')},
    //   {'name': 'فارسی', 'locale': Locale('fa', 'IR')},
    {'name': 'English', 'locale': Locale('en', 'UK')},
    {'name': 'Español', 'locale': Locale('es', 'ES')},
    {'name': 'Kiswahili', 'locale': Locale('sw', 'KE')},
    {'name': 'Kurmancî', 'locale': Locale('ku', 'TR')},
    {'name': 'Norsk', 'locale': Locale('nb', 'NO')},
    {'name': 'Soomaali', 'locale': Locale('so', 'SO')},
    {'name': 'Türkçe', 'locale': Locale('tr', 'TR')},
    {'name': 'عربی', 'locale': Locale('ar', 'AR')},
    {'name': 'پښتو', 'locale': Locale('ps', 'AF')},
    {'name': 'ትግርኛ', 'locale': Locale('ti', 'ET')},
    {'name': 'українська', 'locale': Locale('uk', 'UA')},
    {'name': 'اردو', 'locale': Locale('ur', 'PK')},
  ];
```

Figure 92.- *localeSet list in language_page_nav.dart file*

Figure 92 presents a code snippet extracted from the *language_page.dart* file. To enable translation functionality, the Dart file imports several packages, including 'flutter/material.dart' for UI elements, 'shared_preferences/shared_preferences.dart' for preference storage, and 'get/get.dart' for state management.

Figure 92 also displays the *localeSet* static list, which stores mappings between language names and corresponding 'Locale' objects accessible within the application. The *localeSet* is declared as 'final' and remains unchanged once initialized. Each mapping within *localeSet* consists of two String keys: 'name' and 'locale'. The 'name' key represents the language name displayed to users when selecting the app's language, while the 'locale' key holds a 'Locale' object, aiding language detection by the application. Furthermore, each inner map within *localeSet* contains two String keys: 'name', which defines the language's display name on the Language Page, and 'locale', storing the 'Locale' object that enables language identification for app access.

The *Locale* class is structured to hold two String values: the first indicates a language code, and the second denotes the associated country code. These strings, stored within Locale variables, act as identifiers for the maps retrieved by the *getKeys* method (getter) within the *LocaleString.dart* file. This system facilitates the app in identifying the appropriate map returned by the *getKeys* method (getter), thereby enabling it to present text in the chosen language.

```
47    Future<void> updateLanguage(Locale locale, int index) async {
48      Get.updateLocale(locale);
49      SharedPreferences prefs = await SharedPreferences.getInstance();
50      prefs.setInt('selectedLanguage', index); // Store selected language index
51    }
```

Figure 93.- *updateLanguage function in language_page_nav.dart file*

In *Figure 93*, we present the *updateLanguage()* method, which operates asynchronously and is of the 'Future' type. In Dart and Flutter, 'Future' is utilized for asynchronous programming, representing a value that will be available in the future. Similar to the *updateLanguage* method located in the *language_page.dart* file, this method requires two parameters: 'Locale' and 'int'. Both parameters leverage the static list *localeSet*, showcased in *Figure 92*, to fetch the value for the selected Locale object and its corresponding index in the *localeSet* list. This index corresponds to the selected Locale object (language) within the 'locale' list in the *language_page.dart* file. The objective of this method is to update the chosen language (Locale) in SharedPreferences and subsequently update the app's Locale (language) using *Get.updateLocale*.

Main.dart File

In *Figure 94*, the necessary packages are displayed, which should be imported into the *main.dart* file to enable translation (localization) functionality.

```
1  // main.dart
2  import 'package:flutter/material.dart';
3  import 'package:shared_preferences/shared_preferences.dart';
4  import 'package:get/get.dart';
5  import 'package:livsmestringsapp/services/LocaleString.dart';
6  import 'package:livsmestringsapp/pages/language_page_nav.dart';
```

Figure 94.- *Import packages in main.dart file*

The first parameter retrieves the integer index value for the selected language (Locale) from 'SharedPreferences'. Subsequently, the second parameter, also of type Locale, retrieves the corresponding language (Locale) from the *localeSet* list in *language_page_nav.dart* based on the previously fetched integer value. If a language (Locale) has been previously selected, its corresponding Locale value is retrieved. In cases

where no language (Locale) has been previously selected, such as during the initial app launch, the default language (Locale) is set to 'Norwegian' with the corresponding Locale '(‘nb’, ‘NO’).

To enable the translation functionality facilitated by the 'get/get.dart' package, the app should be encapsulated within a *GetMaterialApp*. This extension of the Flutter framework manages state and navigation seamlessly. The 'translations' property is configured with the *LocaleString*() method, indicating the application's utilization of translations provided by the *LocaleString* class.

The 'locale' property, responsible for determining the language (Locale) of the app, is initialized with the '_locale' variable, which is of type Locale. This variable is set to either the initial Locale value ('nb', 'NO') for Norwegian or to the user's preferred language (Locale) through the *initState()* and *updateLocale()* methods. These properties, 'locale' and 'translations', within the *main.dart* class, serve as key components for detecting the Locale (language) to be set for the app and accessing the corresponding translations (Strings/text) for each selected Locale (language).

The index integer representing the chosen language (Locale) is retrieved from both the 'locale' list in the *language_page.dart* file and the '*localeSet*' list in the *language_page_nav.dart* file using 'SharedPreferences'. This index is then passed as a constructor parameter to the *splash_screen.dart* file. This facilitates the detection of a previously set language (Locale) within the app, ensuring the correct language (Locale) is displayed to users upon app startup.

F4: Text Retrieval Algorithm

Firstly, as depicted in *Figure 37* and *Figure 44*, it is necessary to import the *get.dart* package to enable the utilization of translation (localization) functionality. This involves appending the '.tr' extension to strings, as explained earlier, to match the keys within the inner maps managed by the *getKeys* method (getter) in the *LocaleString.dart* file.

Career_data.dart File

getCareerModulesTitle()

The method is enclosed within a try/catch block to gracefully handle any potential errors during execution. Its primary function involves filtering the 'CareerItems' list to extract strings beginning with a single digit. This initial digit within each string corresponds to one of five module numbers on the career pages. Upon successful execution, the filtered list of strings, representing module titles, is passed to either the *groupModulesTittles()* or *groupAndTranslateModulesTittles()* method, depending on the content of the input string.

The former sorts the list, while the latter sorts and translates it. Additionally, the method is equipped to handle any encountered errors, either by printing a concise error message or by returning an empty list.

groupAndTranslateModulesTittles()

This method is enclosed within a try/catch block to handle potential errors during execution. Initially, the method utilizes an inner helper function employing regular expressions to filter each string in the list and extract single-digit numerical values from the beginning of each string. The first character, representing a digit within each string, corresponds to one of the five module numbers on the career pages. If no numerical value is found, '0' is returned as the default value. Subsequently, a map named 'groupedItems' is initialized, with numerical values as keys and lists containing strings starting with the same numerical value as values. Following this, a list of strings named 'result' is initialized to store the sorted strings with the '.tr' extension appended. Finally, the method handles any potential errors that may arise during execution, either by printing a concise error message or by returning an empty list.

getCareerModulesVideosTittles()

To ensure smooth execution of the code (method), a try/catch block is implemented to handle potential errors. Following this, a list of strings named 'result' is initialized to store values to be returned later by the method. Additionally, an integer variable is initialized to facilitate the iteration through indices in the list(s) on the career pages, aligning with the titles of career modules. Since titles for modules on the career pages begin with 1, while the first index in lists starts with 0, adjustments are made by incrementing the indices in the list(s) on the career pages by 1 to fetch the correct career module title.

Subsequently, the method iterates through the 'CareerItems' list to retrieve strings (titles) corresponding to the selected module. The identification of video titles related to each module follows a similar numerical pattern, commencing with the module's title and followed by a period ('.'). The resulting list is then passed as input to the method *groupAndTranslateModulesVideosTittle()*.

Finally, the method incorporates error handling mechanisms to address any potential errors during execution. Depending on the scenario, it either prints a concise error message or returns an empty list.

groupAndTranslateSubModulesOrVideosTittle()

The method begins by enclosing its operations within a try/catch block to handle errors effectively and provide informative error messages when necessary. Within this method, an inner helper function called *extractNumber()* employs a custom regular

expression tailored to the specific use case, effectively filtering input strings to extract single numerical values located at the beginning of each string. These numerical values represent module numbers on the career pages, totaling five in all. If no numerical value is found, the function defaults to returning '0'.

Following this, a map named 'groupedItems' is initialized, with numerical values serving as keys and lists containing strings beginning with those numerical values as values. Subsequently, the list 'keys' is employed to store and sort the extracted numerical values, which are then assigned as keys within the 'groupedItems' map.

For each key in the sorted 'keys' list, corresponding strings are retrieved from the 'groupedItems' map and sorted into a new list of strings called 'group'. These sorted strings, representing module video titles, are then stored in a pre-initialized list named 'result', optionally translated based on the input string content. Finally, the method returns the 'result' list. To ensure robust error handling, the method incorporates additional try/catch blocks to manage any potential errors that may arise during execution, providing concise error messages or returning an empty list as needed.

getVideoPlayerCareerAppBarTittle()

Initially, a try/catch block is implemented to handle any potential errors during code execution. Subsequently, two string variables are initialized: 'result', which will later store the method's return value, and 'searchIndex', designed to extract the first digit from the input string. The first digit in each string denotes the module number on the career pages, totaling five modules. Next, a 'for-loop' iterates through the strings in the 'CareerItems' list, utilizing an 'if-statement' to identify the correct string representing the module title for the provided video title.

This is achieved by comparing the translated first digit from the list with the first digit from the input string, representing the video title. If no errors occur during code execution and the targeted string representing the module title is found, it is translated and returned by the method. Finally, the method handles potential errors by either printing a brief error message or returning an empty string, depending on the circumstance.

Health_data.dart File

Notably, the *get.dart* package must be imported to facilitate the utilization of translation (localization) functionality by appending the '.tr' extension to strings, as elaborated earlier. These strings, which also serve as keys within the inner maps managed by the *getKeys* method (getter) in the *LocaleString.dart* file mentioned previously, are instrumental in enabling localization.

Additionally, the *career_data.dart* file should be imported into *health_data.dart* to leverage methods implemented there, including sorting algorithms. Furthermore, *Figure 44* illustrates the 'HealthItems' list of strings, utilized to store keys within inner maps managed by the *getKeys* method in the *LocaleString.dart* file. These keys, integral to the inner map in *LocaleString.dart*, are utilized across various methods in the Dart file to retrieve accurately translated titles for different health pages, facilitated by the '.tr' extension.

getHealthModulesTittles()

An 'if-statement' is utilized to filter the string contents and identify those that match the specified criteria. Any matching strings are then appended to the 'result' list, which was previously initialized. This list is later passed as a parameter to the *groupAndTranslateModulesTittles()* method for sorting and translation. This method is already implemented in the *career_data.dart* file, provided no failures occur during execution. In the event of any failure during method execution, appropriate handling is undertaken, which may involve printing an error message or returning an empty list, depending on the circumstance.

getHealthSubModuleTittles()

To begin, a try/catch block is implemented to manage any potential failures during code execution. Subsequently, a list of strings named 'result' is initialized, intended to be returned by the method. Additionally, an integer variable is initialized to facilitate the incrementation of input indices, which correspond to the titles of health sub-modules listed on the health page(s).

Titles for modules on the health pages commence with the numeral 7, while the first index on lists starts with 0. Consequently, to obtain the correct career module title, indices within the list(s) on the health pages are incremented by 7. Following this adjustment, the method iterates through the 'HealthItems' list, retrieving strings (titles) pertaining to the selected module. Each video title associated with a module shares a numerical prefix identical to the module's title, followed by a period ('.') and an underscore ('_') as the fourth character in the string.

Afterward, the input string content determines whether the list is sent to the sorting method *groupAndTranslateSubModulesOrVideosTittle()* for translation, or if it should remain untranslated, it is passed to the *groupSubModulesTittle()* sorting method. The *groupAndTranslateSubModulesOrVideosTittle()* method, responsible for both sorting and translating the strings within the list, is already implemented in the *career_data.dart* file, which is imported into the Dart file as elaborated previously. Additionally, the method handles

potential errors during execution by either printing a brief error message or returning an empty list.

getHealthSubModulesVideosTittle()

Initially, the method is encapsulated within a try/catch block to gracefully handle any potential failures during its execution. Subsequently, a list of strings, named 'result', is initialized to hold the eventual return values of the method. Additionally, two strings are assigned with values to represent the indices for module and sub-module titles, incremented by 7 and 1, respectively. It is worth noting that indices in lists begin with 0 instead of 1. Consequently, the numbering for health modules starts from 7, while sub-modules, indicated by numbers following the first period ('.'), commence from 1. Precisely identifying these numbers aids in retrieving video titles corresponding to each sub-module.

Proceeding, the method iterates through the 'HealthItems' list, fetching strings (titles) relevant to the selected sub-module. Identification occurs through the index of the module's title and the index of the sub-module's title. Video titles pertaining to each module are distinguished by their initial two numerical values, separated by a period ('.'), matching the starting numerical value of the sub-module's title followed by a period ('.').

The list will ultimately serve as input for the *groupAndTranslateSubModulesOrVideosTittle()* sorting and translation method, which is already implemented in the *career_data.dart* file as described earlier. Additionally, the method is designed to handle any potential errors that may arise during execution by either printing a concise error message or returning an empty list, depending on the circumstances.

getSubModuleIndexAndVideosTittle()

The method employs a try/catch block to handle potential errors that may arise during code execution. Additionally, it initializes a list of strings called 'result,' which will later be returned by the method. An integer variable named 'subModuleIndex' is also initialized to store the index corresponding to the sub-module title. Subsequently, a 'for-loop' iterates through the list of sub-module titles to identify the index of the title clicked by the user. If a matching title is found, its index is assigned to the 'subModuleIndex' variable.

In addition, two variables are assigned values corresponding to the indices used for numbering modules and sub-modules. To adjust for the zero-based indexing in lists, module numbering begins at 7, while sub-module numbering starts at 1, following the module number. Accurate identification of these indices facilitates retrieval of video titles associated with each sub-module. The process involves iterating through the 'HealthItems' list and retrieving titles related to the selected sub-module. These titles are identified by matching

the initial two numerical values separated by a period, which mirrors the starting numerical value of the sub-module's title.

The list is passed as input to the *groupAndTranslateSubModulesOrVideosTittle()* sorting and translation method, which is already implemented in the *career_data.dart* file and detailed earlier. Finally, this method manages potential errors that may arise during its execution, either by printing a concise error message or returning an empty list, depending on the circumstances.

getVideoPlayerHealthAppBarTittle()

In preparation for method return, a string variable named 'result' is initialized to hold the eventual output. Additionally, two other string variables, 'searchIndex1' and 'searchIndex2', are initialized to facilitate the extraction of two digits from the input string. These digits correspond to the module and sub-module numbers on the health pages, respectively.

Subsequently, a 'for-loop' iterates through the strings in the 'HealthItems' list. Utilizing an 'if-statement', the loop identifies the correct string within the list that represents the sub-module title for the inserted video title. This is achieved by validating if the string contains only two digits, and if those digits match the first two digits of the input string, which denote the video title. If the code executes without errors and the targeted sub-module title string is found, it is translated and returned by the method. Finally, the method handles potential errors during execution by either printing a concise error message or returning an empty string, depending on the circumstance.