



STANFORD
MUSSALLEM CENTER FOR
BIODESIGN

FernUniversität in Hagen
Faculty of Mathematics and Computer Science

In Cooperation with

Stanford University
Stanford Mussallem Center for Biodesign

Master thesis
in the degree program Practical Computer Science M.Sc.

**Developing a Modular and Reusable mHealth Framework:
A Case Study on Heart Failure Management with
ENGAGE-HF Android App**

by

Kilian Schneider

Matriculation number: 4159292

Date of submission: October 26, 2024

First reviewer and supervisor: Dr. Carina Heßeling

Second assessor: Dr. Paul Schmiedmayer

Acknowledgements

I would like to express my gratitude to those who supported me during my Master's thesis.

Special thanks to Dr. Carina Heßeling for her excellent supervision and valuable advice, and to Dr. Paul Schmiedmayer for his professional expertise and continuous support, which greatly contributed to this thesis. I also thank Dr. Vishnu Ravi for his assistance with Fast Healthcare Interoperability Resources (FHIR) and relevant research, and Eldi Cano for his code reviews, discussions, and optimization suggestions.

Thanks to Dr. Alex Sandhu for his help with ENGAGE-HF case study questions, Nick Riedman for developing the iOS app, Paul Kraft for the ENGAGE-HF Firestore backend, Arek Bachorski for the administration web dashboard, and Dr. Oliver Aalami all of whom significantly enhanced the system's functionality.

Finally, I extend my heartfelt thanks to my parents and girlfriend for their unwavering support.

Thank you to everyone who accompanied me on this journey.

Abstract

The Spezi framework is a modular, reusable platform designed to advance Mobile Health (mHealth) technologies, particularly for managing chronic conditions such as heart failure. While an iOS version of the framework already exists, this work focuses on creating the Android version, making these elements more accessible and enabling them to work for a larger group of people, thereby enhancing cross-platform compatibility. Built with Kotlin and integrated with a replaceable Google Cloud Firebase default implementation, it offers an open-source solution that supports both Android and iOS. Key features include Bluetooth connectivity to medical devices, health monitoring, user-friendly interfaces, Fast Healthcare Interoperability Resources (FHIR) integration, and account management, all aimed at enhancing patient engagement and improving healthcare provider efficiency.

The ENGAGE-HF app serves as a case study, validating the framework's scalability and adaptability in heart failure management. Incorporating feedback from both patients and healthcare providers, ENGAGE-HF demonstrates the framework's potential to deliver personalized care and interactive health management tools. A multi-center clinical study will evaluate the app's effectiveness in real-world settings, highlighting the modular design's transformative impact on mHealth solutions.

This thesis delves into the design and development of the Spezi Android framework, focusing on how its architectural decisions and modular and reusable design facilitated the development of the ENGAGE-HF case study app.

Zusammenfassung

Das Spezi-Framework ist eine modulare und wiederverwendbare Plattform, die speziell zur Förderung von Mobile Health (mHealth)-Technologien entwickelt wurde, mit dem Fokus im Bereich des Managements chronischer Erkrankungen wie Herzinsuffizienz. Während eine Version des Frameworks für iOS bereits verfügbar ist, um diese Elemente zugänglicher zu machen und sie für einen größeren Personenkreis nutzbar zu machen, wodurch die plattformübergreifende Kompatibilität verbessert wird. Das Framework wurde in Kotlin implementiert, bietet eine austauschbare Firebase Default Integration und ist somit ein Open Source Framework, das sowohl Android als auch iOS unterstützt. Zu den wichtigsten Funktionen gehören Bluetooth-Konnektivität mit verschiedenen medizinischen Geräten, Gesundheitsüberwachung, einfach zu bedienende Schnittstellen und eine Fast Healthcare Interoperability Resources (FHIR)-Integration sowie Accountverwaltung neben anderen Funktionen, die alle darauf ausgerichtet sind, das Engagement der zu behandelnden Personen und die Effizienz der Gesundheitsdienstleister zu erhöhen.

Die ENGAGE-HF-App dient als Fallstudie, die die Skalierbarkeit und Anpassungsfähigkeit des Frameworks für das Management von Herzinsuffizienz bestätigt. Unter Einbeziehung des Feedbacks von zu behandelnden Personen und Gesundheitsdienstleistern demonstriert ENGAGE-HF das Potenzial des Rahmens für eine personalisierte Pflege und interaktive Gesundheitsmanagement-Tools. Im Rahmen einer multizentrischen klinischen Studie wird die Wirksamkeit der App in realen Umgebungen evaluiert, um die transformative Wirkung des modularen Designs auf mHealth-Lösungen zu unterstreichen.

Diese Arbeit befasst sich mit dem Design und der Entwicklung des Spezi Android-Frameworks und konzentriert sich darauf, wie die architektonischen Entscheidungen und das modulare und wiederverwendbare Design die Entwicklung der ENGAGE-HF-Fallstudienapp erleichtert haben.

Contents

List of Figures	III
List of Tables	V
1. Introduction	1
2. Theoretical Foundations and State of the Art	5
3. Design and development of the framework	15
4. Technical details on implementation	21
5. Case Study: ENGAGE-HF for Heart Failure Management	67
6. Evaluation	87
7. Discussion and Future Work	93
8. Summary	95
List of abbreviations	99
A. Appendix	101
Bibliography	V

List of Figures

4.1.	Dokka example documentation.	43
4.2.	Login and Register Screen	44
4.3.	Contact Screen Example.	49
4.4.	Consent Screen.	52
4.5.	Invitation Code View.	53
4.6.	Onboarding Screen.	55
4.7.	Sequential Onboarding View.	56
4.8.	Questionnaire.	61
4.9.	FHIR Data Capture Functionality.	62
5.1.	Education Screen Overview.	70
5.2.	Education Screen Single Video.	71
5.3.	Heart Health Screen.	73
5.4.	Add Data Bottom Sheets.	74
5.5.	ENGAGE-HF app’s Messages screen and Health Summary PDF for patient updates and health reports.	78
5.6.	Notification Settings Screen.	80
5.7.	Home Screen.	81
5.8.	Medication Screen.	83
6.1.	Usage of Spezi framework modules by the case study app.	91

List of Tables

4.1. Health Connect Records and Corresponding FHIR Observation Categories 50

1. Introduction

The digitalization of healthcare is progressing and bringing forth innovative technologies that have the potential to profoundly change the care of chronic diseases. mHealth-Technologies that actively involve patients in the management of their health are at the center of this development. We focus on the development of the Spezi Android framework, as demonstrated by the creation of the ENGAGE-HF Android app for heart failure management, which highlights the technological and therapeutic potential of these innovations. This chapter introduces the topic, outlines the motivation behind the project and defines objective of this thesis.

Background and Motivation

The growing prevalence of chronic diseases and the role of technology in their management are key issues in the healthcare sector. Chronic diseases such as diabetes and heart failure place a significant burden on healthcare systems worldwide. As the demand for effective management of chronic diseases continues to rise, leveraging technology, particularly mHealth applications, becomes increasingly important [1]. These applications offer the potential to transform patient care by enabling continuous monitoring, self-management, and better clinical decision-making through real-time data. However, while many mHealth solutions have been developed to address these needs, early frameworks such as CardinalKit revealed significant limitations¹.

The CardinalKit framework was initially developed as a template application designed to simplify and accelerate the development of mHealth solutions¹. It helped developers by offering pre-integrated tools such as healthcare surveys using FHIR, task scheduling, and health data collection via Health Connect. Despite these advancements, CardinalKit had

¹<https://cardinalkit.org/>

1. Introduction

limitations, particularly around its inflexible architecture, which required significant customization to meet specific project needs¹. Moreover, the lack of modern standards integration limited its ability to support the growing demands of interoperable healthcare systems.

To address these challenges, the Spezi framework for iOS was introduced as an open-source, modular alternative that could support the rapid development of mHealth applications². Spezi for iOS shifted away from a rigid template and adopted a modular, Lego-like approach, allowing developers to integrate and extend components based on their project's unique needs². Spezi emphasized the use of standardized healthcare data and Bluetooth connectivity to medical devices, significantly improving real-time data transmission and patient monitoring capabilities².

Recognizing the significant global market share of Android, development efforts in this thesis focus on bringing the Spezi framework to the Android platform. The Spezi framework for Android builds on the lessons learned from the iOS implementation but is tailored to the unique requirements and ecosystem of Android development. The framework is designed to ensure modularity, scalability, and flexibility, allowing for the seamless integration of Bluetooth medical devices and real-time health monitoring. Given the market share of iOS being around 28.46% in March 2024 [2], the Android development marks a crucial step in ensuring that the Spezi framework can reach a broader audience and support a more diverse range of mHealth applications [2].

Of particular note is the direct Bluetooth connection to medical devices, which enables immediate data transmission and processing. This underlines the potential of technology to fundamentally change and improve the management of chronic diseases such as heart failure [1].

Mobile health apps have proven to be an effective solution for managing the burden of chronic disease. These apps promote self-monitoring and health management, provide digital educational resources and feedback, and enable faster access to medical providers [3]. By reducing hospital stays and promoting optimal health outcomes, mobile health apps play a critical role in enhancing the quality of care and improving patient outcomes [3].

²<https://spezi.stanford.edu/framework>

Research Objective

The objective of this master thesis is the development and demonstration of the Spezi Android framework, a modular, scalable and reusable platform for Android for the creation of mHealth-applications. This framework is demonstrated using the ENGAGE-HF Android app as an example and is intended to improve the quality of medical care and optimize patient outcomes.

A key goal is to increase patient engagement by integrating direct Bluetooth connectivity with medical devices. This enables seamless and accurate transmission of health data, benefiting both patients and medical staff. The application aims to improve the interaction between patients and caregivers and thus increase the efficiency of medical care.

Another key objective is to develop a flexible and modular system architecture that maintains a high level of user-friendliness. This includes the implementation of robust authentication mechanisms and customizable Bluetooth connectivity. But also, it includes the ability to adapt to other chronic diseases. Additionally, it involves good documentation. The intuitive user interface is designed to support both medical staff and patients by enabling easy and effective handling.

The demonstration of the framework in a multi-center study will not only prove its direct effectiveness in the treatment of heart failure, but also demonstrate its versatility and universal applicability to chronic diseases. These studies aim to continuously incorporate user feedback and clinical trial results into the improvement and expansion of the system.

Ultimately, this work lays the foundation for a new era of digital health solutions by introducing a high degree of modularity, scalability and user-centricity. This should lead to improved patient care and monitoring and revolutionize the interaction between patients and medical staff.

2. Theoretical Foundations and State of the Art

The rapid development of the mHealth-technologies have the potential to fundamentally transform the healthcare system. This chapter focuses on the basic concepts and the current state of the art in the field of mHealth. Both the historical development and current trends are examined. In particular, the importance of modular and reusable frameworks that can be quickly adapted to new requirements and findings is emphasized.

Overview of mHealth and its Evolution

The field mHealth is becoming increasingly important in patient care and health management, especially for chronic diseases such as heart failure [4] [5]. Efficient, scalable and easy-to-use mHealth solutions can significantly improve patient outcomes, patient education and self-management, and provide healthcare providers with critical data for monitoring and decision-making [4] [5].

Historical Development of mHealth

The development of mHealth-applications began with simple tracking tools that recorded basic health parameters such as steps and calories burned. However, with the proliferation of mobile devices and technological advances in sensors, data processing and connectivity, mHealth-applications have quickly evolved into integrated health management systems [4]. With the introduction of smartphones and tablets, more comprehensive features could be implemented. These devices enable continuous monitoring and management of health data, providing real-time insights and enabling an immediate response to health changes [4] [5]. Where early mHealth-solutions focused mainly on monitoring basic vital signs.

Key Technologies and Platforms

The rise of mHealth was supported by several key technologies:

1. **Internet of Things (IoT):** By connecting medical devices to the internet, data can be collected and analyzed in real time. This enables precise and continuous monitoring of health parameters, which is particularly beneficial for the management of chronic diseases [6].
2. **Cloud Computing:** The use of cloud services allows the storage and processing of large amounts of data generated by mHealth-applications. This facilitates access to health data and collaboration between patients and healthcare providers [7].
3. **Artificial Intelligence (AI):** AI-technologies are increasingly being integrated into mHealth-applications to recognize patterns in health data and provide personalized recommendations. AI can also help in the diagnosis and prognosis of diseases by analyzing large amounts of data and providing relevant insights [8].

Challenges and Potentials of mHealth

Although mHealth-technologies are promising, there are also challenges to overcome. Existing mHealth-platforms often lack the flexibility to adapt to different diseases or user feedback without extensive new development [4]. Many currently available solutions are specifically tailored to a single disease or specific use case and offer little scope for customization or expansion [4] [5].

Studies such as the systematic review and meta-analysis of Free, Phillips, Watson, *et al.* [9] have highlighted the potential of mHealth technologies to improve healthcare services. These studies show that mHealth-interventions can improve health processes and patient outcomes, which forms the basis for the assumption that frameworks such as can have a significant impact [10].

Relevance of the Spezi Framework

The Spezi framework is uniquely suited to address the demands of the ENGAGE-HF project, leveraging insights from a successful pilot at Stan-

ford University. While many existing healthcare platforms and frameworks, such as CardinalKit¹, offer interoperability and open-source solutions to streamline digital health app development, there remain challenges in scaling these solutions to chronic disease management, particularly heart failure. Key issues in current systems include a lack of flexibility, scalability, and insufficient integration with modern healthcare technologies, such as direct Bluetooth connectivity for medical devices, crucial for real-time data collection and patient monitoring [11]. As highlighted by recent studies, including CardinalKit's focus on modular and extensible components, many mHealth platforms still fall short in these areas [4].

Spezi framework fills this gap by offering a modular, open-source platform, specifically designed for Android, that integrates modern technologies like Bluetooth-enabled medical devices and comprehensive patient feedback. This level of integration is crucial for improving patient engagement, self-management, and health outcomes, especially in the management of heart failure. The framework's open-source nature allows for rapid iteration and customization, directly aligning with the patient-centered and scalable approach that modern chronic disease management demands. Furthermore, its flexibility mirrors CardinalKit's objective of lowering barriers to digital health innovation by offering a foundation for developing robust, compliant, and interoperable healthcare applications at reduced cost and development time [11].

Future Trends in mHealth

In the context of discussing future trends in mHealth, integrating AI and interoperability is crucial for enhancing patient care. The Spezi framework is well-positioned to leverage these trends due to its modular and scalable architecture [12]. The framework's adaptability can accommodate advancements like AI-driven personalized care and seamless interoperability between different healthcare systems, ensuring that patient data is securely shared and utilized effectively across platforms.

AI can be integrated into mHealth applications to provide personalized treatment recommendations by analyzing patient data in real-time, helping to improve diagnostic accuracy and care efficiency. AI algorithms can also assist in predicting potential health issues before they become critical,

¹<https://cardinalkit.org/>

allowing for timely interventions.

Additionally, interoperability will be essential in future mHealth developments, ensuring that various healthcare systems and devices can communicate and exchange data smoothly [13]. The Spezi framework, which supports integration with medical devices via Bluetooth and cloud-based platforms like Google Firebase, is designed with these needs in mind.

This patient-centric approach can enable a more connected, intelligent healthcare system that reacts dynamically to patients' evolving health conditions, improving overall patient outcomes while reducing the burden on healthcare providers [13]. Thus, the Spezi framework's design not only supports the current demands of healthcare but is also primed to adapt to these future technological advancements.

Analysis of Existing Solutions and Their Limitations

The increasing importance of mHealth-technologies in healthcare, especially in the management of chronic diseases such as heart failure, has led to a variety of existing frameworks and specific applications. These solutions vary widely in their functionality, ease of use and integration into existing healthcare systems. This chapter will introduce some of these existing mHealth frameworks and applications and highlight their limitations, while also looking at solutions for other chronic diseases.

Existing mHealth-solutions for heart failure

1. **Heart Failure Health Storylines:** This app offers patients the opportunity to track symptoms, medication and vital signs. It enables the creation of personalized health reports and the exchange of information with healthcare providers. Despite its comprehensive functions, the app is often criticized for its complexity and lack of user-friendliness, which is a particular problem for older adults² [14].
2. **Medisafe:** The Medisafe app is a popular tool designed to assist users with medication management. It offers features such as medication reminders, refill alerts, and the ability to share health data with others. Users appreciate its user-friendly interface and the clear,

²<https://hfsa.org/>

visual schedule of medications, which makes the app straightforward and effective for managing prescriptions. Despite its simplicity and ease of use, some users find the app's functionality somewhat limited if they need to monitor a broader range of health metrics beyond medication adherence [15].

3. **MyTherapy**: Another popular app adapted to the needs of patients with chronic conditions. It offers medication reminders, symptom tracking and reports. However, it lacks integration with medical devices and the ability to process real-time data, which limits its usefulness in the acute management of heart failure³.

Existing mHealth solutions for other chronic diseases

1. **Glucose Buddy**: This app is aimed at diabetics and offers functions for tracking blood glucose levels, food intake and physical activity. Despite the usefulness of these functions, there is often criticism of the lack of integration with other health services and insufficient adaptability to individual needs⁴.
2. **AsthmaMD**: This app helps asthma patients track their symptoms and triggers. It makes it possible to share data with doctors and create asthma action plans. However, it lacks advanced features such as sensor data integration and customization to personal health data⁵ [16].
3. **MyFitnessPal**: This app is designed to support individuals with obesity in managing their weight. It offers features such as food logging, exercise tracking, and progress monitoring. Users benefit from a comprehensive food database and community support, yet the app is often criticized for its reliance on user input for accurate data and limited integration with personalized health goals [17].

³<https://www.mytherapyapp.com/de>

⁴<https://www.glucosebuddy.com/>

⁵<https://www.asthmamd.org/>

Limits of existing solutions

Despite the large number of available mHealth applications, there are several limitations that influence their effectiveness and user acceptance:

1. **Lack of integration with healthcare systems:** Many mHealth apps are isolated solutions that are not seamlessly integrated into existing Electronic Health Records (EHRs) or Hospital Information Systems (HISs). This complicates data exchange between patients and healthcare providers and can lead to fragmentation of patient care [18].
2. **Insufficient personalization:** Existing solutions often offer standardized features that do not adequately address patient's individual needs and preferences. The lack of adaptability can reduce the effectiveness of self-management strategies [19].
3. **Limited user-friendliness and accessibility:** Older adults in particular, who represent an important target group for the management of chronic diseases, often have difficulties using complex apps. Studies show that factors such as technological barriers and low user-friendliness significantly influence the adoption rate of mHealth apps in this population group [20].
4. **Lack of modularity and scalability:** Many mHealth applications are not designed with modularity in mind, making it challenging to adapt to new medical findings or technological advancements [21]. This rigidity can stifle innovation and slow the implementation of necessary improvements [22]. Argue that successful mHealth apps must be adaptable to provide value to both healthcare providers and patients, which is often not the case with current offerings [23]. The inability to quickly implement updates or enhancements can lead to obsolescence in a rapidly evolving healthcare landscape [23].

Existing mHealth frameworks

1. **Open mHealth:** Open mHealth is an open source framework that aims to standardize various health data formats and facilitate the integration of this data into mHealth applications. It provides a collection

of APIs and data models that help developers create interoperable healthcare applications. Despite its comprehensive features, the framework is often criticized for its complexity and high implementation effort, which can limit the adoption rate [24].

2. **ResearchKit and CareKit:** Developed by Apple, these frameworks play crucial roles in advancing mobile health applications⁶. ResearchKit is a framework that aids researchers in conducting mobile health studies by enabling the creation of study apps that can recruit participants, collect data, and analyze it⁶. Despite its power, ResearchKit is limited to iOS devices, which restricts the reach of potential studies⁶. On the other hand, CareKit is another Apple framework designed to help developers build apps that empower patients to manage their own health⁷. CareKit allows the creation of apps that can track symptoms, medication, and care plans while providing insights that can be shared with healthcare providers⁷. Like ResearchKit, CareKit is also limited to iOS, which constrains its accessibility to a broader audience⁷.
3. **MyPHD:** Developed as a pre-packaged application, MyPHD offers a streamlined approach for studies that require minimal customization [25]. It is particularly designed for quick deployment in research environments where extensive development work is not feasible [25]. MyPHD provides basic mHealth functionalities like participant engagement tools and standard health monitoring [25]. However, its simplicity comes with limitations, including reduced flexibility and the absence of advanced features such as deep integration with medical devices [25]. This makes it suitable for studies with straightforward needs but less ideal for projects requiring complex, customizable solutions [25].
4. **Sana Mobile:** Sana Mobile is another open source framework developed specifically for healthcare in low-resource environments. It provides a platform for creating mobile healthcare applications that connect patients and healthcare providers. However, one drawback is its limited support for modern technologies and standards, which restricts its use in more advanced healthcare systems [26].

⁶<https://researchkit.org/>

⁷<https://www.researchandcare.org/carekit/>

Limits of the existing frameworks

Despite the large number of available mHealth frameworks, there are several limitations that influence their effectiveness and user acceptance:

1. **Lack of user-friendliness and high cancellation rates:** User experience is critical in the adoption of mHealth applications. Research indicates that up to 30% of users abandon mHealth apps within the first week, primarily due to poor usability and unclear interfaces [27]. Mustafa, Ali, Dhillon, *et al.* highlights that lack of interest or declining motivation is a significant factor, with 31.6% of participants in their study citing this as a reason for abandonment [27]. Furthermore, the complexity of app interfaces contributes to user disengagement, with reports indicating that 35% of users struggle with mHealth apps due to non-intuitive designs [28]. This issue is exacerbated in underserved regions where technical barriers are more pronounced [29].
2. **High costs and time-consuming development processes:** The development of regulatory-compliant mHealth frameworks is resource-intensive. Such financial burdens make it prohibitive for smaller healthcare providers or projects in low-resource settings to develop effective mHealth solutions [30]. The complexity of ensuring compliance with various regulations further complicates the development process, as highlighted by Shiferaw, Workneh, Yirgu, *et al.*, who discusses the regulatory landscape surrounding mHealth applications.
3. **Lack of modularity and scalability:** Modularity is essential for the adaptability of mHealth frameworks to evolving clinical needs [22]. Only few of mHealth frameworks provide sufficient modularity, which limits their ability to incorporate the latest medical findings or technologies [23]. This lack of flexibility stifles innovation and hinders the continuous improvement of mHealth applications, making it challenging to keep pace with advancements in healthcare [23].
4. **Complexity of the applications:** The complexity of many mHealth applications further complicates user engagement. A mixed-methods study by Liew, Zhang, See, *et al.* revealed that usability challenges significantly impact user satisfaction and engagement with mHealth apps [28]. Users often find themselves overwhelmed by complex

interfaces, which leads to frustration and disengagement [31]. This complexity is particularly detrimental in low-resource settings, where users may have limited technical skills [29].

5. **Insufficient support related to healthcare staff:** There is often a lack of sufficient training provided to healthcare staff and insufficient support from them to effectively implement and utilise mHealth applications [30]. This can affect the integration into everyday clinical practice and acceptance by healthcare providers [32].
6. **Limited compatibility and interoperability:** Many mHealth solutions are not sufficiently compatible with existing healthcare systems, which hinders their widespread use [23]. Integration into EHRs and other centralised healthcare systems is particularly affected [23]. One specific problem is the limited use of Bluetooth devices, which often cannot be seamlessly integrated into mHealth apps, making it difficult to collect and transfer health data.

Summary

The analysis of existing mHealth frameworks shows that, despite their potential, there are still considerable gaps. These relate in particular to integration into existing healthcare systems, the personalization of functions, user-friendliness and the modularity of the frameworks. These findings underline the need for a new generation of mHealth frameworks that can address these challenges and thus increase effectiveness and user acceptance.

The Spezi framework for Android developed in this thesis aims to close these gaps by providing a modular, scalable and user-friendly platform that can be customized to meet the specific needs of patients and healthcare providers.

3. Design and development of the framework

The development of a robust and flexible mHealth framework requires careful planning and execution to meet the diverse requirements of the healthcare sector. This chapter first describes the development approach for the modular framework. The focus here is on the principles of modularity and reusability in order to enable simple customization and expansion of the system.

The section on the technical details of the implementation presents the specific technologies, design patterns and architectural decisions that were used to realize the framework. This includes the integration of Bluetooth for the connection with medical devices, the use of a standardized module for the design, and the implementation of protocols for data storage and processing.

A central aim of this chapter is to emphasize the advantages of a modular architecture, which not only facilitates development and maintenance, but also enables rapid adaptation to new requirements and technologies. By detailing the technical implementation, this chapter provides valuable insights into the development of a modern mHealth framework that supports both developers and end users.

System Requirements

The framework aims to provide a reusable, scalable and modular basis for the rapid development of Android apps in the healthcare sector. This will be demonstrated with the help of the ENGAGE-HF case study app, which is used to support heart failure patients. The modules include an authentication module, Bluetooth connectivity module, data processing module and a UI/UX module with reusable Jetpack Compose components.

3. Design and development of the framework

The main users of this framework are software developers aiming to create or extend mobile health applications. This includes developers within the ENGAGE-HF project and potentially third parties who wish to use the framework for similar applications.

The framework is based on Android and uses Kotlin for app development and Google Cloud Firebase for backend services. It is designed for integration with health monitoring devices via Bluetooth.

Functional Requirements

FR1 Modularity: Design of an architecture that supports interchangeable components for disease-specific functionalities. As a result, the system can be flexibly adapted to advances in medical research and the development of new treatment methods without the need for a major overhaul. This adaptability promotes rapid implementation of new knowledge and technologies into clinical practice, improving the effectiveness and efficiency of patient care.

FR2 Reusability: Enables easy customization of the framework for various chronic diseases beyond heart failure.

FR3 Integration with health devices: Seamless connectivity with external scale and blood pressure monitor via Bluetooth.

FR4 Customizable patient education and medication management: Framework support for customizing content and features to the patient's specific needs and treatment plans.

FR5 Authentication: The framework is intended to enable user authentication.

FR6 Notification: The app issues personalized notifications based on specific criteria, such as medication changes, health data entries, and study-related reminders.

FR7 Visualization of health data: The framework is designed to enable visualization and monitoring of heart health data (weight, blood pressure, heart rate) through graphical and list-based representations that promote user engagement and understanding.

FR8 Symptom tracking: The framework is intended to enable the tracking of symptoms.

FR9 Provision of educational content: The framework should offer the expandability to provide educational material.

Non-Functional Requirements

Usability

NFR1 Accessibility and user-friendliness: A design that ensures ease of use for a wide range of patients, including those with limited technical knowledge.

Reliability

NFR2 Reliability: High operational stability and low downtime to ensure continuous availability of health monitoring and services.

Performance

NFR3.1 Scalability: Ability to support a growing user base and new functionalities without sacrificing performance. It must also be able to handle data processing tasks for up to 10,000 simultaneous users without significant delays to ensure a consistently satisfactory user experience.

NFR3.2 Performance: Fast response times and efficient data processing, even with high loads and data volumes, to ensure a satisfactory user experience. The application should have response times of less than 2 seconds under normal operating conditions and less than 3 seconds under peak load conditions with high data volumes.

Supportability

NFR4.1 Maintainability: Easy updating and maintenance of the application with clear documentation.

NFR4.2 Testability: The framework should be designed in such a way that it is easy to test, with a particular focus on achieving a test coverage of at least 70% across the entire code. This requirement includes both

3. *Design and development of the framework*

unit tests and integration tests to ensure that both individual components and the interaction between the components are comprehensively tested.

NFR4.3 Automated workflows: The framework should be integrated into a CI/CD system that supports automated tests, builds, and deployments.

NFR4.4 Documentation quality: Provision of comprehensive and comprehensible developer documentation.

Implementation Requirements

NFR5.1 Use of open source software components: Building the system using open source components and standard development programming languages.

NFR5.2 Automated deployments: Building the system using open source components and standard development programming languages.

NFR5.3 Expandability: The framework must be designed in such a way that it can easily be expanded with new functions or modules without impairing existing functionalities.

NFR5.4 Use of modern frameworks: Use of current and widely used frameworks to promote developer efficiency.

Interface Requirements

NFR6.1 Compatibility: The framework should be designed to be backward compatible with older Android versions to reach a broad user base.

NFR6.2 Industry standards: Ensure that the interfaces comply with current industry standards.

Legal Requirements

NFR7 Security and data protection: Robust data protection mechanisms to protect sensitive patient information.

Packaging Requirements

NFR8 **Installation:** Ensure quick and easy installation of the software.

Description of the development approach for the modular framework

Modular design principles are crucial for the development of software that is easy to maintain and extend. The basic idea of a modular design is to break a system down into smaller, independent modules, each of which fulfils a specific function or task. This has several advantages:

Advantages for maintainability

- **Easier troubleshooting and updates:** The division into modules makes it easier for developers to localise and rectify errors. If a module is faulty, it can be isolated and repaired without affecting the entire application.
- **Clear responsibilities:** Each module has a clearly defined task, which reduces complexity and makes maintenance easier. Developers can focus on specific parts of the framework without having to understand the entire framework [33].
- **Reusability:** A well-designed module can be reused in different projects [33]. This reduces the effort involved in developing new systems and promotes consistent solutions [33].
- **Improved testability:** Modules can be tested independently of each other, which makes it easier to create unit tests. This leads to higher test coverage and improves the quality of the entire system [33].

Advantages for expandability

- **Simple integration of new functions:** New features can be developed as stand-alone modules and integrated into the existing applications without the need for extensive changes to the existing code base [34].

3. *Design and development of the framework*

- **Scalability:** Modular systems are easier to scale. If additional functionality is required, new modules can be added without having to restructure the entire system [34].
- **Flexibility in development:** Development teams can work on different modules in parallel, which increases the speed of development. This also promotes specialisation within the teams, as developers can concentrate on specific modules and their functions [34].

4. Technical details on implementation

This chapter analyses the technical details of the implementation of the mHealth framework in depth, with a particular focus on the software architecture and the design patterns used. The choice of architecture and design patterns plays a decisive role in the performance, scalability and maintainability of the software.

A key component of this framework is the use of a modular architecture. This architecture allows different components of the application to be developed, tested and deployed independently of each other. Such a modular system not only increases the flexibility and reusability of the software. Particularly in the context of mHealth applications, which often place high demands on reliability and customisation, modular architecture offers significant advantages.

Alongside architecture, design patterns are an integral part of software development. They offer proven solutions for frequently occurring problems and contribute to improving software quality. As part of this project, various design patterns were used to optimize the functionality and maintainability of the framework. The patterns used include the adapter pattern, the observer pattern, and the factory pattern.

- **The Adapter Pattern:** is used to adapt the interfaces of incompatible classes and ensure smooth integration of external devices such as blood pressure monitors and scales. This is particularly important for Bluetooth connectivity, which plays a central role in the mHealth framework [34].
- **The Observer Pattern:** enables the implementation of an event-driven system in which changes to a patient's health data are automatically forwarded to the relevant components of the application. This

4. Technical details on implementation

contributes to the real-time monitoring and rapid response capability of the application [34]. On Android, this is achieved with Flows¹.

- **The Factory Pattern:** is used to encapsulate the creation of complex objects and increase the modularity of the software. This makes it easier to add new functionalities to the system and ensures a clear separation between the creation and use of objects [34].
- **The Model View ViewModel (MVVM) pattern:** is used to separate the user interface (View) from the business logic and data handling (Model) by utilizing ViewModel as a mediator. This pattern allows for easier testing and maintenance by clearly defining responsibilities and reducing tight coupling between components. In Android development, this is achieved using LiveData or Kotlin Flows for the communication between the ViewModel and the View [35].

Overall, this chapter provides a comprehensive overview of the technical details and shows how a powerful, scalable and maintainable mHealth framework can be developed through the targeted use of modern architectures and design patterns. All individual core modules are discussed.

Overview of the modules:

Convention Plugins

In the realm of software development, particularly within the framework, convention plugins serve as essential tools for standardizing and streamlining the build process. These plugins reside in the build-logic folder and are tailored specifically for common module configurations in the Spezi ecosystem [36]. Their main objective is to ensure consistency, improve reusability, and simplify the build process, ultimately enhancing developer productivity and reducing cognitive load [36].

Features:

- **Standardization:** Convention plugins ensure that all library modules adhere to specific conventions. This standardization is crucial

¹<https://developer.android.com/kotlin/flow>

for maintaining consistency across various projects. By following predefined rules, the development process becomes more predictable and manageable.

- **Improved Reusability:** The separation and modularization of build logic promote code reusability. Developers can leverage these reusable components across different projects, enhancing the efficiency of the development process. This is in line with idiomatic Gradle practices that advocate for modular and reusable build scripts [37].
- **Reduced Complexity and Cognitive Load:** By encapsulating commonly used configurations and conventions, convention plugins simplify individual build scripts. This encapsulation reduces the complexity of managing build scripts, thereby decreasing the cognitive load on developers. Simplified build scripts are easier to understand, maintain, and extend [38].
- **Improved Build Performance:** Convention plugins, unlike the traditional `buildSrc` directory, can be precompiled and treated like any other dependency [38]. This precompilation avoids the performance penalty associated with recompiling build logic on every build, leading to faster build times, particularly in large projects [38].
- **Increased Modularity and Isolation:** Using convention plugins allows for the modularization and isolation of build logic from the rest of the build script. This modular approach facilitates cleaner code management and minimizes the risk of bugs propagating through the build script. Additionally, it simplifies the testing of build logic [38].

To apply a convention plugin in the `build.gradle.kts`, one only has to add the following lines from listing 4.1.

```
1 plugins {  
2     alias(libs.plugins.spezi.application)  
3     alias(libs.plugins.spezi.compose)  
4 }
```

Source text 4.1: Convention Plugins.

Available Convention Plugins

- **spezi.application:** This convention plugin applies `com.android.application` and `org.jetbrains.kotlin.android` by default. It

4. Technical details on implementation

also incorporates the default project configuration of the `spezi.base` plugin. Moreover, it includes the `:core:logging` implementation and `:core:testing` test implementation dependencies.

- **spezi.compose**: This plugin provides the necessary configuration and dependencies for using Jetpack Compose. Note that it requires either `spezi.application` or `spezi.library` to be applied as well.
- **spezi.base**: The `spezi.base` plugin ensures consistent configuration of versions and compile options across all modules in the project. It is recommended for modules that are dependencies in either `spezi.application` or `spezi.library` plugins.
- **spezi.hilt**: This plugin includes all dependencies needed to use Hilt for dependency injection².
- **spezi.library**: Similar to `spezi.application`, this plugin applies `com.android.library` and `org.jetbrains.kotlin.android` by default. It also applies the default project configuration of the `spezi.base` plugin, along with the `:core:logging` implementation and `:core:testing` test implementation dependencies².

The adoption of convention plugins within the framework brings numerous benefits:

- **Consistency**: Ensures uniformity across different modules and projects.
- **Reusability**: Facilitates code reuse, reducing duplication and effort [37].
- **Simplification**: Makes build scripts easier to manage and comprehend [38].
- **Performance**: Enhances build performance by avoiding unnecessary recompilation [38].
- **Modularity**: Promotes clean code practices and modular development [38].

²<https://github.com/StanfordSpezi/SpeziKt/tree/main/build-logic>

Convention plugins are a strategic approach to managing build configurations in projects, significantly contributing to the efficiency and maintainability of the codebase. By adhering to these conventions, developers can focus more on the core functionalities of their projects rather than the intricacies of the build process.

Module Designsystem

The Design System Module is part of the framework, designed to provide a cohesive user interface and user experience components. It ensures consistent aesthetics and functionality across different parts of the application, enhancing both developer efficiency and user satisfaction.

Features

- **Theming:** Supports light and dark modes and customizable color schemes.
- **Components:** Includes reusable UI components such as a button optimized for accessibility and ease of use as well as a validated outlined text field.
- **Typography:** Implements a scalable typography system that adapts to different screen sizes and orientations.
- **Icons and Graphics:** Provides a set of commonly used icons and graphics that maintain high resolution and scalability across devices.
- **Spacings and Sizes:** Ensures consistent and customizable spacing and sizing for UI elements, promoting a uniform look and feel throughout the application.

To integrate the Design System Module into a module, add the following dependency to your `build.gradle` file 4.2.

```
1 dependencies {  
2     implementation(project(":core:designsystem"))  
3 }
```

Source text 4.2: Design System Module.

4. Technical details on implementation

To use a component from the design system, refer to the specific documentation³ included in the module.

A central design module that is used across all modules offers numerous advantages, especially in large projects like this framework.

- **Consistency:** A unified design system ensures that all user interfaces are consistent, which increases usability and reduces the learning curve for users. Once users have learned how a component works, they can apply this knowledge to the entire application [39]. This reduces the cognitive load and improves the user experience [39] [40].
- **Productivity and speed:** Using a centralized design system speeds up the development process [39] [40]. Developers and designers can use predefined components instead of starting from scratch each time. This saves time and resources and allows teams to focus on solving more complex problems [39] [40].
- **Quality and maintainability:** A well-maintained design system improves the maintainability and quality of the code. Bug fixes and updates in the design system are automatically applied to all modules and applications based on it, which increases consistency and stability [40] [39].
- **Scalability:** A design system facilitates the scaling of the framework. New features can be integrated more quickly as existing, tested components can be used [40]. This is particularly important in large projects with many parallel development threads [39] [40].

To summarize integrating the Design System Module into the framework and applications enhances both the aesthetic and functional quality of mHealth applications. By providing a standardized set of components and theming options, developers can ensure a consistent user experience, which is crucial for patient engagement and usability. For instance, the ENGAGE-HF project utilizes this module to create a seamless and intuitive interface for heart failure management, demonstrating the module's effectiveness in a real-world healthcare setting. But even within the framework itself, all modules use the design system.

³<https://spezi.health/SpeziKt/core/design/index.html>

Bluetooth

The Bluetooth module is crucial for the effective use of mHealth applications such as the ENGAGE-HF app, especially in the area of heart failure monitoring. The direct connection to medical devices such as blood pressure cuffs and scales enables continuous and accurate collection of health data. This data is essential for real-time monitoring and chronic disease management. Bluetooth integration allows patients to easily and seamlessly transmit their vital signs to the app, resulting in improved data accuracy and timely medical interventions [41].

The Bluetooth module provides comprehensive tools for managing Bluetooth Low Energy (BLE) functionalities on Android devices. It includes components for scanning, connecting and interacting with BLE devices. The functionality and integration into the Android framework is described below.

Components of the Bluetooth module:

- **BLEService:** The main API that encapsulates the capabilities for managing BLE device connections. It provides methods for starting and stopping the BLE service as well as flows for monitoring the service status and receiving events [42].
- **BLEDeviceScanner:** Responsible for scanning for nearby BLE devices. It enables scanning to be started and stopped and sends events for detected devices or scan errors [42].
- **BLEDeviceConnector:** Manages the connection to individual BLE devices. It manages the Bluetooth GATT connection and sends events for connection status changes and received measurements [42].
- **MeasurementMapper:** Maps Bluetooth GATT characteristics to specific measurement types, such as blood pressure or weight measurements [42].
- **PermissionChecker:** Checks Bluetooth-related authorisations on the device [42].

StateFlows enable efficient handling of state changes through reactive programming⁴. As soon as a device provides new data, the state is up-

⁴<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow>

4. Technical details on implementation

dated and the application can react immediately. Integration with Kotlin coroutines simplifies the handling of asynchronous operations, which significantly improves the readability and maintainability of the code⁵. In addition, StateFlows are lifecycle-aware and thread-safe, which means that UI updates only occur when the associated component is active⁵. This helps to minimise memory leaks and race conditions.

After the module has been added via Gradle, an instance of the `BLEService` can be used with the help of dependency injection. The `start()`-method of this service can be used to initiate the scan for nearby devices. It is possible to listen to both the states and the events of the BLE. The service can also be stopped again by calling the `stop()`-method.

An example of collecting the states could look as in listing 4.3, whereby the handling of the respective states is implemented.

```
1 @HiltViewModel
2 class BluetoothViewModel @Inject constructor(
3     private val bleService: BLEService,
4 ) : ViewModel() {
5     private fun start() {
6         bleService.start()
7         viewModelScope.launch {
8             bleService.state.collect { state ->
9                 when (state) {
10                    is BLEServiceState.Scanning -> {
11                        // Handle Scanning state
12                    }
13                    // Handle other states
14                }
15            }
16        }
17    }
18 }
```

Source text 4.3: BluetoothViewModel: Collect States.

The Bluetooth module has implemented default mappers that can convert received measurement data, such as blood pressure and weight measurements, into specific objects. These events can also be collected and processed, as shown in Listing 4.4. The measurements mapped in this way can be used directly, for example to update the UI state and display a

⁵<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow>

dialogue with the new measured values. Alternatively, this data can also be processed in the background.

An example of the implementation is shown in Listing 4.4. This listing shows how the measurements can be used to update the UI state or process it in the background.

```
1 viewModelScope.launch {
2     bleService.events.collect { event ->
3         // Handle BLE service events
4         when (event) {
5             is BLEServiceEvent.MeasurementReceived -> {
6                 _uiState.update {
7                     it.copy(
8                         measurementDialog = uiStateMapper.
9                             mapToMeasurementDialogUiState(
10                                event.measurement
11                            )
12                    )
13                }
14                // handle other events
15            }
16        }
17    }
```

Source text 4.4: BluetoothViewModel: MeasurementReceived.

This Listing 4.4 shows how events are intercepted and the corresponding measurements are processed in order to update the UI status or perform further background processing.

When developing the Bluetooth module, we deliberately decided against the use of callbacks and LiveData and chose StateFlows instead. This decision is based on several important factors:

Callbacks:

- **Complexity and maintainability:** Callbacks can quickly lead to a confusing ‘callback hell’, especially with nested asynchronous operations. This makes the readability and maintainability of the code considerably more difficult [43].
- **Error handling:** The handling of errors and the chaining of operations are complex and error-prone [43].

4. Technical details on implementation

LiveData:

- **UI connectivity:** LiveData is mainly designed for UI-bound data in the MVVM architecture. It offers less flexibility for general asynchronous operations and state management [44].
- **Complexity in non-UI components:** In non-UI components such as services or repositories, the use of LiveData is not always ideal and can lead to unnecessary complexity [44].

The choice of StateFlows for the development of the Bluetooth module of the mHealth framework enables accurate and efficient collection of medical data, which is essential for real-time monitoring and management of chronic diseases such as heart failure⁶. The integration of StateFlows to handle the BLE-functionality supports the development of a robust, scalable and user-friendly mHealth application that meets the modern requirements for mobile health solutions.

Logging

The logging module is an essential component in any software development framework, especially in mHealth applications, where accurate tracking of events, errors, and user activities is crucial for maintaining system reliability, ensuring data integrity, and aiding in debugging and performance monitoring. This section provides a detailed examination of the logging module implemented in the Spezi framework [45].

The logging module in the Spezi framework offers a versatile and efficient logging utility named **Logger**. This utility is crafted to manage various logging strategies that are essential for the complex and dynamic environment of mHealth applications. The **Logger** uses inline functions to log messages, which helps in avoiding unnecessary memory allocation for large string messages, thereby optimizing performance [46].

Key features of the **Logger** include:

- **Flexible Configuration:** It allows comprehensive configuration of logger settings, enabling developers to tailor the logging behavior to the specific needs of their application [46].

⁶<https://developer.android.com/kotlin/flow/stateflow-and-sharedflow>

- **Tagging and Grouping:** The logger supports tagging, which helps in categorizing and filtering logs based on different components or features within the application [46].
- **Customizable Logging Strategies:** Through the use of tags and configurations, different logging strategies can be implemented, making the system adaptable to various operational scenarios [46].

The logging API offers various options for logging messages and is therefore flexible for different logging requirements. The following listing 4.5 shows a few examples of the configuration options [46].

```

1 private val logger by SpeziLogger()
2
3 private val groupLogger by groupLogger("specificFeature")
4
5 private val customLogger by Logger {
6     tag = "TAG"
7     messagePrefix = "prefix"
8     loggingStrategy = LoggingStrategy.LOG
9     forceEnabled = true
10 }

```

Source text 4.5: Log: Configuration.

In the listing 4.5 [46] you can see:

- **Standard logger:** Derives the tag name automatically from the component in which it is defined [46].
- **Group logger:** Uses a predefined tag (`specificFeature`) and can have optional configurations [46].
- **Custom logger:** Allows you to set a custom tag and message prefix and can force logging independently of the global configuration [46].

Which can then be used as in listing 4.6 [46].

```

1 logger.i { "Example_log_using_default_config_and_'MyClass'_as_tag" }
2
3 logger.tag("NEW_TAG").i { "Example_log_using_default_config_and_
4     NEW_TAG_(only_for_this_log)_as_tag" }
5
6 customLogger.i { "Example_log_using_config_passed_in_customLogger" }

```

4. Technical details on implementation

```
7 groupLogger.i { "Example_log_using_default_config, tag 'myFeature'
  and_prefixes_the_message_with 'MyClass-' " }
8
9 Logger.e(Error("Something_went_wrong")) { "Alternative_log_using_
  default_log_config_and 'edu.stanford..logger_as_tag' " }
```

Source text 4.6: Log: Configuration.

The logging module in the Spezi framework exemplifies a robust and flexible solution for handling logging. By providing configurable logging utilities, it ensures that developers can efficiently monitor, debug, and maintain their applications, thereby enhancing the overall reliability and effectiveness of mHealth solutions. This modular and scalable approach aligns with the framework's goal of supporting dynamic and user-centric healthcare applications.

Navigation

This module offers a systematic approach to managing navigation events within the framework. It defines navigation events and a navigator interface to oversee navigation within the application.

The `NavigationEvent` interface acts as a foundation for all navigation events. Any class that represents a navigation event should implement this interface. Framework users can define their custom navigation events to specific routes by implementing this interface.

For example in listing 4.7 the `OnboardingNavigationEvent` is a sealed class that extends `NavigationEvent`. Each Module provides an implementation of `NavigationEvent` so the user has access to all public available navigation destinations.

```
1 sealed class OnboardingNavigationEvent : NavigationEvent {
2     data object InvitationCodeScreen : OnboardingNavigationEvent()
3     data object OnboardingScreen : OnboardingNavigationEvent()
4     data object SequentialOnboardingScreen :
5         OnboardingNavigationEvent()
6     data object ConsentScreen : OnboardingNavigationEvent()
7 }
```

Source text 4.7: OnboardingNavigationEvent.

The `Navigator` (Listing: 4.8) is an interface defining the contract for a navigator responsible for handling navigation events.

```

1 interface Navigator {
2     val events: SharedFlow<NavigationEvent>
3
4     fun navigateTo(event: NavigationEvent)
5 }

```

Source text 4.8: Navigator.

The `events` in the navigator are a `SharedFlow` of `NavigationEvent` objects. This flow emits navigation events to be observed and acted upon from the app's implementation of the `Navigator`.

The `navigateTo(event:NavigationEvent)` method takes a `NavigationEvent` and triggers the navigation to the corresponding screen.

For a framework developer, this means that in new modules, they only need to implement the `NavigationEvent` interface and provide the navigation destinations. For framework users, it means they only need to provide a `NavGraph` in their app like in listing 4.9.

```

1 fun NavGraphBuilder.mainGraph() {
2     composable<Routes.RegisterScreen> {
3         val args = it.toRoute<Routes.RegisterScreen>()
4         RegisterScreen(args.isGoogleSignIn)
5     }
6 }

```

Source text 4.9: NavGraphBuilder.

Then provide a `Navigator` implementation in the app or simply use dependency injection to provide the default implementation as shown in listing 4.10 [47].

```

1 @Module
2 @InstallIn(SingletonComponent::class)
3 object NavigationModule {
4     @Provides
5     @Singleton
6     fun provideNavigator(): Navigator = DefaultNavigator()
7 }

```

Source text 4.10: NavigationModule.

In a `Routes` class 4.11, the necessary routes can be defined.

```

1 @Serializable
2 sealed class Routes {
3
4     @Serializable

```

4. Technical details on implementation

```
5 data class RegisterScreen(val isGoogleSignIn: Boolean) : Routes()
6 }
```

Source text 4.11: Routes.

One has to provide an `AppNavigation` composable. In listing 4.12 is an example of how to implement the `AppNavigation` in the App module. You can use the `Navigator` interface to navigate to the desired screen based on `Routes`.

```
1 @Composable
2 fun AppNavigation(navigator: Navigator) {
3     val navController = rememberNavController()
4     val coroutineScope = rememberCoroutineScope()
5     LaunchedEffect(navigator) {
6         coroutineScope.launch {
7             navigator.events.collect { event ->
8                 when (event) {
9                     is OnboardingNavigationEvent.ConsentScreen ->
10                        navController.navigate(Routes.ConsentScreen)
11                }
12            }
13        }
14        NavHost(
15            navController = navController,
16            startDestination = Routes.OnboardingScreen,
17        ) {
18            mainGraph()
19        }
20 }
```

Source text 4.12: AppNavigation.

After setting up, the user can utilize the `Navigator` interface within their app module to navigate to the desired screen. To achieve this, they can inject the `Navigator` into any class that requires navigation, as demonstrated in listing 4.13.

```
1 class DefaultOnboardingRepository @Inject constructor(
2     private val navigator: Navigator
3 ) : OnboardingRepository
```

Source text 4.13: Inject navigator.

To navigate to the desired screen, the user can utilize the `navigateTo` function with the appropriate `NavigationEvent`. Detailed implementation

can be found in listing 4.14.

```
1 navigator.navigateTo(OnboardingNavigationEvent.  
    SequentialOnboardingScreen)
```

Source text 4.14: `navigateTo()`.

This framework's design effectively addresses key functional requirements, focusing on modularity (FR1) and reusability (FR2), which are essential for such a framework. This navigation architecture supports interchangeable components for example for disease-specific functionalities, promoting adaptability to new medical research and treatment methods. This modularity also ensures quick integration of new knowledge and technologies into clinical practice, enhancing the effectiveness and efficiency of patient care. By implementing the `NavigationEvent` interface, new modules can seamlessly integrate with the existing system, facilitating rapid deployment of updates and new functionalities without requiring major overhauls. This approach aligns with best practices in software architecture, emphasizing modular design for creating flexible and maintainable systems [48]. Reusability is ensured through shared interfaces and sealed classes, enabling developers to define specific navigation events and routes for different conditions without rewriting core logic. This ensures the framework can be reused and adapted across multiple applications, reducing development time and effort while maintaining consistency and reliability.

By addressing these functional requirements in the navigation, the framework not only promotes the efficient and effective delivery of patient care but also ensures that it can be continuously adapted and improved, thereby aligning with both current needs and future advancements in medical technology.

Testing

Automated testing is a vital aspect of ensuring code quality, particularly within the development of mobile health applications such as ENGAGE-HF and Open Source Frameworks like Spezi [49]. Our testing process is embedded in GitHub Actions through the workflow titled **Build, Test, and Analyze**, which supports automated builds, testing, and analysis of the codebase. This approach aligns with the Non-Functional Requirement Automated Workflows (NFR4.3), ensuring all tests are automatically trig-

4. Technical details on implementation

gered each time a pull request is submitted to the main branch, providing continuous integration and immediate feedback on the code's integrity [50]. Key elements of the GitHub Action workflow include:

1. **Triggers:**

- **workflow_dispatch:** Manually triggered.
- **workflow_call:** Can be triggered by another workflow.

2. **Jobs:**

- **Detekt:** Runs static code analysis for Kotlin using Detekt.
- **Build, Test, and Analyze:** Builds the project, runs tests, performs CodeQL analysis, and uploads code coverage reports.
- **Test:** Runs instrumented tests on different Android API levels and devices.
- **Dokka:** Generates and deploys project documentation to GitHub Pages. More in section 4.

3. **Steps:** Actions like checking out code, setting up Java, running tests, and deploying documentation.

4. **Secrets:** Uses GitHub secrets for authentication (e.g., Codecov, GitHub tokens).

This automated process also supports Use of Open Source Software Components (NFR5.1) by integrating widely adopted open-source libraries such as Detekt for static code analysis and JaCoCo for code coverage. This integration enhances the development workflow and promotes adherence to Industry Standards (NFR6.2), as these libraries are standard tools in modern Android and Kotlin development [49].

This project utilizes JUnit for executing tests, Truth for assertions, and MockK for mocking dependencies. This combination aligns with Testability (NFR4.2), ensuring that our framework is designed for comprehensive testing at both the unit and integration levels [49].

1. **JUnit:** A widely-used test runner that controls the execution of tests.
2. **Truth:** Provides clear and readable assertions, making it easier to interpret test results.

3. **MockK**: A Kotlin-specific mocking framework that simplifies creating mock objects and verifying behavior.

These tools were selected to optimize the testability of the codebase, ensuring that each component can be thoroughly tested in isolation (unit tests) and as part of an integrated system (integration tests). The structure of the tests follows a **Given-When-Then** format, a common best practice that makes the tests clear and understandable [51].

The following test exemplifies this structure, ensuring a proper separation of concerns and clarity in defining initial conditions, actions, and expected outcomes.

```
1 @Test
2 fun 'it should handle idle state correctly'() = runTestUnconfined {
3     // given
4     createViewModel()
5
6     // when
7     bleServiceState.emit(BLEServiceState.Idle)
8
9     // then
10    assertBluetoothUiState(state = BluetoothUiState.Idle)
11 }
```

Source text 4.15: Unit Test Setup Example.

- **Given**: Setting up the test environment and initial conditions (`createViewModel()`).
- **When**: Triggering the behavior under test (`bleServiceState.emit()`).
- **Then**: Verifying the outcome with an assertion (`assertBluetoothUiState()`).

This format, combined with the tools mentioned, contributes to increased testability and alignment with NFR4.2.

In addition to unit tests, instrumented tests are executed using an Android emulator within the GitHub Action workflow. The emulator simulates real Android environments, allowing us to validate the app's behavior across different Android API levels and device configurations. This ensures

4. Technical details on implementation

compliance with Industry Standards (NFR6.2), as we ensure compatibility with various Android environments.

The following code shows how we structure these instrumented tests:

```
1 class HealthPageTest {
2
3     @get:Rule
4     val composeTestRule = createComposeRule()
5
6     @Test
7     fun 'test health page root is displayed'() {
8         // given
9         setState(state = getSuccessState())
10
11         // then
12         healthPage {
13             assertIsDisplayed()
14         }
15     }
16
17     private fun healthPage(block: HealthPageSimulator.() -> Unit) {
18         HealthPageSimulator(composeTestRule).apply(block)
19     }
20 }
```

Source text 4.16: Android Test Example Setup.

The `HealthPageSimulator` (Listing: 4.17) encapsulates test logic for interacting with the Android UI components. This abstraction improves the test code's maintainability and reusability, adhering to Testability (NFR4.2), which mandates that the framework should allow easy and comprehensive testing.

```
1 class HealthPageSimulator(
2     private val composeTestRule: ComposeTestRule,
3 ) {
4     private val root = composeTestRule.onNodeWithIdentifier(
5         HealthPageTestIdentifier.ROOT)
6
7     fun assertIsDisplayed() {
8         root.assertIsDisplayed()
9     }
10 }
```

Source text 4.17: Android Test Simulator Example.

A critical issue encountered during our testing process involves JaCoCo and its inability to generate coverage reports due to the large size of the `org.hl7.fhir.r4.JsonParser` class⁷. This failure results in the pipeline's inability to process code coverage properly, especially for Android-specific tests, leading to incorrect coverage calculations for new pull requests.

Despite trying various solutions, such as excluding the class from coverage and optimizing heap memory, the issue persists. We have reported this to the respective maintainers (e.g., HAPI FHIR issue #1688⁸ and related issue #3268⁹ and JaCoCo issue #1653¹⁰). This unresolved issue forces us to temporarily deactivate the code coverage threshold in the detekt configuration, which affects the overall pipeline's efficacy.

Automated testing in the project plays a crucial role in maintaining code quality and ensuring adherence to Testability (NFR4.2) and Automated Workflows (NFR4.3). By leveraging open-source software components such as GitHub Actions, JUnit, MockK, and Truth, we maintain high standards for code reliability, making sure that our app aligns with industry standards (NFR6.2). Although challenges with JaCoCo and large dependencies exist, ongoing efforts are being made to resolve these issues and ensure a fully automated and reliable testing pipeline.

Dokka Documentation

Dokka was chosen over other tools due to its seamless integration with Kotlin, offering native support for Kotlin-specific features and syntax, which other documentation generators often lack¹¹. This ensures that our documentation remains accurate and fully aligned with the language's capabilities, reducing the need for manual adjustments or workarounds¹¹. It allows the generation of structured documentation directly from source code comments, ensuring consistency and clarity¹². For open-source projects, this is particularly valuable as it enables developers to maintain comprehensive and up-to-date documentation [52]. In this section, we explore how

⁷<https://raw.githubusercontent.com/hapifhir/org.hl7.fhir.core/master/org.hl7.fhir.r4/src/main/java/org/hl7/fhir/r4/formats/JsonParser.java>

⁸<https://github.com/hapifhir/org.hl7.fhir.core/issues/1688>

⁹<https://github.com/hapifhir/hapi-fhir/issues/3268>

¹⁰<https://github.com/jacoco/jacoco/issues/1653>

¹¹<https://kotlinlang.org/docs/dokka-introduction.html>

¹²<https://github.com/Kotlin/dokka>

4. Technical details on implementation

the integration of Dokka into our project helps fulfill several non-functional requirements, specifically focusing on maintainability (NFR4.1), automated workflows (NFR4.3) and documentation quality (NFR4.4).

```
1 fun Project.setupDokka() {
2     apply(plugin = rootProject.libs.plugins.dokka.get().pluginId)
3
4     if (this != rootProject) {
5         rootProject.tasks.named("dokkaHtmlMultiModule") {
6             dependsOn("${project.path}:dokkaHtml")
7         }
8     }
9
10    tasks.withType<DokkaTaskPartial>().configureEach {
11        dokkaSourceSets.configureEach {
12            noAndroidSdkLink.set(false)
13            skipDeprecated.set(true)
14            skipEmptyPackages.set(true)
15            includeNonPublic.set(false)
16            jdkVersion.set(JavaVersion.VERSION_17.majorVersion.toInt())
17            if (file("README.md").exists()) {
18                includes.from("README.md")
19            }
20        }
21    }
22
23    val dokkaHtmlMultiModule = tasks.findByName("dokkaHtmlMultiModule")
24    ?: tasks.create("dokkaHtmlMultiModule",
25        DokkaTaskPartial::class.java)
26
27    rootProject.tasks.named("dokkaHtmlMultiModule") {
28        dependsOn(dokkaHtmlMultiModule)
29    }
30 }
```

Source text 4.18: Dokka Setup.

In the setup configuration (Listing 4.18), the Dokka plugin is automatically applied to the entire project so that all modules benefit from standardised documentation creation. This configuration also supports our multi-module structure and ensures that the documentation is created across all modules, which ensures consistency, especially in our large code base. To make the documentation more relevant and clearer, obsolete methods and empty

packages are skipped. By integrating `README.md`, cross-project explanations are included in the documentation, which increases its scope and comprehensibility. This setup automates the documentation process, making it easier to update and maintain, which directly aligns with NFR4.1. To ensure continuous updates and deployment of the documentation, we set up a GitHub Actions workflow (Listing 4.19):

```
1 name: Dokka Documentation Deployment
2 on:
3   push:
4     branches:
5       - main
6
7 jobs:
8   dokka:
9     name: Dokka Documentation Deployment
10    runs-on: ubuntu-latest
11    steps:
12      - name: Checkout code
13        uses: actions/checkout@v4
14
15      - name: Set up JDK 17
16        uses: actions/setup-java@v4
17        with:
18          distribution: 'temurin'
19          java-version: '17'
20          cache: gradle
21
22      - name: Setup Gradle
23        uses: gradle/actions/setup-gradle@v3
24
25      - name: Run Dokka with Gradle
26        run: ./gradlew dokkaHtmlMultiModule
27
28      - name: Deploy to GitHub Pages
29        if: github.ref == 'refs/heads/main'
30        uses: JamesIves/github-pages-deploy-action@v4
31        with:
32          branch: gh-pages
33          folder: build/dokka/htmlMultiModule
```

Source text 4.19: Dokka Github Action.

This workflow automates the following steps: Firstly, the code is checked out from the GitHub repository. Java and Gradle are then set up by

4. *Technical details on implementation*

installing JDK 17 and preparing Gradle for the build. Dokka then creates the documentation based on the project configuration. Finally, the generated documentation is automatically published on the gh-pages branch, making it accessible as public project documentation. By automating these steps, we address NFR4.3, ensuring that builds, tests, and deployments are handled automatically without manual intervention. This streamlines the development process, improving both efficiency and reliability. As outlined in NFR4.1, clear and up-to-date documentation is crucial for maintaining a project efficiently. By utilizing Dokka and Github Actions, we ensure that the documentation is always in sync with the latest code changes, as it is generated directly from code comments [52]. This eliminates the need for developers to manually update separate documentation files, significantly reducing the likelihood of inconsistencies. Additionally, by configuring Dokka to include key files like the README.md, we ensure that project overviews are always part of the documentation (Figure: 4.1), improving overall clarity and reducing maintenance overhead. Automated documentation tools are essential for maintainability, as they reduce technical debt and support clear communication within the development team [52].



Figure 4.1.: Dokka example documentation.

Account Module

The Account Module within the framework is a central component designed to handle user authentication and account management to provide security

4. Technical details on implementation

(NFR7). This module includes both the Login Screen and Register Screen, as illustrated in listing 4.2. These screens enable users to log in to their existing accounts or create new ones, while demonstrating how the module integrates with various authentication services.

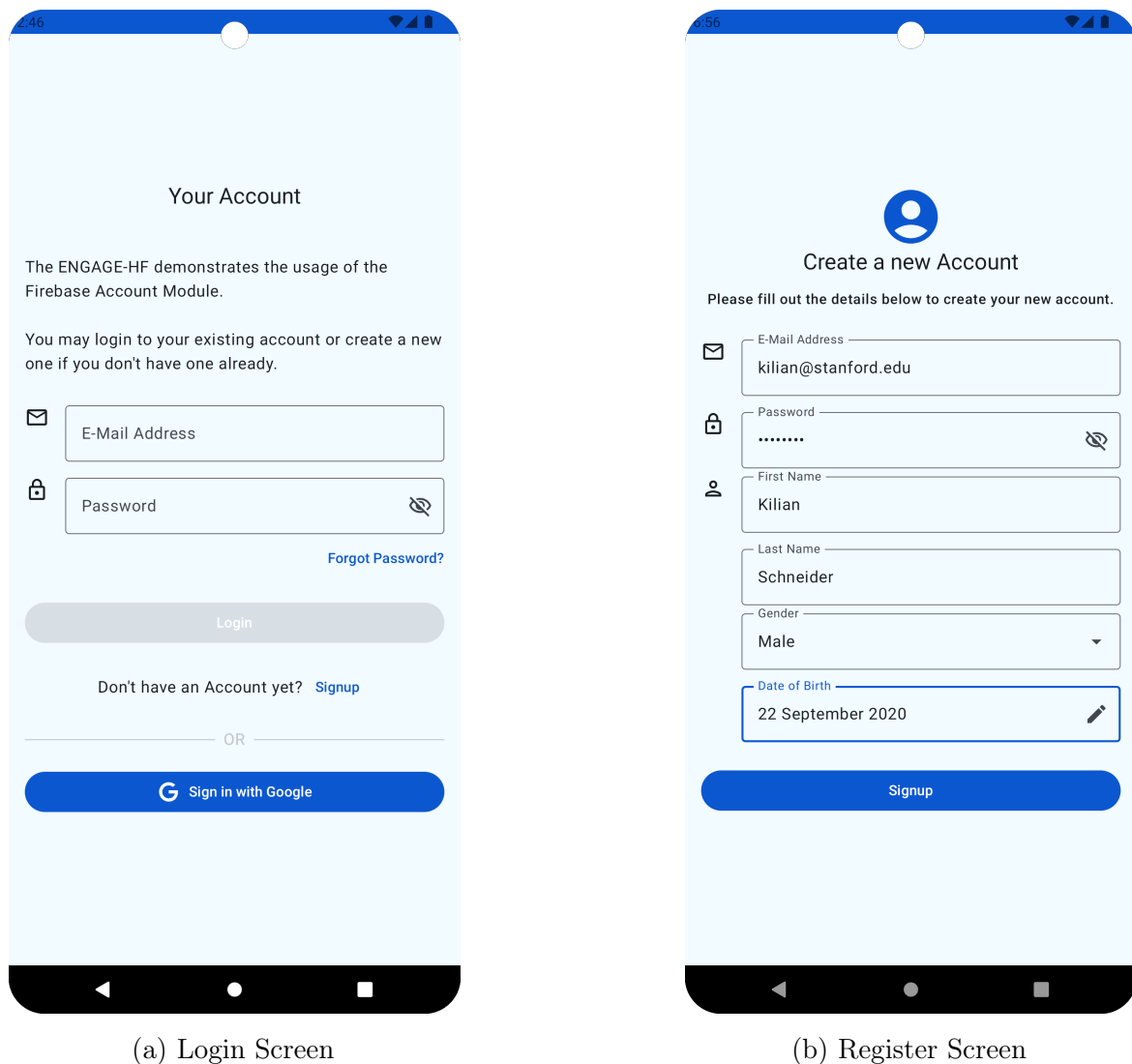


Figure 4.2.: Login and Register Screen

This module is built to handle the core functionality required for managing user accounts, utilizing the `AuthenticationManager` interface (Listing 4.20).

```
1 interface AuthenticationManager {
2     suspend fun linkUserToGoogleAccount(
3         googleIdToken: String,
4         user: User
```



```

5     ): Result<Unit>
6
7     suspend fun signUpWithEmailAndPassword(
8         user: User
9     ): Result<Unit>
10
11    suspend fun saveUserData(
12        user: User
13    ): Result<Unit>
14
15    suspend fun sendForgotPasswordEmail(email: String): Result<Unit>
16
17    suspend fun signIn(email: String, password: String): Result<Unit>
18
19    suspend fun signInWithGoogle(): Result<Unit>
20 }

```

Source text 4.20: AuthenticationManager Interface.

Key functions implemented include:

- **signUpWithEmailAndPassword**: Allows users to register with their email and password.
- **saveUserData**: Handles saving user details such as name, gender, and date of birth after registration.
- **sendForgetPasswordEmail**: Sends a password reset email to users who request it.
- **signInWithEmailAndPassword**: Allows users to log in using email and password credentials.
- **signInWithGoogle**: Implements a more modern authentication method using Google's sign-in service, enhancing user convenience.

The `FirebaseAuthenticationManager` provides a default implementation of the `AuthenticationManager` interface using Firebase. However, one of the significant advantages of the Spezi framework is its flexibility and modularity (FR1). The interface can be easily implemented using other services like AWS or Supabase, demonstrating the system's reusability and adaptability in different environments.

Additionally, another important interface in the module is the `UserSession Manager` (Listing: 4.21).

4. Technical details on implementation

```
1 interface UserSessionManager {
2     suspend fun uploadConsentPdf(pdfBytes: ByteArray): Result<Unit>
3     suspend fun getUserState(): UserState
4     fun observeUserState(): Flow<UserState>
5     fun getUserUid(): String?
6     fun getUserInfo(): UserInfo
7 }
```

Source text 4.21: UserSessionManager Interface.

This interface offers functionality related to user session management, including:

- **uploadConsentPdf**: Handles the uploading of consent documents.
- **getUserState and observeUserState**: Functions to retrieve and monitor the user's state, whether they are registered, anonymous, or awaiting consent.
- **getUserUid and getUserInfo**: These provide access to user identifiers and details within the app.

The **UserState** defines the user's status in various ways. The initial state, called **NotInitialized**, occurs when no information has been received yet. For users who are not registered or logged in, the status is labeled as **Anonymous**. When a user has completed the registration process, the status changes to **Registered**, which also includes information on whether the user has submitted their consent.

As **observeUserState** returns a flow, this can be collected elsewhere and reacted to. For example, you can navigate directly to the start page when the application is started if the user is logged in, or automatically redirect them back to the login page if they get logged out.

This system's modular approach allows developers to integrate additional services and components without disrupting the overall architecture. The Login (Figure: 4.2a) and Register screens (Figure: 4.2b) serve as key interaction points for users, leveraging the **UserInfo** object to display and manage user information.

In addition to the core functionality provided by the Account Module, the Login and Register Screens themselves are fully customizable (NFR5.3). While the module comes with predefined views for logging in and registering

users, developers have the flexibility to create their own custom screens if needed.

Developers can choose to bypass the provided screens altogether and instead integrate the `AuthenticationManager` directly into their own custom views futuring modularity (FR1). This means that the logic behind user registration, login, and other authentication-related tasks can be reused, while maintaining the freedom to design entirely custom user interfaces that fit specific branding, user experience requirements, or additional workflows.

Moreover, developers can inject the ViewModels associated with the `AuthenticationManager` (Listing: 4.20) into these custom views, ensuring that all necessary logic for authentication remains in the background (Listing: 4.22).

```
1 @Composable
2 fun LoginScreen() {
3     val viewModel = hiltViewModel<LoginViewModel>()
4     val customViewModel = hiltViewModel<CustomLoginViewModel>()
5     val uiState by viewModel.uiState.collectAsState()
6     LoginScreen(
7         uiState = uiState,
8         onAction = viewModel::onAction,
9         customOnAction = customViewModel::onAction,
10    )
11 }
```

Source text 4.22: Inject Default LoginViewModel beside a CustomLoginViewModel.

Additionally, the ViewModels can be extended with custom logic, enabling further customization based on the unique needs of the application. This approach allows for modular (FR1), flexible development, where the authentication system is reused (FR2) while offering complete control over how the user interface is presented and interacts with the system.

By offering customizable authentication mechanisms and easy integration of third-party services, the module can be adapted for various use cases across different mHealth applications, such as managing chronic diseases like heart failure.

The ability to switch between services (Firebase¹³, Ory¹⁴, Supabase¹⁵,

¹³<https://firebase.google.com/docs/auth>

¹⁴<https://www.ory.sh/>

¹⁵<https://supabase.com/>

4. Technical details on implementation

etc.) without rewriting the core application logic highlights its scalability (NFR3.1). Moreover, developers can fully control the user interface by customizing screens or integrating the underlying logic into their own designs. This level of flexibility is especially useful for creating tailored applications for diverse patient populations or studies (FR4).

This modular design aligns perfectly with the project's goal of a resusability (FR2), providing flexibility and expandability (NFR5.3) to adjust to new requirements, technologies, or studies. It also promotes faster development cycles since custom screens can reuse existing logic while maintaining unique branding or user experience requirements.

The framework thus not only supports a variety of possible authentication services (FR5) but also allows customization of the presentation layer, ensuring it fits seamlessly into any clinical or research context, making it ideal for large-scale (NFR3.1) mHealth studies.

Contact

The Contact module provides a view to display contact information in an application. This functionality is crucial for ensuring that users can easily access contact details, which is especially important in mHealth applications where communication with healthcare providers, support staff, and other stakeholders is critical [19].

Providing a dedicated view (Figure: 4.3) for contact information enhances the user experience by making it easy for users to find and use contact details. This also means this structured and consistent way to display contact information helps maintain a professional and user-friendly interface, which is essential for building trust and reliability in health-related applications.

There are various predefined contact options: Telephone number, e-mail, website and geographical address. A contact person can optionally also have a name, title, description and company. This makes the contact page flexible and versatile. Each contact option has a preset action: for phone numbers a call is started and for addresses the navigation is opened. To use this module, only the provided interface needs to be implemented - everything else is set up automatically.

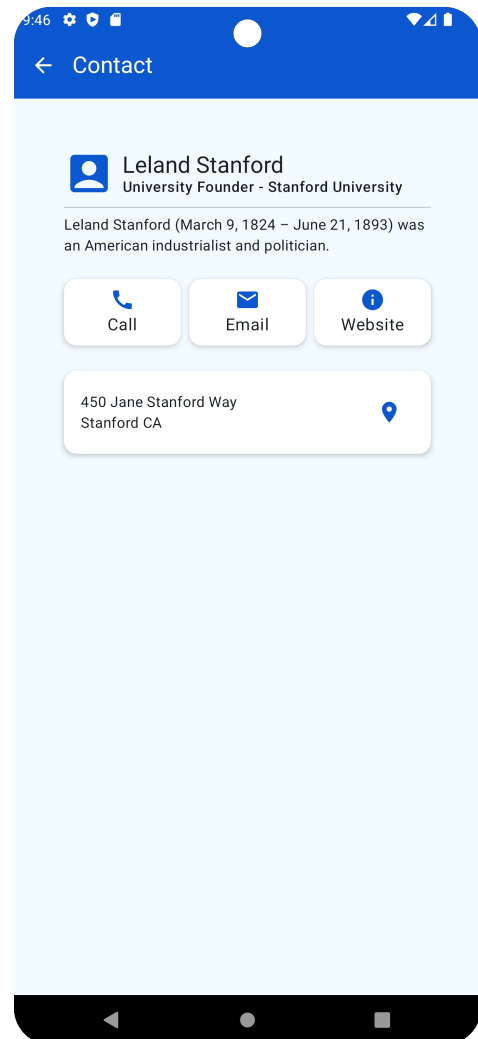


Figure 4.3.: Contact Screen Example.

```
1 interface ContactRepository {  
2     fun getContact(): Contact  
3 }
```

Source text 4.23: ContactRepository Interface.

Health Connect on FHIR

This module serves as a critical component in the integration of mobile health data with standardized healthcare records.

The primary purpose of the `HealthConnectOnFHIR` module is to bridge the gap between Android’s Health Connect data and HL7® FHIR® R4 standards [11]. It achieves this by providing a sophisticated mapper that converts various health records from Android’s Health Connect into corresponding FHIR Observations and vice versa¹⁶ [53]. These conversions use standardized codes such as Logical Observation Identifiers Names and Codes (LOINC) to ensure consistency and interoperability with other health data systems [54].

The mapping process involves translating different types of health records, such as blood pressure, heart rate, and body weight, into FHIR Observations [55]. This ensures that health data collected on mobile devices can be seamlessly integrated into EHR and other clinical systems that utilize the FHIR standard [56].

The module uses a predefined mapping table that correlates specific health records from Android Health Connect with their respective FHIR Observation categories, LOINC codes, and measurement units. For instance see table 4.1:

Health Connect Record	FHIR Observation Category	LOINC Code	Unit	Display
Active Calories Burned	Activity	41981-2	kcal	Calories burned
Blood Glucose	Vital Signs	41653-7	mg/dL	Glucose Glucometer (BldC) [Mass/Vol]
Blood Pressure	Vital Signs	85354-9	mmHg	Blood pressure panel with all children optional
Heart Rate	Vital Signs	8867-4	/min	Heart rate
Weight	Vital Signs	29463-7	kg	Body weight

Table 4.1.: Health Connect Records and Corresponding FHIR Observation Categories

¹⁶<https://developer.android.com/health-and-fitness/guides/health-connect>

For example `ActiveCaloriesBurnedRecord` is mapped to an `Activity` FHIR Observation with LOINC code 41981-2, measured in kilocalories (kcal) and `BloodPressureRecord` is categorized under `Vital Signs` with LOINC code 85354-9, where the measurements are in millimeters of mercury (mmHg) [57]. This mapping table is ensuring that the data is not only standardized but also accurately reflects the clinical observations it represents.

The `HealthConnectOnFHIR` module is implemented as a Hilt module, ensuring that dependencies such as the interfaces `RecordToObservationMapper` and `ObservationsToRecordMapper` are provided as singletons within the application which uses the module. The core functionality is encapsulated in the internal `RecordToObservationMapperImpl` and `ObservationToRecordMapperImpl` class, which defines methods for converting specific health records into FHIR Observations and vice versa.

For example, the `mapBloodPressureRecord`-method converts a `Blood PressureRecord` into a structured FHIR Observation, complete with systolic and diastolic components [57]. This method ensures that each health metric is properly represented and can be used effectively within clinical systems [57].

The `HealthConnectOnFHIR` module exemplifies the framework's commitment to creating interoperable (FR1) and reusable components (FR2). By adhering to FHIR standards, this module allows health data collected on Android devices to be integrated into a broader healthcare ecosystem, enabling better patient monitoring, data analysis, and clinical decision-making.

Onboarding

The Onboarding module provides user interface components to onboard a user to an application, including the possibility of retrieving consent for study participation. This module ensures that the onboarding process is smooth and user-friendly, facilitating the collection of necessary user information and consents efficiently.

4. Technical details on implementation

Consent

This view is essential for legally documenting the user's consent to participate in the study, ensuring compliance with ethical and legal standards. It provides a clear and straightforward way for users to understand what they are consenting to and gives them the means to provide their signature electronically.

The Consent View (Figure 4.4) is designed to gather user consent for study participation. It includes fields for entering the user's first and last names and a section for capturing the user's signature. This view ensures that users can easily provide their consent in a legally binding format.

The Consent Screen (Figure 4.4) offers a range of useful functions that enable users to enter their personal data in a simple and efficient manner and to give legal consent. In the 'First Name & Surname' section, users can enter their personal information. There is also a signature area where users can draw their signature, which is required for legal consent. If the signature does not meet expectations, an 'Undo' button allows users to delete the signature and start again. Finally, there is an 'I agree' button which is used to submit the form as soon as the user is satisfied with their details and signature. In addition, a Markdown parser is provided so that the user of the page requires as little technical knowledge as possible and can simply import a Markdown document as a contract. A `PDFService` is also provided, which converts the Markdown document into a PDF document and adds the signature at the bottom. The user can then decide for themselves what happens to the PDF or use the predefined `FirestorePdfUploadService`, which takes care of the upload to a Firebase storage. This is done in the same way as

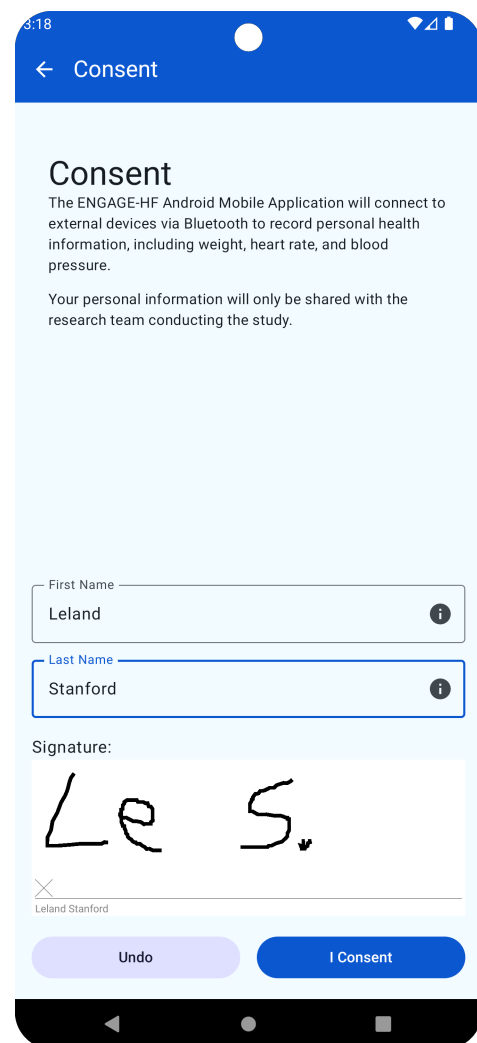


Figure 4.4.: Consent Screen.

in the `EngageInvitationCodeRepository` see Listing 4.24.

Invitation

The Invitation Code View (Figure: 4.5) is crucial for maintaining the security and integrity of the study. By requiring an invitation code, we can ensure that only eligible participants, who have been pre-approved and invited, can join the study. This helps in maintaining the quality and control of the study population. The view (Figure: 4.5) is intended for users to enter their invitation codes to join e.g. the ENGAGE-HF case study.

In the `Invitation code` field, users can enter the invitation code they have received. By clicking on the `Redeem invitation code` button, the code is submitted for verification. For users who already have an account, there is also a `I already have an account` link that offers an alternative login option.

The use of the view is designed to be extremely user-friendly. As shown in Listing 4.24, the user of the framework only has to provide an implementation of the `InvitationCodeRepository` interface. The actions and data contained in this interface are then used in the view. This can be easily called up using the methods described in the navigation Listing 4.13.

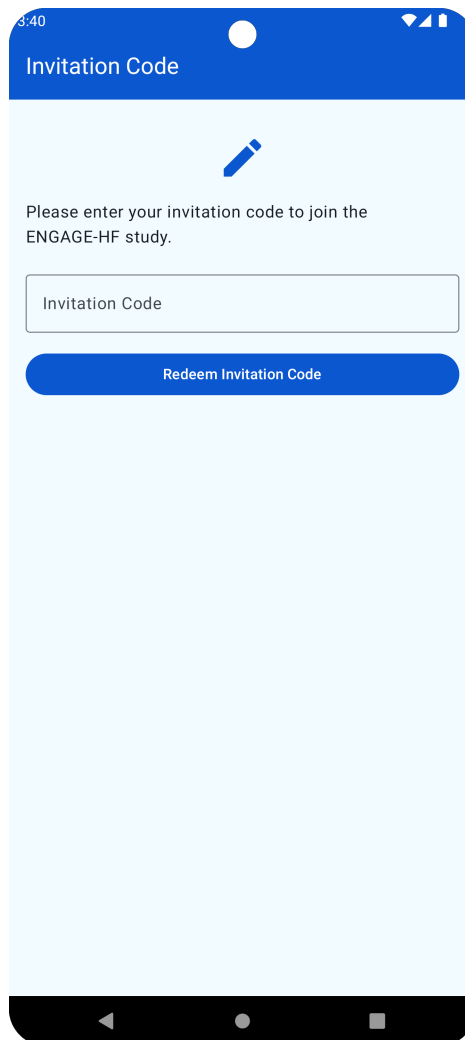


Figure 4.5.: Invitation Code View.

```
1 class EngageInvitationCodeRepository @Inject constructor(  
2     private val navigator: Navigator,  
3 ) : InvitationCodeRepository {  
4  
5     override fun getScreenData(): InvitationCodeScreenData {  
6         return InvitationCodeScreenData(  
7             title = "Invitation□Code",
```

4. Technical details on implementation

```
8         description = "Please enter your invitation code to join  
9         the ENGAGE-HF study.",  
10        redeemAction = { navigator.navigateTo(  
11            AccountNavigationEvent.LoginScreen(false)) },  
12        gotAnAccountAction = { navigator.navigateTo(  
13            AccountNavigationEvent.LoginScreen(true)) }  
    )  
}
```

Source text 4.24: EngageInvitationCodeRepository.

In addition to a title and a description, you can also specify which code should be executed when the invitation code has been successfully redeemed (`redeemAction`) or which action should take place if the user indicates that they already have an account (`gotAnAccountAction`). This means that the user's needs can be customised and they can be directed to any subsequent pages.

The `InvitationAuthManager` interface (Listing: 4.25) is responsible for managing the validation of invitation codes.

```
1 interface InvitationAuthManager {  
2     suspend fun checkInvitationCode(invitationCode: String): Result<  
3         Unit>  
}
```

Source text 4.25: InvitationAuthManager Interface.

Currently, the `FirestoreInvitationAuthManager`¹⁷ serves as the default implementation, provided through dependency injection. However, it is designed to be replaceable by alternative implementations if needed.

In the default implementation when the invitation code is redeemed, a Firebase function is executed that links an anonymous Firebase account assigned to the user in this view with the code entered. If the user is e.g. on the registration page at a later point in time, a pre-register function in Firebase ensures that before the registration is executed, it is checked whether the user has correctly linked their anonymous account to an invitation code before the anonymous account is converted into a fully-fledged account with access data. This ensures that even if someone found a possibility to bypass the screen he still wouldn't be able to bypass the second verification of the invitation code.

¹⁷<https://github.com/StanfordSpezi/SpeziKt/blob/main/modules/account/src/main/kotlin/edu/stanford/spezi/module/account/manager/FirebaseInvitationAuthManager.kt>

Welcome Onboarding

This view (Figure: 4.6) is necessary for providing users with an initial orientation, helping them to quickly grasp the main features and sections of the application. It ensures that users do not feel lost and can navigate the application more confidently from the beginning [58].

The Welcome Onboarding View introduces new users to the application. It provides a brief overview of the key areas they need to be familiar with and helps them understand what to expect as they proceed.

Features:

- **Area Descriptions:** Icons and brief descriptions for different areas of the application, helping users to get a quick understanding.
- **Learn more Button:** Moves the user to the next step in the onboarding process.

The onboarding screen is also similar to the `InvitationCodeScreen` architecture listing 4.24: The user of the framework only has to implement an interface via which the required information is provided. This includes defining which areas should be displayed and which actions should be executed when the `Continue`-button is pressed.

Sequential Onboarding

This view is important for ensuring a comprehensive onboarding experience. By breaking down the onboarding process into sequential steps, it ensures that users can absorb information at their own pace and do not feel overwhelmed. It also ensures that all necessary information is covered systematically, enhancing user understanding and engagement [58].

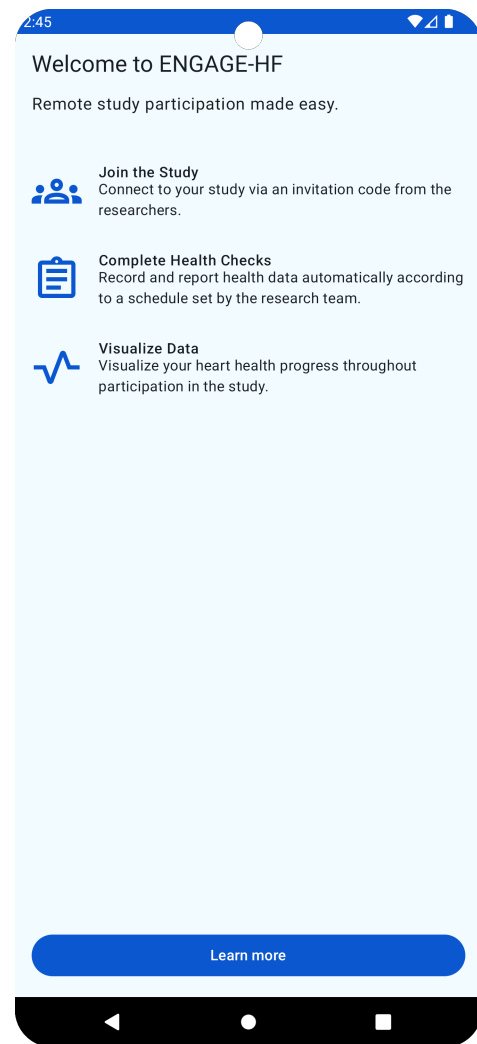


Figure 4.6.: Onboarding Screen.

4. Technical details on implementation

The Sequential Onboarding View (Figure: 4.7) guides users through a step-by-step process to complete their onboarding. Each step provides detailed information and instructions, ensuring users do not miss any critical parts of the onboarding.

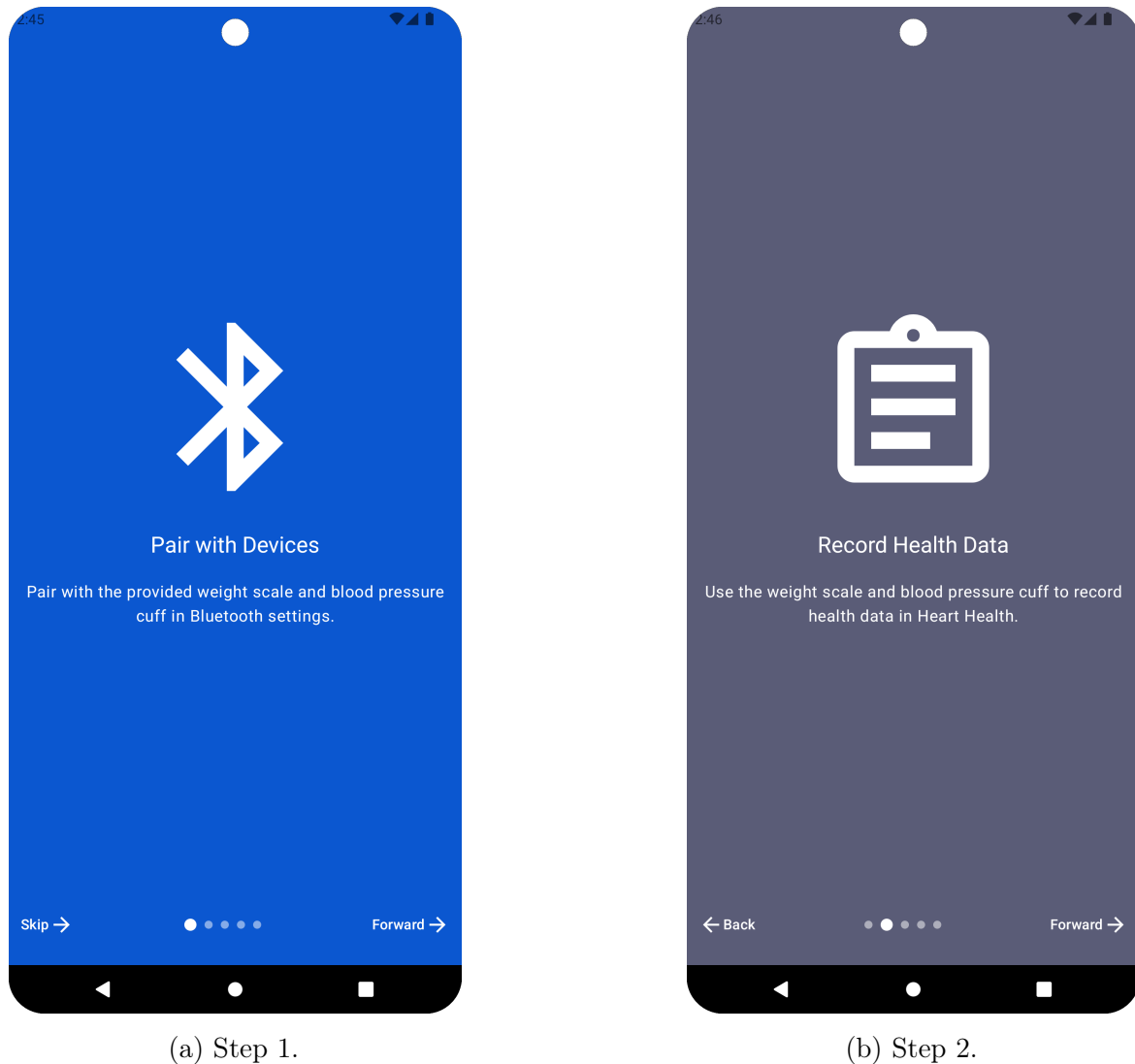


Figure 4.7.: Sequential Onboarding View.

Features:

- **Step Indicators:** Shows which step the user is currently on and how many steps remain.
- **Skip and Forward Buttons:** Allows users to move forward or skip steps if they are already familiar with the information.

- **Central Icon and Description:** Provides a visual and textual description of the current step.

Storage

The storage module is a crucial component in the Spezi framework, designed to handle key-value and file storage securely and efficiently. This module provides a wrapper around default storage implementations to enhance ease of use and security, ensuring robust data management for mHealth applications.

Default storage implementations in Android, such as **SharedPreferences** for key-value storage and standard file I/O for file storage, are widely used but have several limitations:

- **Security:** Default implementations often lack built-in encryption, leaving sensitive health data vulnerable¹⁸.
- **Usability:** Interacting directly with these storage mechanisms can be cumbersome, requiring repetitive boilerplate code for common operations like reading, writing, and deleting data¹⁸.
- **Consistency:** Ensuring consistent access patterns and error handling across different parts of the application can be challenging when using the default storage solutions directly¹⁸.

To address these limitations, we implemented a custom wrapper around the default storage mechanisms, offering the following advantages:

- **Enhanced Security:**
 - **EncryptedFileStorage:** For file storage, this implementation ensures that all files are encrypted using industry-standard encryption algorithms before being written to disk. This prevents unauthorized access to sensitive health data.
 - **EncryptedSharedPreferencesStorage:** This implementation provides encrypted key-value storage, ensuring that all stored preferences are securely encrypted.

¹⁸<https://developer.android.com/codelabs/android-preferences-datastore>

4. Technical details on implementation

- **Improved Usability:** The wrapper provides a simplified and consistent API for common storage operations, reducing the need for boilerplate code by offering straightforward function calls like `saveData`, `readData`, and `deleteData` through the `KeyValueStorage` and `FileStorage` interfaces, abstracting away the complexity of dealing with underlying storage mechanisms.
- **Consistency and Error Handling:** By centralizing storage operations in a single module, we ensure consistent access patterns and error handling throughout the application. This approach reduces the likelihood of bugs and makes the codebase easier to maintain. Additionally, the use of `Flow` in `KeyValueStorage` enables reactive programming, allowing the application to respond to changes in stored data in real-time. This capability is particularly useful for health monitoring applications where timely updates are crucial.
- **Flexibility in Implementation:** The shared `KeyValueStorage` interface allows seamless switching between encrypted and non-encrypted implementations, providing developers the flexibility to choose the appropriate storage solution based on the security requirements of different data types. Each implementation is designed to handle the same inputs, ensuring compatibility and ease of integration. For instance, while `EncryptedSharedPreferences` does not natively support `ByteArray`, this functionality was added to support the full feature set of `DataStore`, but with encryption on top.

After the module (`:modules:storage`) has been added to the `build.gradle` file, the desired storage implementation can be provided via Dependency Injection (DI) and then only one of the desired interfaces needs to be injected:

- `EncryptedFileStorage` for the `FileStorage` interface
- `EncryptedSharedPreferencesStorage` or `LocalStorage` for the `KeyValueStorage` interface

Listing 4.26 shows the `KeyValueStorage` Interface.

```
1 interface KeyValueStorage {
2     suspend fun <T : Any> saveData(key: PreferenceKey<T>, data: T)
3     fun <T> readData(key: PreferenceKey<T>): Flow<T?>
```

```

4     suspend fun <T> readDataBlocking(key: PreferenceKey<T>): T?
5     suspend fun <T> deleteData(key: PreferenceKey<T>)
6 }

```

Source text 4.26: KeyValueStorage Interface.

How the KeyValueStorage interface can be used is shown in listing 4.27.

```

1 val stringKey = PreferenceKey.StringKey("user_name")
2 keyValueStorage.saveData(stringKey, "test_user_name")
3 keyValueStorage.readDataBlocking(stringKey)?.let {
4     println("Read_string_data_blocking:_$it")
5 }
6 keyValueStorage.deleteData(stringKey)

```

Source text 4.27: KeyValueStorage Usage.

It is also possible to use the Flow interface to observe changes as shown in listing 4.28.

```

1 val job = launch {
2     keyValueStorage.readData(stringKey).collect { data: String? ->
3         println("Read_string_data:_$data")
4     }
5 }

```

Source text 4.28: KeyValueStorage Flow Usage.

The FileStorage provides a simple interface for storing and retrieving files as shown in listing 4.29.

```

1 interface FileStorage {
2     suspend fun readFile(fileName: String): ByteArray?
3     suspend fun deleteFile(fileName: String)
4     suspend fun saveFile(fileName: String, data: ByteArray)
5 }

```

Source text 4.29: FileStorage Interface.

It can be used as shown in listing 4.30.

```

1 val fileName = "testFile.data"
2 val data = "Hello, Stanford!".toByteArray()
3 fileStorage.saveFile(fileName, data)
4 val readData = fileStorage.readFile(fileName)
5 readData?.let {
6     println("Read_file_data:_${String(it)}")
7 }
8 fileStorage.deleteFile(fileName)

```

Source text 4.30: FileStorage Usage.

4. *Technical details on implementation*

The addition of the custom storage module in the Spezi framework significantly enhances the security (NFR7), usability, and consistency (NFR6.2) of data management. By providing encrypted storage solutions and a simplified API, the framework ensures that sensitive health data is handled securely and efficiently, aligning with the project's goals of creating a modular, scalable, and reusable platform for mHealth applications.

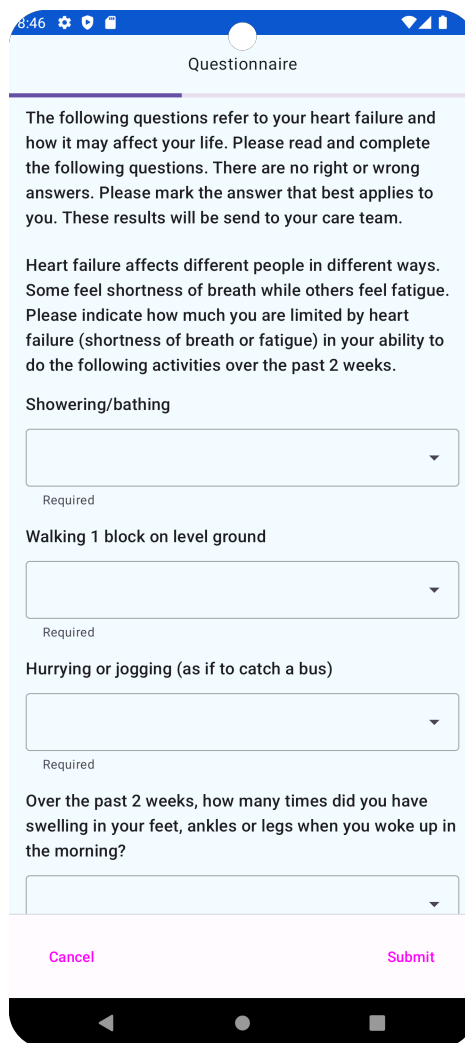
By implementing this storage module, the Spezi framework not only addresses the limitations of default storage solutions but also sets a new standard for secure and efficient data management in mobile health applications. The flexibility to switch between encrypted and non-encrypted implementations via the shared `KeyValueStorage` interface and the enhanced compatibility for handling `ByteArray` inputs in `EncryptedSharedPreferences` further illustrate the robustness and adaptability of this solution.

Questionnaire

In the realm of mHealth applications, accurate and efficient symptom capture is crucial for both patient management and clinical decision-making. One such method for capturing patient-reported outcomes is through the use of digital questionnaires that adhere to the FHIR standard. This chapter discusses the implementation of a symptom-capturing questionnaire, focusing on the technical integration challenges and solutions, particularly concerning the use of the FHIR Data Capture library in a Jetpack Compose-based application. For a detailed example of how this integration looks in practice, please refer to figure 4.8. The FHIR Data Capture library provides a robust solution for integrating FHIR-compliant questionnaires into Android applications [59]. It facilitates the loading, presentation, and submission of questionnaires that adhere to FHIR standards, enabling seamless integration with healthcare systems that utilize FHIR for data exchange [59] [60]. For an example of how input errors and answering questions is handled refer to figure 4.9.

However, the FHIR Data Capture library is designed with traditional Android UI components in mind, specifically utilizing Fragments [60]. Jetpack Compose, the modern toolkit for building native UI in Android, operates differently, relying on Composables rather than Fragments [60]. This divergence presents a significant challenge when attempting to embed FHIR-based questionnaires within a Compose-centric application architecture [59].

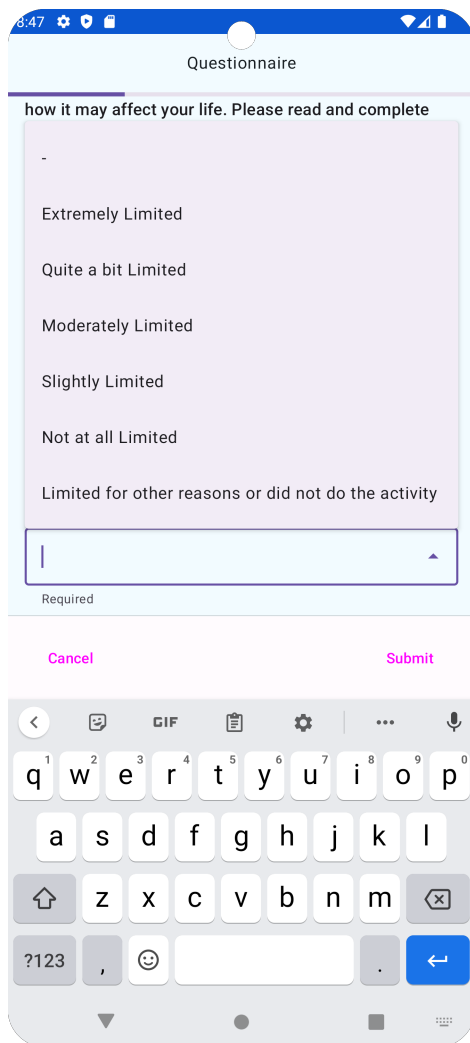
To address the incompatibility between the FHIR Data Capture library and Jetpack Compose, a hybrid approach was adopted. The `AndroidFragment`



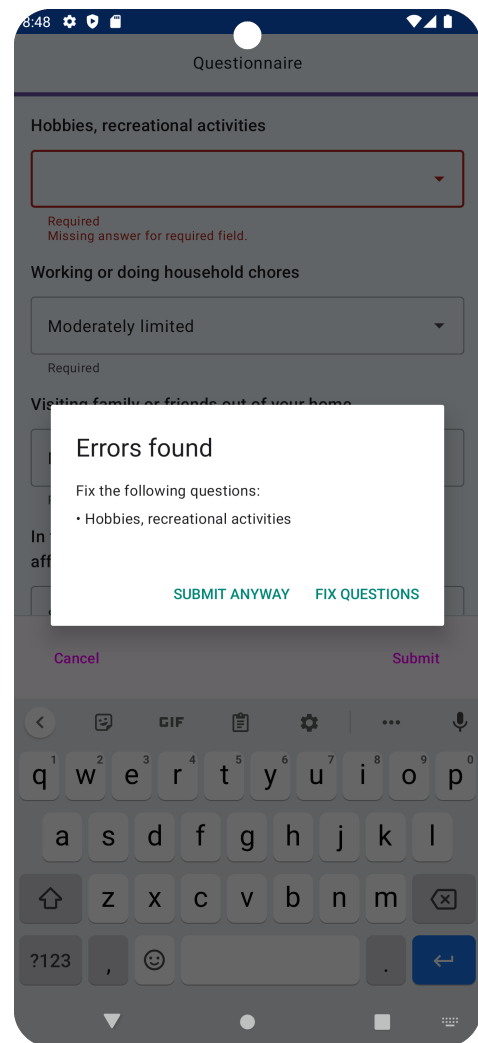
The screenshot shows a mobile application interface for a questionnaire. The title bar is blue with the text "Questionnaire". Below the title bar, there is a light blue background with the following text: "The following questions refer to your heart failure and how it may affect your life. Please read and complete the following questions. There are no right or wrong answers. Please mark the answer that best applies to you. These results will be send to your care team." Below this, there is another paragraph: "Heart failure affects different people in different ways. Some feel shortness of breath while others feel fatigue. Please indicate how much you are limited by heart failure (shortness of breath or fatigue) in your ability to do the following activities over the past 2 weeks." There are three dropdown menus, each with a downward arrow and the word "Required" below it. The first dropdown is labeled "Showering/bathing", the second is labeled "Walking 1 block on level ground", and the third is labeled "Hurrying or jogging (as if to catch a bus)". Below the third dropdown, there is a question: "Over the past 2 weeks, how many times did you have swelling in your feet, ankles or legs when you woke up in the morning?" followed by a fourth dropdown menu. At the bottom of the screen, there are two buttons: "Cancel" on the left and "Submit" on the right. The bottom of the screen shows the Android navigation bar.

Figure 4.8.: Questionnaire.

4. Technical details on implementation



(a) Choice Selection.



(b) Input Error.

Figure 4.9.: FHIR Data Capture Functionality.

Composable was utilized to embed a `QuestionnaireFragment` within the Jetpack Compose UI [60]. This solution allows the app to leverage the FHIR Data Capture library's existing functionality while maintaining consistency with the Compose architecture. The integration is achieved as in listing 4.31.

```
1 AndroidFragment<QuestionnaireFragment>(
2     fragmentState = fragmentState,
3     modifier = Modifier
4         .fillMaxSize(),
5     arguments = uiState.args
6 ) { fragment ->
```

```

7     fragment.setFragmentManagerListener(
8         QuestionnaireFragment.SUBMIT_REQUEST_KEY
9     ) { _, _ ->
10         onAction(
11             QuestionnaireViewModel.Action.SaveQuestionnaireResponse(
12                 fragment.getQuestionnaireResponse()
13             )
14         )
15     }
16     fragment.setFragmentManagerListener(
17         QuestionnaireFragment.CANCEL_REQUEST_KEY
18     ) { _, _ ->
19         onAction(
20             QuestionnaireViewModel.Action.Cancel
21         )
22     }
23 }

```

Source text 4.31: QuestionnaireFragment implementation with AndroidFragment.

This code snippet in listing 4.31 demonstrates the embedding of a `QuestionnaireFragment` within a Jetpack Compose Composable. The fragment is responsible for managing the questionnaire’s lifecycle and interactions, while the Compose-based UI ensures seamless integration with the rest of the app’s user interface.

Navigating to the questionnaire screen within a app using the Spezi framework is straightforward, leveraging the existing Navigation approach already provided by the Spezi framework. The navigation to the questionnaire is managed via the `QuestionnaireScreen` route, which is defined as in listing 4.32.

```

1 @Serializable
2 data class QuestionnaireScreen(val questionnaireId: @Serializable
   String) : Routes()

```

Source text 4.32: FileStorage Interface.

This route allows for easy and consistent navigation within the app, using the familiar Routes pattern [61]. This approach not only simplifies the user experience but also maintains the modular and reusable nature of the app’s architecture. See Navigation for more details.

The `QuestionnaireViewModel` relies on a `QuestionnaireRepository` interface, which provides the necessary methods for loading and saving questionnaire data. This repository interface can be easily swapped with

4. Technical details on implementation

any custom implementation, allowing for flexibility in how and where the data is managed.

By default, the app provides an implementation that interacts with Firestore, loading questionnaires from the database and storing the responses back in Firestore. This default implementation ensures that a app is ready to work out-of-the-box with Firestore, when using the Spezi Framework but developers can easily provide alternative implementations if they wish to use a different backend or data source.

During implementation, a known issue with nested scrolling inside fragments embedded within a draggable Compose component was encountered¹⁹. Specifically, this bug prevented the questionnaire component from being placed within a bottom sheet, as intended, due to scrolling conflicts¹⁹. As a workaround, the questionnaire was navigated to on a separate screen rather than being embedded in a bottom sheet. This solution, while not ideal, ensured that the framework and possible app's Single Activity Architecture remained intact, thus preserving the modularity and consistency of the user experience.

The chosen implementation approach offers several advantages:

- **Maintains Architectural Consistency:** By embedding the `QuestionnaireFragment` within a Composable, the app adheres to its Single Activity Architecture, which simplifies navigation and state management across the application.
- **Utilizes FHIR Standards:** Leveraging the FHIR Data Capture library ensures that the questionnaire is compliant with widely accepted healthcare standards, facilitating interoperability with other healthcare systems.
- **Modular Integration and Flexibility:** The use of fragments within Compose allows for modular integration of complex UI components that may not yet be fully supported by Compose. Additionally, the ability to swap out the `QuestionnaireRepository` interface with custom implementations provides flexibility in data management, making the app adaptable to different backend systems and workflows.
- **Seamless Navigation:** The `QuestionnaireScreen` route provides an intuitive and consistent method for navigating to questionnaires

¹⁹<https://issuetracker.google.com/issues/277651136?pli=1>

within the app, enhancing the overall user experience while adhering to established navigation patterns.

The integration of FHIR-compliant questionnaires into the Spezi Framework exemplifies the challenges and solutions involved in adapting traditional Android libraries to modern UI paradigms like Jetpack Compose. By embedding a fragment within a Composable, we successfully navigated the limitations of the current libraries while maintaining a cohesive and modular application architecture. This approach not only preserves the integrity of the framework's design but also ensures that it remains compliant with healthcare interoperability standards, thereby enhancing its utility in clinical settings. The flexible navigation and customizable data handling further strengthen the framework's adaptability, making this modul a robust tool for managing patient symptoms in real-world healthcare environments.

5. Case Study: ENGAGE-HF for Heart Failure Management

The development of the ENGAGE-HF prototype represents a significant milestone in the creation of the Spezi Framework. This is the first time the Android framework has been used. This chapter details the comprehensive process undertaken to design, architect, and implement the ENGAGE-HF application, leveraging the modular and reusable Spezi framework.

In this chapter, we will explore the key design principles and architectural considerations that guided the development of the ENGAGE-HF prototype. We will discuss the utilization of the Spezi framework, highlighting its modular components and the customizability that allows for adaptation to various clinical and research scenarios. Additionally, the implementation and deployment strategies will be examined, showcasing how the app ensures security, scalability, and maintainability.

By detailing the development process, this chapter aims to provide a clear understanding of the technical and design methodologies employed in creating an effective mHealth solution for heart failure management. This serves as an example of how modern frameworks and technologies can be harnessed to develop impactful healthcare applications, ultimately contributing to improved patient outcomes and more efficient healthcare delivery.

Design and architecture of the app

The ENGAGE-HF prototype, utilizes the Spezi framework to support heart failure management. The design and architecture of the ENGAGE-HF app are driven by key principles of modularity, scalability, and user-centricity, ensuring both flexibility and ease of use.

User Interface (UI) and User Experience (UX) Design Principles

The ENGAGE-HF app prioritizes a user-friendly interface, with a clean and intuitive design aimed at enhancing patient engagement by the usage of the Spezi Framework. Key UI/UX features include:

- **Consistent Design Language:** Utilizes the cohesive design system from Spezi Framework with consistent theming, including light and dark modes and customizable color schemes.
- **Accessible Components:** Incorporates accessible UI components optimized for ease of use, including buttons, forms, and navigation elements that adhere to accessibility standards.
- **Educational Content Integration:** Provides patients with educational materials via a dedicated section, integrating YouTube videos and other multimedia resources.
- **Health Data Visualization:** Features graphical and list-based representations of health data, such as weight, blood pressure, and heart rate, to promote user understanding and engagement.
- **Medication Management:** The app provides a comprehensive overview of current medications. Each medication category has its own tabs that provide educational information about the medication, including dosing guidelines, target doses and the patient's current doses. In addition, the app displays the patient's current medication status and indicates whether a higher dose is possible or the maximum dose has already been reached
- **Home Screen Overview:** The Home Screen provides an overview of notifications that display important events such as medication changes, significant weight gains and authorisation for medication changes. These notifications can be expanded for more information and removed after reading. The to-do list tracks daily or weekly tasks, such as watching a welcome video, setting personal notifications, taking measurements (blood pressure, weight) and reviewing health summaries before visits. The app displays the latest vital signs if they have not been completed for the day.

Security Architecture and Data Privacy Measures

The ENGAGE-HF app is designed with robust security architecture to ensure the protection of sensitive patient information:

- **Authentication Mechanisms:** Implements secure authentication methods, including username/password, and social sign-ins (e.g., Google), ensuring only authorized users access the app.
- **Data Encryption:** Utilizes encryption protocols for data transmission and storage to safeguard patient data.

Utilization of the SPEZI Framework

The development of the ENGAGE-HF app showcases the versatility of the Spezi framework, demonstrating its capability to integrate various functionalities essential for managing heart failure.

Framework Modules and Integration

- **Authentication Module:** Provides a secure login system, utilizing Firebase for authentication and user management, supporting both anonymous and social sign-ins. More on this in chapter 5.
- **Bluetooth Connectivity Module:** Ensures seamless integration with medical devices, allowing the app to connect directly to Bluetooth-enabled weight scales and blood pressure monitors. This module includes automatic device pairing and real-time data collection.
- **FHIR Module:** The module manages the processing, of health data. It supports encoding health measurements in the FHIR (Fast Healthcare Interoperability Resources) format and uploading them to the Firebase database. Additionally, the FHIR module is used to map health records to FHIR Observations and vice versa, ensuring interoperability and standardization of health data.
- **UI/UX Module:** Offers reusable components built with Jetpack Compose for Android, facilitating the creation of a consistent and accessible user interface.

Implementation and Deployment

- **Open-Source Development:** The ENGAGE-HF app is developed as an open-source project on GitHub, ensuring transparency and community collaboration.
- **CI/CD Integration:** Utilizes GitHub Actions for continuous integration and continuous deployment (CI/CD), automating testing, building, and deployment processes to ensure the app remains up-to-date and functional.

Education

The Education tab (Figure: 5.1) of the ENGAGE-HF application features a well-structured layout that adheres to best practices for video content delivery and UI/UX design. Initially, users see a preview image for each video category, created using Coil, an image loading library for Android that supports efficient image handling¹. This design choice ensures faster load times, reducing the initial load burden and optimizing the user experience by loading only essential data at first¹. Each video category such as Medication Videos or Vitals Videos can be expanded or collapsed using a simple drop-down mechanism. This minimizes visual clutter on the main page, making it easier for users to focus on relevant content². Once expanded, users can browse through available videos, previewing their titles and thumbnail images, but the video itself is only displayed when clicked. This opens a new Video Detail Page (Figure: 5.2), where the selected video is embedded via YouTube integration.

¹<https://coil-kt.github.io/coil/>

²<https://dribbble.com/resources/education/ui-design-principles>

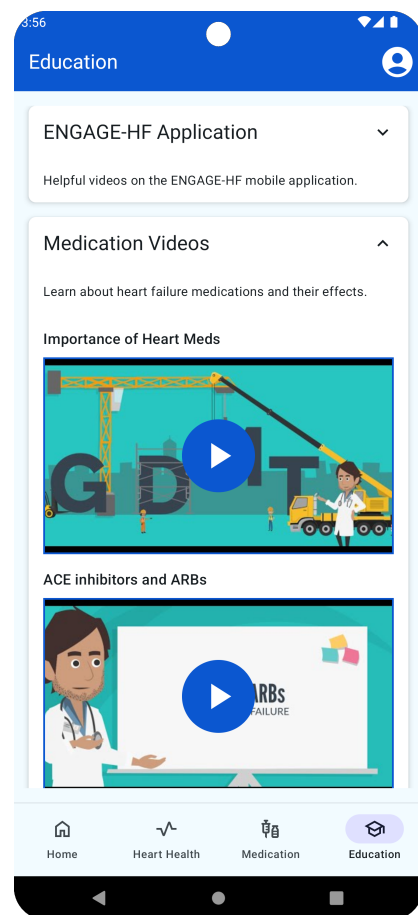


Figure 5.1.: Education Screen Overview.

This page presents detailed information alongside the video, allowing the user to consume educational content in a focused manner. This modular design can be reused for other functions in the application, such as the message action feature in message section 5. By using the same navigation structure, specific actions like `/videoSection/1/video/1` could directly link to a particular video, allowing seamless integration of video content throughout different parts of the app. This could also be applied to notifications, directing users straight to relevant video content based on their health data.

This approach aligns with modern mobile UI best practices by ensuring that the system remains both performant and user-friendly. By only loading essential information like previews at first, it reduces memory and CPU consumption³. In addition, the use of lazy loading strategies—where full media content like YouTube videos are only loaded when needed—improves the overall responsiveness of the application. Research also supports that progressive disclosure, where users are presented with only the necessary amount of information upfront, improves engagement and retention, particularly in health-related apps [62].

Moreover, this modularity ensures that the app's components are reusable, making future development more efficient and adaptable to changes or additional features.

Although the implementation of this education page is not a direct component of the Spezi framework, it nevertheless represents a useful extension. It adds the function: the provision of educational content for patients.

In almost all mHealth applications, especially those that treat chronic diseases such as heart failure, knowledge transfer plays a crucial role [63]. Studies show that access to relevant information improves patient self-

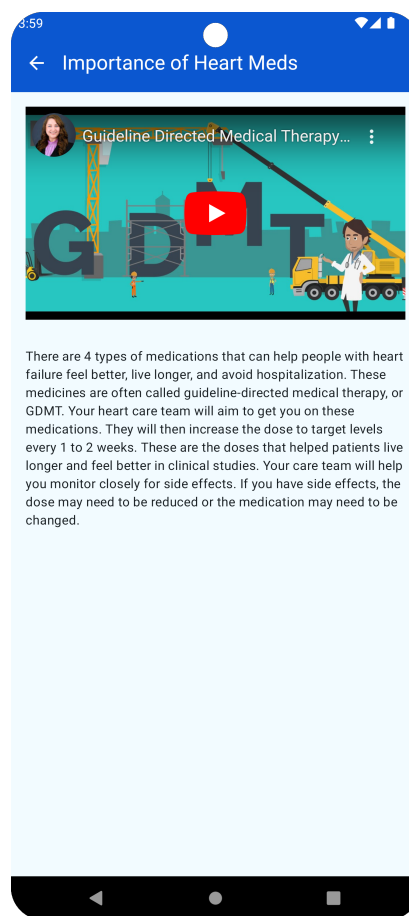


Figure 5.2.: Education Screen Single Video.

³<https://appsgeyser.com/blog/optimizing-app-performance-best-practices-for-enhanced-user-experience/>

5. Case Study: *ENGAGE-HF for Heart Failure Management*

management, increases compliance and ultimately leads to better health outcomes [63]. As many of these applications support patients to actively participate in their health management, the integration of educational resources is an almost indispensable function.

Extending the Spezi framework to include this module that provides educational content such as videos and interactive learning materials would not only be a logical development, but would also promote reusability. As most mHealth apps include some form of knowledge transfer, a standardised solution could help to minimise the effort required to implement such features while improving the user experience. This could easily be integrated into other areas of apps, as described earlier, such as push notifications or direct links to specific content (e.g. via message actions like `/videoSection/1/video/1`).

Heart Health

The Heart Health Screen (Figure: 5.3) is a central feature designed to empower patients by providing them with clear, accessible visualizations of their health metrics. This includes data on weight, heart rate, blood pressure, and symptoms, all of which are crucial for effective heart failure management. The implementation of this feature is grounded in a highly extensible and modular architecture, leveraging Kotlin's sealed interfaces to streamline data handling and ensure scalability.

In addition to the ability to add measurements via Bluetooth-connected devices, the app also provides users with the option to enter their health data manually through a dedicated dialog shown in figure 5.4. This ensures that patients can always keep their records up to date, even if they do not have access to the relevant medical devices at the time.

The interface organizes health data into several tabs, each focused on a specific metric:



Figure 5.3.: Heart Health Screen.

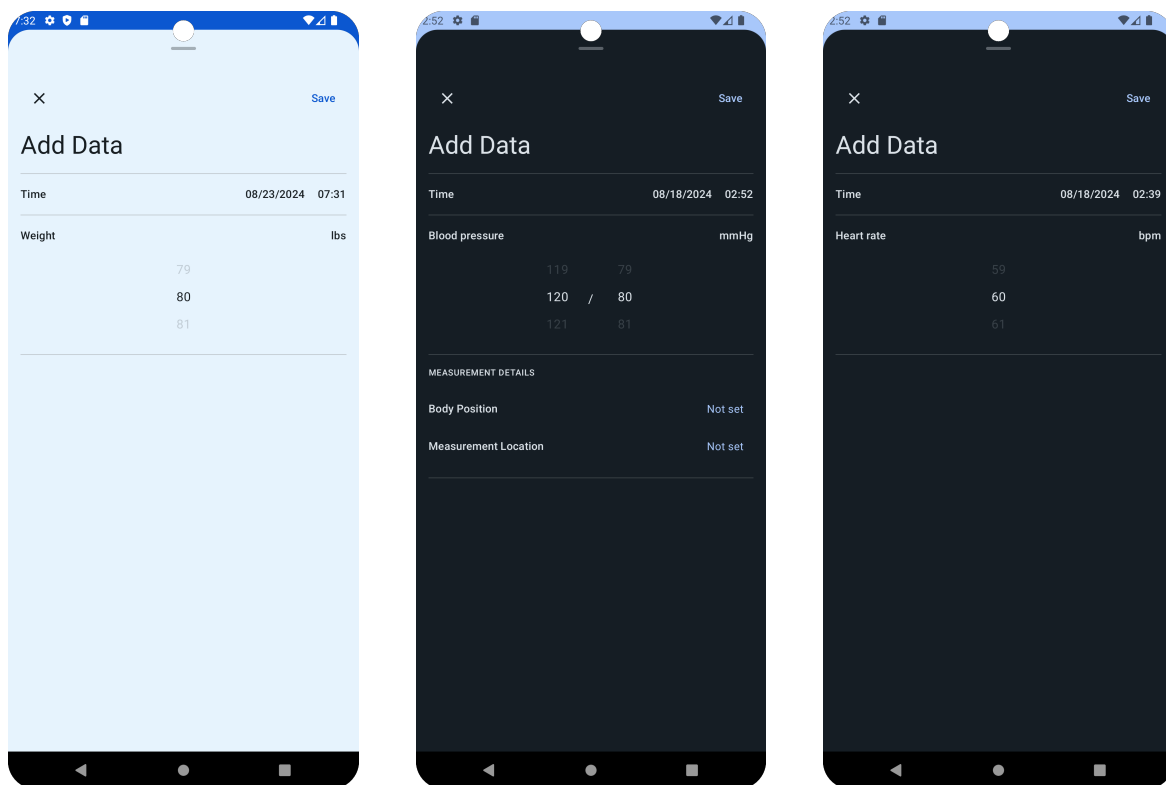
- **Symptoms:** Displays symptom data collected through questionnaires, offering trend visualization to track changes over time.
- **Weight:** Graphically represents weight data, enabling patients to monitor fluctuations, which are critical indicators in heart failure management.
- **Blood Pressure:** Blood pressure readings are shown through both charts and historical lists, allowing for easy monitoring of cardiovascular health.

5. Case Study: ENGAGE-HF for Heart Failure Management

- **Heart Rate:** Heart rate data is also visualized in a graph, with historical data listed below to help identify patterns and trends.

The UI is designed to enhance patient engagement by making health data more understandable and actionable through clear visual elements.

In the development of the Heart Health feature, the Health Connect on FHIR module from the framework was utilized to map saved FHIR Observations back into Health Connect Records. This reverse mapping allowed for the accurate display of previously saved health metrics, such as heart rate, blood pressure, and weight, within the app. By leveraging this functionality, the app ensures that data collected in the FHIR format can be seamlessly transformed and presented in a format compatible with Android's Health Connect, enhancing the user experience and data interoperability.



(a) Add Weight Data - Light Mode.

(b) Add Blood Pressure Data - Dark Mode.

(c) Add Heart Rate Data - Dark Mode.

Figure 5.4.: Add Data Bottom Sheets.

At the heart of this extensibility is the sealed interface `EngageRecord`, a pivotal element in the app's architecture shown in listing 5.1. This

interface defines a standardized way to handle different types of health records, ensuring that the app can be easily extended to support new data types in the future.

```
1 private sealed interface EngageRecord {
2     val record: Record
3
4     data class Weight(override val record: WeightRecord) :
5         EngageRecord
6     data class BloodPressure(override val record: BloodPressureRecord
7         ) : EngageRecord
8     data class HeartRate(override val record: HeartRateRecord) :
9         EngageRecord
10
11     val zonedDateTime: ZonedDateTime
12     get() = when (this) {
13         is Weight -> record.time.atZone(record.zoneOffset)
14         is BloodPressure -> record.time.atZone(record.zoneOffset)
15         is HeartRate -> record.startTime.atZone(record.
16             startZoneOffset)
17     }
18
19     val clientId get() = record.metadata.clientRecordId
20
21     companion object {
22         fun from(record: Record) = when (record) {
23             is WeightRecord -> Weight(record = record)
24             is BloodPressureRecord -> BloodPressure(record = record)
25             is HeartRateRecord -> HeartRate(record = record)
26             else -> error("Unsupported record type: ${record::
27                 javaClass.name}")
28         }
29     }
30 }
```

Source text 5.1: Sealed Interface EngageRecord.

The `EngageRecord` interface serves as a unified abstraction for various health records like `WeightRecord`, `BloodPressureRecord`, and `HeartRateRecord`. By using a sealed interface, all possible data types are known at compile time, which enhances type safety and ensures that all record types are accounted for in the app's logic. This design allows the app to handle different health metrics uniformly, enabling the seamless integration of new data types with minimal changes to the codebase.

5. Case Study: *ENGAGE-HF* for Heart Failure Management

When the app processes health data, it uses the `EngageRecord.from(record)` method to convert raw health data into a corresponding `EngageRecord` subtype. This abstraction allows the rest of the app to interact with a consistent interface, regardless of the specific health metric being processed.

Functions like `mapUiStateTimeRange` use this unified interface to group, filter, and visualize health data. The `sealed interface` ensures that the code can handle each type of health record appropriately, whether displaying a graph of weight trends or a table of blood pressure readings. Should a new health metric, such as `GlucoseRecord`, need to be integrated, this can be done simply by adding a new data class that implements `EngageRecord`. The companion object within `EngageRecord` can then be updated to handle this new type, ensuring that it is seamlessly integrated into the existing visualization and data processing logic.

The use of sealed interface `EngageRecord` directly supports the goal of creating a scalable and flexible system. This design approach allows for:

- **Efficient Scalability:** New health metrics can be integrated without overhauling the existing code, ensuring that the app can evolve with the needs of patients and healthcare providers.
- **Consistent User Experience:** Regardless of the number or types of health metrics added, the app maintains a consistent user experience, with new data types being seamlessly incorporated into the existing UI framework.

In conclusion, the sealed interface `EngageRecord` is a cornerstone of the architecture of this module, enabling the effective visualization and management of diverse health metrics. By leveraging this approach, the app not only meets current patient needs but is also well-positioned to adapt to future advancements in health monitoring and mHealth technologies. This flexibility and extensibility are key to the app's potential for ongoing relevance and impact in the field of heart health management.

Bluetooth Devices

The integration process of Bluetooth devices in the *ENGAGE-HF* app was straightforward due to the Bluetooth module (Section: 4) of the

Spezi framework. Leveraging components like the `BLEDeviceScanner` and `BLEDeviceConnector`, the app was able to quickly pair with both a weight scale and a blood pressure monitor. This modularity made it easy to handle the entire lifecycle of BLE-connections from scanning for nearby devices to connecting, retrieving data, and then mapping it to the app's UI. The `MeasurementMapper` played a crucial role by converting raw BLE data into specific health measurements, such as weight and blood pressure, without requiring extensive reconfiguration.

The primary challenge was processing and handling the measurement data. However, this was simplified by the use of `StateFlows`, which enabled the real-time updating of the app's state based on incoming data. The integration with Kotlin coroutines further improved asynchronous handling, allowing the app to perform data collection in the background without blocking the user interface. For instance, after receiving a new measurement from the weight scale or blood pressure monitor, the app displayed a dialog to the user, confirming the new values. This dialog-driven approach simplified user interaction: the user merely had to confirm the measurement, and no further actions were required on their part.

The `ViewModel` converts these confirmed values into FHIR Observations using the `HealthConnectOnFHIR` module. Those Observations are saved to the Firestore database, which was easily integrated with the framework due to its existing compatibility with Google Cloud Firebase. The framework's architecture allowed the `ViewModel` to handle the logic of what happens after the user confirms the values, minimizing the need for custom implementation.

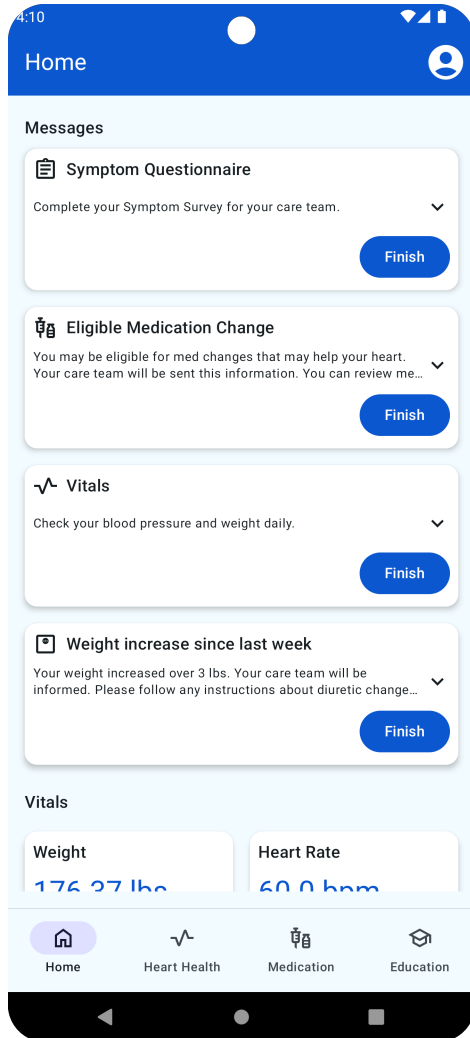
This modular approach significantly reduced the development complexity. Instead of focusing on the intricacies of Bluetooth communication, data conversion or backend integration, we could focus on more critical aspects, such as improving the user experience.

Messages

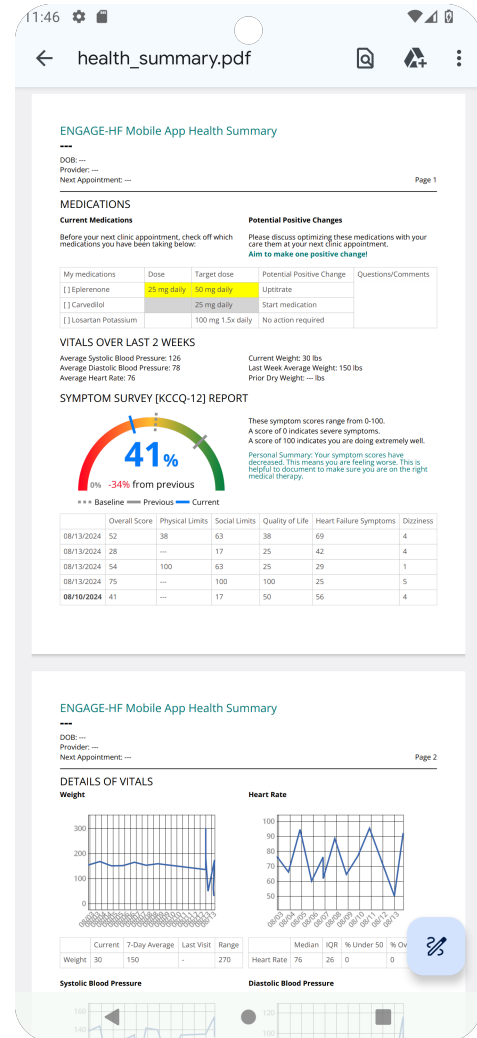
The messaging system in the ENGAGE-HF app plays a crucial role in improving patient engagement by delivering targeted notifications based on each patient's health status and treatment plan. These messages are designed to keep patients informed and prompt them to take action when necessary, such as filling out symptom questionnaires or reviewing updated

5. Case Study: ENGAGE-HF for Heart Failure Management

medication information. The messages are triggered by specific events or data changes and are intended to ensure timely patient responses to important health-related tasks.



(a) Messages screen with notifications for welcome, weight, medication, pre-visit, and symptom updates.



(b) Health Summary PDF showing medications, vitals trends, and symptom survey results.

Figure 5.5.: ENGAGE-HF app's Messages screen and Health Summary PDF for patient updates and health reports.

As shown in the screenshot (Figure: 5.5a), messages in the app fall into different categories, each tied to an actionable item for the patient. Some of the key Message Types include⁴:

⁴<https://github.com/StanfordBDHG/ENGAGE-HF-Firebase>

- **MedicationChange:** Triggered when a medication request is updated. The patient is notified to review and take action regarding their medication plan.
- **WeightGain:** Sent when a new body weight observation shows a weight increase of more than 3 pounds compared to the previous week's median, but only once every seven days.
- **MedicationUptitration:** Triggered when the patient's medication recommendations are updated, typically every two weeks.
- **Welcome:** Displayed when a new user account is created. It serves as an introductory message directing the patient to educational materials or videos.
- **Vitals:** These are daily notifications reminding the patient to log their vital signs (blood pressure, weight), especially when data for the current day is missing.
- **SymptomQuestionnaire:** Sent every 14 days, this message prompts the patient to complete a symptom questionnaire.
- **PreAppointment:** Sent 24 hours before an upcoming appointment to remind the patient of the visit.

Each message contains a due date to encourage timely action, and the system allows for marking messages as complete once the required action is performed. There are also various actions associated with messages, providing a seamless way for patients to interact with the content⁴. For instance, in addition to **HealthSummaryAction** where a pdf gets downloaded and shown (Figure: 5.5b), some messages guide the patient to educational resources⁴. For example, the Welcome message or any other message with the path `videoSections$videoSectionId$videos$videoId$` directs users to the Education Video Detail Page (Figure: 5.2), where they can watch relevant videos about using the app or managing their health⁴. This functionality helps patients access critical information directly through the app interface, reinforcing the system's educational component. It also highlights the adaptability of the ENGAGE-HF app in addressing patient needs through clear, structured communication.

5. Case Study: ENGAGE-HF for Heart Failure Management

In addition to the Message logic, Firebase Cloud Messages (FCM) is also supported. Notifications from FCM are handled by reusing the same message logic described earlier. This includes utilizing intents which uses the actions mentioned above within the notifications to directly open specific areas of the app, such as medication screens or other relevant sections. This integration ensures that push notifications trigger user interactions effectively, by allowing users to tap and be directed to specific features like medication tracking. For an enhanced user experience, all notification interactions reuse the established logic and intent handling mechanisms, maintaining consistency across both in-app messages and push notifications.

Notification Settings Screen (Figure: 5.6), showcases how users can customize their notification preferences within the app. These settings are grouped by sections, ensuring that users have control over which notifications they wish to receive.

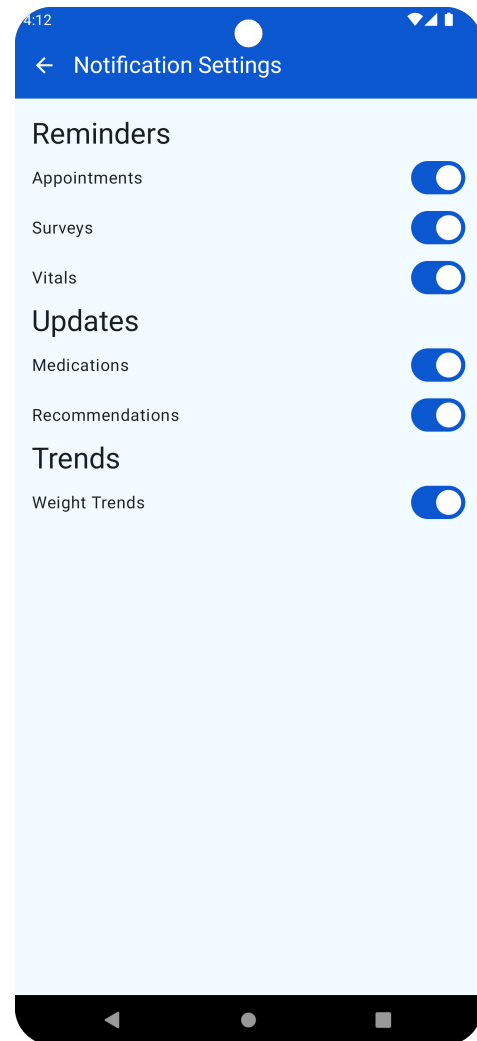


Figure 5.6.: Notification Settings Screen.

Home Screen

The Home Screen (Figure: 5.7) of the ENGAGE-HF app provides a central overview on which the most important health data and functions are easily accessible. A central component is the Messages area, which displays the messages from the previous section Messages.

Another important area is Connected Devices, where the status of the connected Bluetooth devices is displayed. If no devices are connected or Bluetooth is deactivated, the user is prompted to activate Bluetooth to receive measurement data from the connected medical devices, such as a blood pressure monitor or a heart rate monitor. Further information on setting up and using the Bluetooth connection can be found in the chapter

Bluetooth Devices.

The last measured values of the most important vital data are displayed in the lower area of the home screen. The latest measurements of weight, heart rate and blood pressure can be seen here, each with the corresponding measured value and the time of recording. This data is automatically transmitted by the connected devices and gives the user an up-to-date overview of their own health status.

The Home Screen thus efficiently combines communication via messages, real-time monitoring of vital data and the use of Bluetooth functionality to optimise heart health management.

Medication

During the development of the ENGAGE-HF application, it became apparent that the Spezi framework, despite its versatility and modularity, lacked a dedicated medication management interface. Given the critical importance of medication adherence in managing heart failure, this was a significant gap that needed to be addressed specifically for ENGAGE-HF. This chapter discusses the development of a custom medication management interface for ENGAGE-HF and explores the potential benefits of transforming this feature into a reusable module within the Spezi framework.

As we delved deeper into the requirements for ENGAGE-HF, it became evident that a robust medication management tool was essential for the app's success [64]. Patients managing chronic conditions like heart failure often need to keep track of multiple medications, adjust dosages, and stay informed about their treatment plans [64]. However, the Spezi framework did not include a pre-existing module to handle these tasks. This absence prompted us to develop a custom solution tailored to the specific needs of

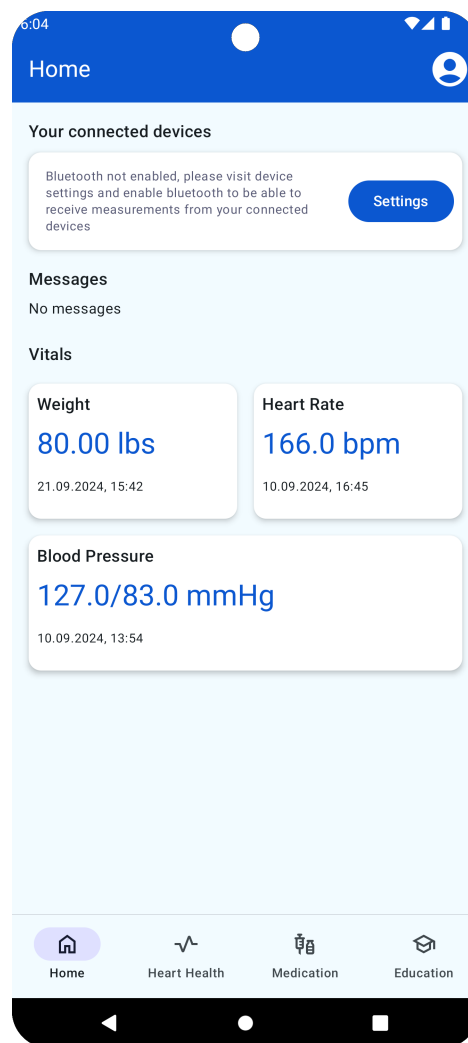


Figure 5.7.: Home Screen.

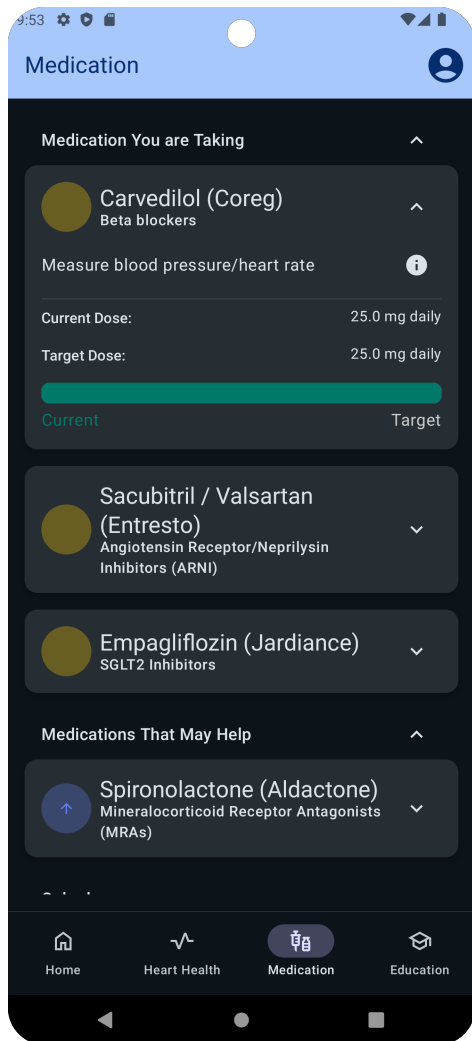
5. Case Study: *ENGAGE-HF for Heart Failure Management*

ENGAGE-HF.

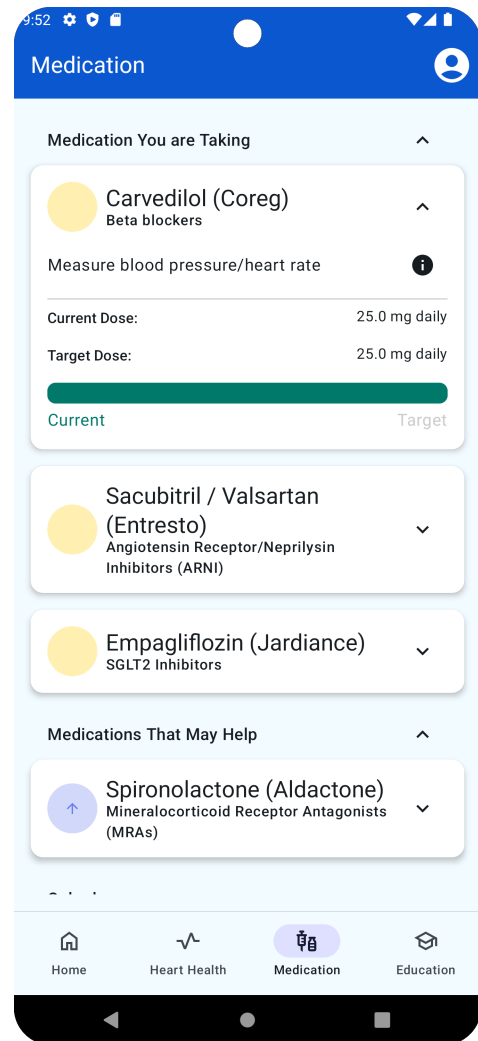
The medication management feature seen in figure 5.8 developed for ENGAGE-HF was designed with the user at its core, ensuring that patients could easily manage their medications while receiving clear and actionable information.

- **Medication Overview:** Each medication is displayed with a title and a subtitle, providing a straightforward overview that allows patients to quickly identify their medications. This design element is crucial in ensuring that users can easily navigate through their medication list without confusion.
- **Detailed Descriptions and Educational Content:** The interface includes a section for a detailed description of each medication. This section not only covers basic information such as usage and side effects but also links to educational videos that help patients understand the purpose and correct use of their medications. This aligns with ENGAGE-HF's broader goal of enhancing patient education through multimedia resources.
- **Current and Target Dosages:** A visual display shows both the current dose the patient is taking and the target dose prescribed by their healthcare provider. The use of a progress bar helps patients understand how close they are to achieving their target dose, making the titration process more transparent and less daunting.
- **Medication Prioritization:** The medications are sorted dynamically based on their current status. Medications that require immediate action, such as those needing dosage increases or those due for intake, are prioritized at the top of the list. This helps patients focus on what needs attention, enhancing adherence and reducing the risk of missed doses.

While the medication management interface was developed specifically for ENGAGE-HF, its utility extends far beyond this single application. The absence of such a module in the Spezi framework points to an opportunity for improvement. By modularizing this feature and incorporating it into the Spezi ecosystem, other developers working on mHealth applications could easily integrate medication management into their apps without needing to develop it from scratch.



(a) Dark Mode.



(b) Light Mode.

Figure 5.8.: Medication Screen.

The modular design would allow this feature to be adapted to various use cases, not just for heart failure management but for any condition requiring strict medication adherence. This would significantly enhance the Spezi framework's versatility and reusability, aligning with its core philosophy of modular, scalable development.

Onboarding, Login, Register & Account

The Account Module (Section: 4) was directly reused to manage user authentication, which included both login and registration screens. This module provided a robust interface with support for email and Google-

5. Case Study: *ENGAGE-HF* for Heart Failure Management

based authentication methods, which were crucial for ensuring security and convenience. The pre-built `AuthenticationManager` interface from the framework allowed for easy integration of authentication services, saving substantial development time by avoiding the need to build these systems from scratch. The module's compatibility with Firebase ensured a smooth experience for users and a secure backend.

In addition to `AuthenticationManager`, the `UserSessionManager` was crucial for managing user sessions across the app. This interface handled tasks such as uploading consent documents and tracking user state (e.g., whether a user is registered, anonymous, or has provided consent). The ability to observe user state through a reactive data stream allowed for real-time updates in the app's UI. For instance, if a user were logged out or their consent status changed, the app would immediately reflect this by navigating them back to the appropriate screen.

The `UserSessionManager` also provided access to important user information through methods like `getUserId()` and `getUserInfo()`, which ensured that the user's data was always up-to-date and accessible throughout the app, enhancing both security and personalization.

The Onboarding Module (Section: 4) was another reusable component that streamlined the user onboarding process. It allowed for the efficient collection of information and user consent, which was critical for the study's ethical and legal compliance. The module's customizable consent screens made it easy to adapt the onboarding flow for the heart failure study without needing to rebuild these components from scratch.

The Invitation Code logic also available in the Onboarding Module (Section: 4) provided a secure way to ensure that only pre-approved participants could join the *ENGAGE-HF* study. The `InvitationAuthManager` interface managed the verification of these codes by providing the default `FirebaseInvitationAuthManager` implementation. It was linking the invitation codes to user accounts in Firebase. By implementing this system, we ensured that only authorized users, identified by their unique invitation codes, could complete the registration process, preserving the integrity of the study.

The reuse of these modules not only reduced development time but also maintained a high standard of security, consistency, and user experience across different components. This modular approach illustrates the framework's potential for reusability in various mHealth applications, saving

time while maintaining flexibility for customization.

6. Evaluation

This chapter aims to evaluate the effectiveness of the developed mHealth framework, especially in terms of its reusability and adaptability. This evaluation is essential to ensure that the framework not only works effectively in the current ENGAGE-HF application, but can also be easily transferred to other healthcare applications. This includes both technical and functional aspects of the framework.

Validation of Reusability

The modular design of the Spezi framework has proven effective in supporting the ENGAGE-HF app, which is designed to manage heart failure patients. One of the major objectives of the project was to ensure that the framework could be reused across various mHealth applications without requiring significant re-engineering. During the testing phase, various modules of the framework—such as the authentication system, consent management, and health monitoring—were seamlessly reused in ENGAGE-HF without any need for modification. This aligns with findings from Wilhide, Peeples, and Kouyaté, who emphasize the importance of systematic approaches in designing mHealth apps that can be adapted to multiple healthcare settings [65].

Moreover, the framework's compatibility with industry-standard tools like Google Firebase further facilitated its reusability by offering a secure backend that was already compatible with a wide range of applications. The successful implementation in ENGAGE-HF highlights the framework's capability to provide developers with pre-built, modular solutions that save development time and effort. Ndlovu, Mars, and Scott also support this notion, indicating that interoperability frameworks are crucial for linking mHealth applications to electronic record systems, thereby enhancing reusability [23].

Reusable Modules in the ENGAGE-HF Case Study

In the ENGAGE-HF app, the following modules from the Spezi framework were reused and customized to meet the needs of heart failure management:

- **Authentication Module:** Used for secure login and registration. This module was integrated without modification and worked seamlessly to authenticate patients and healthcare providers.
- **Bluetooth Module:** Reused to connect medical devices such as blood pressure monitors and scales via Bluetooth Low Energy (BLE). This module was critical for real-time data collection in the heart failure management process, supporting the findings of Carrillo, Kroeger, Cárdenas, *et al.*, who discuss the importance of mobile technologies in health monitoring [66].
- **Logging Module:** Provided centralized logging functionality, making it easier to track errors, monitor application health, and perform debugging. The module also contributed to performance monitoring in the ENGAGE-HF app.
- **Design System Module:** Offered reusable UI components, which were instrumental in maintaining a consistent and accessible user interface. Elements such as buttons, text fields, and other UI components were directly integrated to maintain a high level of user experience.
- **Navigation Module:** Enabled smooth transitions between screens, such as patient onboarding, health data monitoring, and medication management.
- **Consent and Data Capture Module:** Ensured that the application handled sensitive patient data in a legally compliant manner, capturing patient consent before collecting and storing health data.
- **Onboarding Module:** This module facilitated the introduction of patients to the ENGAGE-HF app by guiding them through initial setup steps such as profile creation, device connection (e.g., Bluetooth pairing with medical devices), and basic app functionality tutorials.
- **Contact Module:** Provided patients with access to contact information for their healthcare providers.

- **Storage Module:** Utilized, to securely store data from medical devices.

These reusable modules significantly reduced the development time for the ENGAGE-HF app. Additionally, their modularity ensured that they could be easily adapted for other healthcare applications beyond heart failure.

Modules Implemented on Top of the Spezi Framework

While the Spezi framework provided a solid foundation, certain modules specific to the ENGAGE-HF app had to be implemented to fulfill the unique requirements of heart failure management. These include:

- **Medication Management Module:** This module was developed to allow users to manage their medications, set reminders, and track adherence. It required specific integration with health data and patient profiles to ensure the correct medication plans were followed.
- **Patient Engagement and Education Module:** To improve patient engagement, a module was developed that provides educational content, personalized health tips, and interactive features for better heart failure management. This was crucial for ensuring that patients were actively involved in managing their health.
- **Custom Health Monitoring Module:** Although the Bluetooth module from the framework provided the necessary connectivity, this module was created to handle specific heart-related metrics like blood pressure and heart rate, along with personalized health data visualizations tailored for heart failure patients.
- **Message and Alerts Module:** This module was designed to send alerts and messages to both patients and healthcare providers based on real-time health data or scheduled medication. It was critical for timely intervention and ensuring adherence to treatment plans.

These additional modules were necessary to meet the clinical requirements of heart failure management, which go beyond the generic capabilities of

6. Evaluation

the Spezi framework. By adding these on top of the existing framework, the ENGAGE-HF app was able to provide a comprehensive and specialized solution for heart failure patients.

Validation of Adaptability

Adaptability is another core feature evaluated in the Spezi framework. It is designed to handle different healthcare needs, ranging from heart failure management to potentially other chronic disease management such as diabetes or hypertension. This adaptability was evident when integrating various Bluetooth-connected medical devices for real-time data capture. The framework's architecture allowed for the quick integration of additional devices without major code refactoring, demonstrating its flexibility to adapt to new technological advancements. This is consistent with the findings of Ranjan, Rashid, Stewart, *et al.*, who describe the importance of scalability in mHealth platforms [67].

The framework's compliance with healthcare standards like FHIR also enhances its adaptability, as it ensures the framework can be integrated into various healthcare ecosystems across different regions. This flexibility will be instrumental in expanding the framework's reach beyond ENGAGE-HF, supporting various health applications in the future [68].

The modular architecture further strengthens adaptability, allowing for the integration of new features and supporting various medical needs. This design approach not only ensures a robust solution for current applications but also enables future enhancements, such as incorporating new health metrics or emerging technologies, with minimal modification.

Testing and User Feedback

To ensure that the framework meets the high demands of clinical applications, testing was carried out using automated testing tools integrated with GitHub Actions. The system passed key tests, including unit tests and integration tests, confirming that it is ready for deployment in real clinical environments. Additionally, initial user feedback from both patients and healthcare providers in the ENGAGE-HF study has been positive, emphasizing the ease of use and clarity of health data presentation, which aligns with the findings of Alnosayan, Chatterjee, Alluhaidan, *et al.* regarding

usability in mHealth systems [69].

High-Level Visualization of Framework and Application

To better understand the architecture, the following diagram (Figure: 6.1) provides a high-level overview of the separation between the Spezi framework and the ENGAGE-HF application.

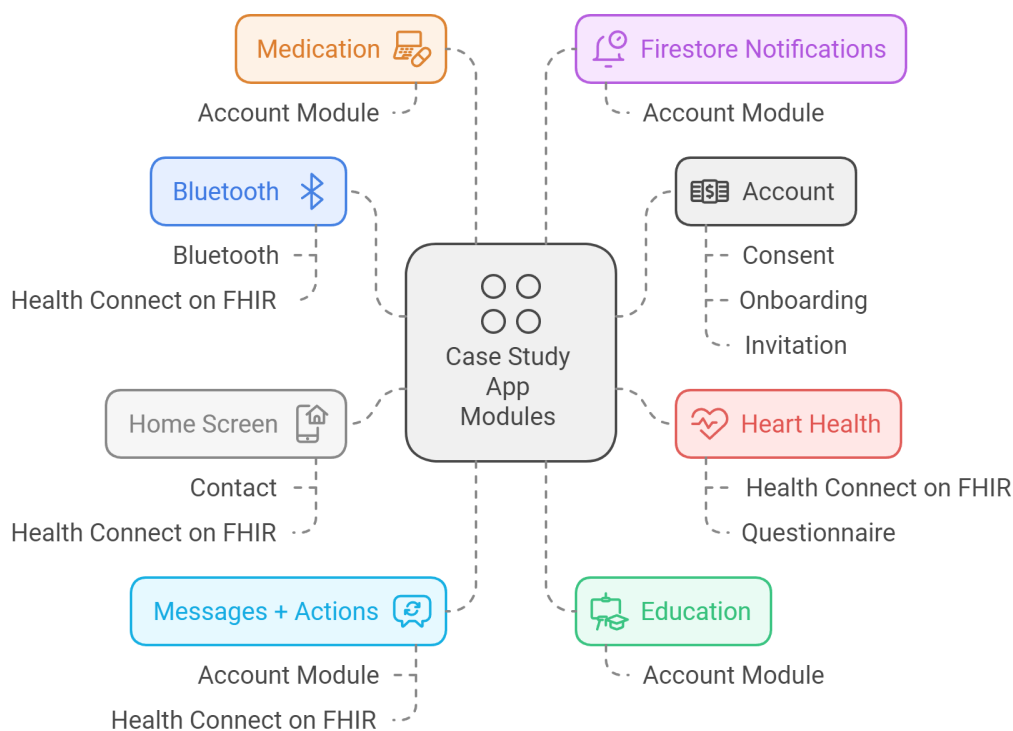


Figure 6.1.: The sub-items of the case study modules are the Spezi framework modules that were used in the case study modules. The figure illustrates the high reusability of these Spezi modules in the case study app. In addition to the modules mentioned, the utility modules, such as build-logic, logging, design and testing, form the foundation of all case study modules.

The Spezi Framework incorporates essential core modules, including Authentication, Bluetooth, Logging, and a Design System, which are designed to be reusable across various applications. The ENGAGE-HF Application utilizes these framework modules for specialized tasks such as patient management, health monitoring, and medication management. This clear separation of concerns allows the framework to supply the foundational

6. Evaluation

infrastructure, enabling the application to concentrate on user-specific workflows and healthcare requirements.

Building on the strengths of the Spezi Framework, its design ensures a clear separation of responsibilities by isolating general functionalities from application-specific logic. For instance, modules such as Bluetooth and authentication are broadly applicable across healthcare applications, while ENGAGE-HF adds specific logic tailored to heart failure management.

In summary, the evaluation of the Spezi framework shows that it fulfills its design goals of reusability, adaptability, and readiness for deployment in diverse mHealth applications. The success of the ENGAGE-HF app as a case study demonstrates the framework's robust architecture, paving the way for future implementations in other areas of healthcare. Additionally, by providing a modular, reusable system, the framework is well-positioned to adapt to new healthcare trends, further enhancing its value in the digital health ecosystem.

7. Discussion and Future Work

This chapter explores the potential future applications of the Spezi framework, identifies its limitations, and suggests directions for future research. The aim is to discuss how the framework can be further developed and improved to enhance its applicability in the mHealth domain.

Potential for Future Applications of the Framework

One of the most promising aspects of the Spezi framework is its adaptability beyond heart failure management. Given its flexible architecture, the framework could be adapted to manage other chronic conditions such as diabetes or hypertension. Additionally, it could be used in remote monitoring applications for mental health, enabling better access to care for patients in underserved areas.

The modular design allows for seamless integration of new features and ensures adaptability to different medical needs. This design principle not only provides a robust solution for current applications but also ensures that future applications can easily incorporate new health metrics or technologies without major modifications.

Limitations and Suggestions for Future Research

Despite the success of the Spezi framework, several limitations need to be addressed. One significant aspect is cross-platform support. At present, separate implementations of Spezi are needed for Android and iOS. While this approach maintains platform-native functionality, it results in API discrepancies between the two versions and increases development and maintenance efforts. This study emphasized best practices from Android development, leading to inconsistencies between the platforms.

7. Discussion and Future Work

A promising approach for future research could involve the use of Kotlin Multiplatform. This would enable the reuse of business logic and core functionality across both Android and iOS, streamlining development and minimizing the need for platform-specific code maintenance [70]. Additionally, future research should explore ways to align the APIs across both platforms to optimize the current separate development paths, ensuring a more uniform interface for users of Spezi across different platforms.

Another area for improvement is device abstraction. While the Bluetooth module currently supports standard BLE devices, incorporating additional communication protocols such as Zigbee, NFC, RFID and Z-Wave could broaden the range of connectable devices [71]. This expansion would enhance the framework's versatility and applicability in various healthcare settings, enabling integration with more advanced medical devices [71].

Finally, while the framework supports secure data handling via Firebase, the increasing complexity of healthcare regulations across different regions may necessitate more robust compliance features. Future work should consider integrating more comprehensive legal compliance modules, particularly concerning data privacy and patient consent across multiple jurisdictions.

Conclusion

The Spezi framework has proven to be a flexible, reusable, and scalable solution for the development of mHealth applications. While initial results are promising, particularly in managing heart failure through the ENGAGE-HF app, there are clear pathways for future research. These include the expansion into other chronic disease management areas, the integration of AI-driven technologies, and further refinement to ensure compliance with global healthcare regulations.

Additionally, there are several major iOS modules that are not yet reflected on Android, as they were not needed on Android so far. These include SpeziDevices, SpeziLLM, Chat, Speech, and HealthKit (which would be HealthConnect on Android).

8. Summary

This master's thesis presents the development of the Spezi framework for Android, which serves as a modular, reusable foundation for mHealth applications and is demonstrated using the example of the ENGAGE-HF app for heart failure management. The key findings of this thesis emphasise the importance of a flexible, scalable and modular approach to the development of mHealth solutions that not only improve patient care but also significantly reduce development effort.

Key findings

The Spezi framework has proven to be extremely reusable and customisable. In the ENGAGE-HF app, central modules such as the authentication, Bluetooth, logging, design and navigation modules as well as the module for recording patient data could be easily reused. This modularity facilitated development and provided the flexibility to fulfil specific heart failure management requirements through customised extensions such as medication management and patient education. The framework has proven its ability to integrate modern technologies such as Bluetooth Low Energy for medical devices and demonstrated compatibility with industry standards such as FHIR.

A key advantage of the framework is the clear separation of responsibilities between the generic functionalities of the framework and the application-specific requirements of the app. This not only enables the efficient development of new mHealth applications, but also creates a basis that can be easily adapted to other chronic diseases such as diabetes or high blood pressure.

Significance of the project for the mHealth community

The Spezi framework offers the mHealth community a significant step towards a standardised approach to healthcare application development. With the increasing prevalence of chronic diseases and the need to monitor and process patient data in real time, the framework provides a valuable resource for developers looking to quickly create scalable and secure healthcare solutions.

Due to its open-source nature, the Spezi framework offers the opportunity for collaborative development, enabling continuous improvement of functionalities and adaptation to new technologies and medical standards. This platform has the potential to have a significant impact on the future of digital healthcare by accelerating the development of new applications and improving access to innovative healthcare solutions.

Next steps

Despite the successful implementation in the ENGAGE-HF case study app, there are clear directions for future developments:

- **Expansion to cross-platform support:** currently, developing apps for Android and iOS requires separate implementations with Spezi iOS and Spezi Android. An obvious further development would be the use of Kotlin Multiplatform in order to be able to reuse business logic and core functionalities on both platforms. This would further reduce the development effort and increase consistency between the platforms.
- **Integration of additional protocols and devices:** Expanding support for additional communication protocols, such as Zigbee, RIFD, NFC or Z-Wave, would enable the framework to connect a wider range of medical devices, which could be particularly useful in specialised clinical environments.
- **Greater integration of AI:** Future developments could include the integration of AI-based predictive models that enable personalised

recommendations and predictions based on health data collected from patients. This could help to recognise health problems at an early stage and take preventative measures.

In summary, this work has shown that the Spezi framework provides a promising foundation for the next generation of mHealth solutions. Through its modularity and flexibility, it enables faster and more cost-efficient development of healthcare applications that both improve patient care and reduce the burden on the healthcare system.

List of abbreviations

mHealth Mobile Health

BLE Bluetooth Low Energy

MVVM Model View ViewModel

FHIR Fast Healthcare Interoperability Resources

LOINC Logical Observation Identifiers Names and Codes

AI Artificial Intelligence

HIS Hospital Information System

DI Dependency Injection

FCM Firebase Cloud Messages

EHR Electronic Health Record

A. Appendix

Software and Tools Used

The following tools were used to assist with the development, management, and writing of this thesis:

- **GitHub:** GitHub was used for version control and collaboration on code projects. (Available at: <https://github.com>)
- **GitHub Desktop:** GitHub Desktop was used as a GUI client to manage repositories and synchronize changes with GitHub. (Available at: <https://desktop.github.com>)
- **Android Studio:** Android Studio was the primary IDE used for Android app development. (Available at: <https://developer.android.com/studio>)
- **Citavi:** Citavi was used for reference management and organizing literature sources. (Available at: <https://www.citavi.com>)
- **DeepL Translator:** DeepL was used to translate text passages from English to German and vice versa. (Available at: <https://www.deepl.com>)
- **Grammarly:** Grammarly was employed to check grammar, spelling, and writing style. (Available at: <https://www.grammarly.com>)
- **Firebase:** Firebase was utilized for backend services, including authentication, real-time database, and analytics. (Available at: <https://firebase.google.com>)

These tools contributed to both the technical implementation and the linguistic quality of this thesis.

Code Developed

The code developed during the course of this thesis is hosted on GitHub and can be accessed via the following repository:

- **GitHub Repository:** The full codebase, including all relevant scripts and configurations, is available on GitHub. (Available at: <https://github.com/StanfordSpezi/SpeziKt>)

This repository contains all source code, project files, and documentation that were instrumental in the practical implementation described in the previous chapters.

Bibliography

- [1] *The Role of Digital Technology in Combating Chronic Disease* — *institute.global*, <https://www.institute.global/insights/public-services/role-digital-technology-combating-chronic-disease>, [Accessed 12-05-2024].
- [2] *Mobile Betriebssysteme - Internetnutzung März 2024 | Statista* — *de.statista.com*, <https://de.statista.com/statistik/daten/studie/184335/umfrage/marktanteil-der-mobilen-betriebssysteme-weltweit-seit-2009>, [Accessed 27-05-2024].
- [3] DelveInsight, *Mobile Apps for Chronic Diseases Management | Top Chronic Illness Apps* — *delveinsight.com*, <https://www.delveinsight.com/blog/chronic-disease-management-apps>, [Accessed 12-05-2024].
- [4] N. A. Cruz-Ramos, G. Alor-Hernández, L. O. Colombo-Mendoza, J. L. Sánchez-Cervantes, L. Rodríguez-Mazahua, and L. R. Guarneros-Nolasco, “Mhealth apps for self-management of cardiovascular diseases: A scoping review,” *Healthcare*, vol. 10, no. 2, p. 322, Feb. 2022, ISSN: 2227-9032. DOI: 10.3390/healthcare10020322.
- [5] G. Zisis, M. J. Carrington, B. Oldenburg, *et al.*, “An m-health intervention to improve education, self-management, and outcomes in patients admitted for acute decompensated heart failure: Barriers to effective implementation,” *European Heart Journal - Digital Health*, vol. 2, no. 4, pp. 649–657, Nov. 2021, ISSN: 2634-3916. DOI: 10.1093/ehjdh/ztab085.
- [6] J. T. Kelly, K. L. Campbell, E. Gong, and P. A. Scuffham, “The internet of things: Impact and implications for health care delivery,” *Journal of Medical Internet Research*, vol. 22, e20135, 11 2020. DOI: 10.2196/20135.
- [7] X. Liu, Y. Luo, and X. Yang, “Traceable attribute-based secure data sharing with hidden policies in mobile health networks,” *Mobile Information Systems*, vol. 2020, pp. 1–12, 2020. DOI: 10.1155/2020/3984048.

- [8] J. Ye, “The role of health technology and informatics in a global public health emergency: Practices and implications from the covid-19 pandemic,” *JMIR Medical Informatics*, vol. 8, e19866, 7 2020. DOI: 10.2196/19866.
- [9] C. Free, G. Phillips, L. Watson, *et al.*, “The effectiveness of mobile-health technologies to improve health care service delivery processes: A systematic review and meta-analysis,” *en, PLoS Med.*, vol. 10, no. 1, e1001363, Jan. 2013.
- [10] C. S. Hall, E. Fottrell, S. Wilkinson, and P. Byass, “Assessing the impact of mhealth interventions in low- and middle-income countries – what has been shown to work?” *Global Health Action*, vol. 7, 1 2014. DOI: 10.3402/gha.v7.25606.
- [11] O. Aalami, M. Hittle, V. Ravi, *et al.*, “Cardinalkit: Open-source standards-based, interoperable mobile development platform to help translate the promise of digital health,” *JAMIA Open*, vol. 6, no. 3, Jul. 2023, ISSN: 2574-2531. DOI: 10.1093/jamiaopen/ooad044.
- [12] P. Zagar, V. Ravi, L. Aalami, S. Krusche, O. Aalami, and P. Schmiedmayer, *Dynamic fog computing for enhanced llm execution in medical applications*, 2024. DOI: 10.48550/ARXIV.2408.04680.
- [13] C. O. Alenoghena, A. J. Onumanyi, H. O. Ohize, *et al.*, “Ehealth: A survey of architectures, developments in mhealth, security concerns and solutions,” *International Journal of Environmental Research and Public Health*, vol. 19, no. 20, p. 13071, Oct. 2022, ISSN: 1660-4601. DOI: 10.3390/ijerph192013071.
- [14] R. M. Masterson Creber, M. S. Maurer, M. Reading, G. Hiraldo, K. T. Hickey, and S. Iribarren, “Review and analysis of existing mobile phone apps to support heart failure symptom monitoring and self-care management using the mobile application rating scale (mars),” *JMIR mHealth and uHealth*, vol. 4, no. 2, e74, Jun. 2016, ISSN: 2291-5222. DOI: 10.2196/mhealth.5882.
- [15] <https://www.facebook.com/MinJourneys/>, *Medisafe App Review: An honest assessment of a popular medication app — minimalistjourneys.com*, <https://www.minimalistjourneys.com/medisafe-app-review/>, [Accessed 29-05-2024].
- [16] A. J. Burbank, S. D. Lewis, M. Hewes, *et al.*, “Mobile-based asthma action plans for adolescents,” *en, J. Asthma*, vol. 52, no. 6, pp. 583–586, Jul. 2015.
- [17] *The Complete Guide to Using MyFitnessPal for Successful Weight Loss - 33rd Square — 33rdsquare.com*, <https://www.33rdsquare.com/myfitnesspal-review/>, [Accessed 29-05-2024].

- [18] C.-K. Kao and D. M. Liebovitz, “Consumer mobile health apps: Current state, barriers, and future directions,” *PMR*, vol. 9, no. 5S, May 2017, ISSN: 1934-1563. DOI: 10.1016/j.pmrj.2017.02.018.
- [19] L. Woods, E. Cummings, J. Duff, and K. Walker, “Design thinking for mhealth application co-design to support heart failure self-management,” en, *Stud. Health Technol. Inform.*, vol. 241, pp. 97–102, 2017. DOI: 10.3233/978-1-61499-794-8-97.
- [20] M. I. Cajita, N. A. Hodgson, C. Budhathoki, and H.-R. Han, “Intention to use mhealth in older adults with heart failure,” en, *J. Cardiovasc. Nurs.*, vol. 32, no. 6, E1–E7, Nov. 2017.
- [21] K. D. Lopez, S. Chae, G. Michele, *et al.*, “Improved readability and functions needed for mhealth apps targeting patients with heart failure: An app store review,” *Research in Nursing and Health*, vol. 44, pp. 71–80, 1 2020. DOI: 10.1002/nur.22078.
- [22] E. E. Tripoliti, G. S. Karanasiou, F. G. Kalatzis, K. K. Naka, and D. I. Fotiadis, “The evolution of mhealth solutions for heart failure management,” in *Advances in Experimental Medicine and Biology*, ser. Advances in experimental medicine and biology, vol. 1067, Cham: Springer International Publishing, 2018, pp. 353–371. DOI: 10.1007/5584_2017_99.
- [23] K. Ndlovu, M. Mars, and R. E. Scott, “Interoperability frameworks linking mhealth applications to electronic record systems,” *BMC Health Services Research*, vol. 21, 1 2021. DOI: 10.1186/s12913-021-06473-6.
- [24] D. Estrin and I. Sim, “Health care delivery. open mhealth architecture: An engine for health care innovation,” en, *Science*, vol. 330, no. 6005, pp. 759–760, Nov. 2010.
- [25] *MyPHD x2013; The solution for running successful large-scale biomedical research with wearable and multiomics data.* — *myphd.stanford.edu*, <https://myphd.stanford.edu/>, [Accessed 03-09-2024].
- [26] A. Raghu, D. Praveen, D. Peiris, L. Tarassenko, and G. Clifford, “Engineering a mobile health tool for resource-poor settings to assess and manage cardiovascular disease risk: Smarthealth study,” *BMC Medical Informatics and Decision Making*, vol. 15, no. 1, Apr. 2015, ISSN: 1472-6947. DOI: 10.1186/s12911-015-0148-4.
- [27] A. S. Mustafa, N. Ali, J. S. Dhillon, G. Alkawsy, and Y. Baashar, “User engagement and abandonment of mhealth: A cross-sectional survey,” *Healthcare*, vol. 10, p. 221, 2 2022. DOI: 10.3390/healthcare10020221.

- [28] M. S. Liew, J. Zhang, J. See, and Y. L. Ong, “Usability challenges for health and wellness mobile apps: Mixed-methods study among mhealth experts and consumers,” *JMIR mHealth and uHealth*, vol. 7, e12160, 1 2019. DOI: 10.2196/12160.
- [29] M. Greve, A. Brendel, N. v. Osten, and L. M. Kolbe, “Overcoming the barriers of mobile health that hamper sustainability in low-resource environments,” *Journal of Public Health*, vol. 30, pp. 49–62, 1 2021. DOI: 10.1007/s10389-021-01536-8.
- [30] S. Shiferaw, A. Workneh, R. Yirgu, G. Dinant, and M. Spigt, “Designing mhealth for maternity services in primary health facilities in a low-income setting – lessons from a partially successful implementation,” *BMC Medical Informatics and Decision Making*, vol. 18, 1 2018. DOI: 10.1186/s12911-018-0704-9.
- [31] O. Haggag, J. Grundy, M. Abdelrazek, and S. Haggag, “A large scale analysis of mhealth app user reviews,” *Empirical Software Engineering*, vol. 27, 7 2022. DOI: 10.1007/s10664-022-10222-6.
- [32] J. P. Ratanawong, J. A. Naslund, J. P. Mikal, and S. W. Grande, “Achieving the potential of mhealth in medicine requires challenging the ethos of care delivery,” *Primary Health Care Research and Development*, vol. 23, 2022, ISSN: 1477-1128. DOI: 10.1017/s1463423622000068.
- [33] R. M. C., *Clean architecture: A craftsman’s guide to software structure and design*, en. 2017.
- [34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0201633612.
- [35] H. A. Epiloksa, D. S. Kusumo, and M. Adrian, “Effect of mvvm architecture pattern on android based application performance,” *Jurnal Media Informatika Budidarma*, vol. 6, p. 1949, 4 2022. DOI: 10.30865/mib.v6i4.4545.
- [36] *Understanding Plugins — docs.gradle.org*, https://docs.gradle.org/release-nightly/userguide/custom_plugins.html, [Accessed 26-06-2024].
- [37] *GitHub - jjohannes/idiomatic-gradle: How do I idiomatically structure a large build with Gradle — github.com*, <https://github.com/jjohannes/idiomatic-gradle>, [Accessed 26-06-2024].
- [38] *Herding Elephants — developer.squareup.com*, <https://developer.squareup.com/blog/herding-elephants/>, [Accessed 26-06-2024].

- [39] *The Benefits of a Design System: Making Better Products, Faster* | Toptal® — *toptal.com*, <https://www.toptal.com/designers/design-systems/benefits-of-design-system>, [Accessed 27-06-2024].
- [40] *11 Benefits of Design Systems for Designers, Developers, Product Owners, and Teams* | *Built In* — *builtin.com*, <https://builtin.com/articles/11-benefits-design-systems>, [Accessed 27-06-2024].
- [41] M. Yoon, S. Lee, J. Y. Choi, *et al.*, “Effectiveness of a smartphone app-based intervention with bluetooth-connected monitoring devices and a feedback system in heart failure (smart-hf trial): Randomized controlled trial,” *Journal of Medical Internet Research*, vol. 26, e52075, Apr. 2024, ISSN: 1438-8871. DOI: 10.2196/52075.
- [42] E. Cano, *SpeziKt/core/bluetooth/README.md at main · StanfordSpezi/SpeziKt* — *github.com*, <https://github.com/Stanford/Kt/blob/main/core/bluetooth/README.md>, [Accessed 24-07-2024].
- [43] C. Stewart, *Callback Hell in JavaScript: Taming Asynchronous Complexity* — *redsurgetechnology.com*, <https://redsurgetechnology.com/callback-hell-in-javascript-taming-asynchronous-complexity/>, [Accessed 24-07-2024].
- [44] *LiveData overview* | *Android Developers* — *developer.android.com*, <https://developer.android.com/topic/libraries/architecture/livedata>, [Accessed 24-07-2024].
- [45] M. L. Dechert, *Besser zentral: Professionelles Logging* — *heise.de*, <https://www.heise.de/ratgeber/Besser-zentral-Professionelles-Logging-2532864.html>, [Accessed 14-07-2024].
- [46] E. Cano, *SpeziKt/core/logging at main · StanfordSpezi/SpeziKt* — *github.com*, <https://github.com/Stanford/Kt/tree/main/core/logging>, [Accessed 24-07-2024].
- [47] *Dependency injection with Hilt* | *Android Developers* - *developer android com*, <https://developer.android.com/training/dependency-injection/hilt-android>, [Accessed 24-06-2024].
- [48] P. Achimugu, B. Afolabi, O. Adeniran, I. Gambo, and O. Oluwagbemi, “Software architecture and methodology as a tool for efficient software engineering process: A critical appraisal,” *Journal of Software Engineering and Applications*, vol. 03, pp. 933–938, 10 2010. DOI: 10.4236/jsea.2010.310110.

- [49] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, “Automated testing of android apps: A systematic literature review,” *IEEE Transactions on Reliability*, vol. 68, pp. 45–66, 1 2019. DOI: 10.1109/tr.2018.2865733.
- [50] J. Petrić, T. Hall, and D. Bowes, “How effectively is defective code actually tested?” *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2018. DOI: 10.1145/3273934.3273939.
- [51] C. T. Watson, M. Tufano, K. Moran, G. Bavota, and D. Shybyvanyk, “On learning meaningful assert statements for unit test cases,” *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020. DOI: 10.1145/3377811.3380429.
- [52] J. Middleton and T. Atapattu, “Beyond accuracy: Assessing software documentation quality,” *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Softw*, 2020. DOI: 10.1145/3368089.3417045.
- [53] <http://hl7.org/fhir>, *Observation - FHIR v6.0.0-ballot1 - build.fhir.org*, <https://build.fhir.org/observation.html>, [Accessed 15-08-2024].
- [54] *BfArM - LOINC — bfarm.de*, https://www.bfarm.de/DE/Kodiersysteme/Terminologien/LOINC-UCUM/LOINC-und-RELMA/_node.html, [Accessed 15-08-2024].
- [55] *InstantRecord | Android Developers — developer.android.com*, <https://developer.android.com/reference/android/health/connect/datatypes/InstantRecord>, [Accessed 15-08-2024].
- [56] “Ieee standard for open mobile health data—representation of metadata, sleep, and physical activity measures,” *IEEE Std 1752.1-2021*, pp. 1–24, 2021. DOI: 10.1109/IEEESTD.2021.9540821.
- [57] M. Vishnu Ravi, *Building for Digital Health with FHIR*, <https://docs.google.com/presentation/d/1Rf1d1Hr3QDNBGI0NH0kWtbVTxoFY4Y3qgEe4STGvqcY> & <https://vishnu.io/>, [Accessed 24-08-2024].
- [58] *Designing the User Onboarding Experience | UX Booth — uxbooth.com*, <https://uxbooth.com/articles/designing-the-user-onboarding-experience/>, [Accessed 28-06-2024].
- [59] <http://hl7.org/fhir>, *Questionnaire - FHIR v5.0.0 — hl7.org*, <https://hl7.org/fhir/questionnaire.html>, [Accessed 24-08-2024].

- [60] <http://hl7.org/fhir>, *SDC Home Page - Structured Data Capture v3.0.0 - build.fhir.org*, <https://build.fhir.org/ig/HL7/sdc/>, [Accessed 24-08-2024].
- [61] *Navigation with Compose | Jetpack Compose | Android Developers — developer.android.com*, <https://developer.android.com/develop/ui/compose/navigation>, [Accessed 24-08-2024].
- [62] S. Amagai, S. Pila, A. J. Kaat, C. J. Nowinski, and R. C. Gershon, “Challenges in participant engagement and retention using mobile health apps: Literature review,” *Journal of Medical Internet Research*, vol. 24, no. 4, e35120, Apr. 2022, ISSN: 1438-8871. DOI: 10.2196/35120.
- [63] Y. Tsai, P. Hsiao, M. Kuo, *et al.*, “Mobile health, disease knowledge, and self-care behavior in chronic kidney disease: A prospective cohort study,” *Journal of Personalized Medicine*, vol. 11, p. 845, 9 2021. DOI: 10.3390/jpm11090845.
- [64] J. Piette, K. Lun, L. Moura, *et al.*, “Impacts of e-health on the outcomes of care in low- and middle-income countries: Where do we go from here?” *Bulletin of the World Health Organization*, vol. 90, no. 5, pp. 365–372, May 2012, ISSN: 0042-9686. DOI: 10.2471/blt.11.099069.
- [65] C. Wilhide, M. Peeples, and R. A. Kouyaté, “Evidence-based mhealth chronic disease mobile app intervention design: Development of a framework,” *JMIR Research Protocols*, vol. 5, e25, 1 2016. DOI: 10.2196/resprot.4838.
- [66] M. Carrillo, A. Kroeger, R. Cárdenas, S. D. Monsalve, and S. Runge-Ranzinger, “The use of mobile phones for the prevention and control of arboviral diseases: A scoping review,” *BMC Public Health*, vol. 21, 1 2021. DOI: 10.1186/s12889-020-10126-4.
- [67] Y. Ranjan, Z. Rashid, C. Stewart, *et al.*, “Radar-base: Open source mobile health platform for collecting, monitoring, and analyzing data using sensors, wearables, and mobile devices,” *Jmir Mhealth and Uhealth*, vol. 7, e11734, 8 2019. DOI: 10.2196/11734.
- [68] I. B. R. G. Tumeh, C. D. Bergerot, D. Lee, E. J. Philip, and R. Freitas-Júnior, “Mhealth program for patients with advanced cancer receiving treatment in a public health hospital in brazil,” *Psycho-Oncology*, vol. 32, pp. 125–132, 1 2022. DOI: 10.1002/pon.6059.
- [69] N. Alnosayan, S. Chatterjee, A. S. Alluhaidan, E. Lee, and L. Feenstra, “Design and usability of a heart failure mhealth system: A pilot study,” *JMIR Human Factors*, vol. 4, e9, 1 2017. DOI: 10.2196/humanfactors.6481.

- [70] I. Olenych and R. Korostenskyi, “Analysis of the effectiveness of using kotlin multiplatform mobile technology for creating cross-platform applications,” *Electronics and Information Technologies*, vol. 21, 2023. DOI: 10.30970/ei.21.3.
- [71] W. Zhao and S. Sampalli, “Sensing and signal processing in smart healthcare,” *Electronics*, vol. 9, p. 1954, 11 2020. DOI: 10.3390/electronics9111954.