Hochschule Konstanz
Department of Computer Science

Submitted by:
Lars Bürger
Marcel Biselli
Simon Rauch
info@wallat.com

Mobile Anwendungen - SS 23

Projektdokumentation

Konstanz, 2023-07-12

Contents

List of Figures

1 Einleitung

Wir, das Entwicklerteam bestehend aus Marcel Biselli, Lars Bürger und Simon Rauch, haben im Rahmen des AIN Kurses "Mobile Anwendungen" im Sommersemester 2023 eine mobile App namens "Wall@" entwickelt. Unser Ziel war es, eine innovative Lösung für das Scannen und Verwalten von Dokumenten zu schaffen.

Mit Wall@ bieten wir eine benutzerfreundliche und effiziente Möglichkeit, Dokumente digital zu erfassen und in einem sicheren digitalen Geldbeutel zu verwalten. Die App richtet sich an alle, die eine bequeme Methode suchen, um Dokumente wie Quittungen, Rechnungen, Ausweise oder andere wichtige Unterlagen zu scannen, zu speichern und jederzeit griffbereit zu haben.

Bei der Entwicklung von Wall@ haben wir bewährte Software-Engineering-Praktiken und -Standards berücksichtigt, um eine stabile, skalierbare und gut strukturierte App zu gewährleisten. Durch den Einsatz der Programmiersprache Dart und des Flutter-Frameworks ist es uns gelungen, eine plattformübergreifende Lösung zu entwickeln, die gleichermaßen auf den Betriebssystemen iOS und Android funktioniert.

Diese Dokumentation dient als verbindliche Ressource für Entwickler, die an der App weiterarbeiten möchten, und bietet Einblicke in die Implementierungsdetails sowie die zugrunde liegenden Technologien.

Bei weiteren Fragen, Feedback oder Anregungen stehen wir Ihnen gerne zur Verfügung. Wir hoffen, dass diese Dokumentation einen wertvollen Beitrag zur effizienten Nutzung und Weiterentwicklung von Wall@ leistet.

Figure 1.1: Wall@ Logo

2 Entwicklungsprozess

Dieses Kapitel bietet einen Überblick über den Entwicklungsprozess von Wall@. Es behandelt die geplanten User Stories und den Einsatz von GitLab für das Projektmanagement und die Zusammenarbeit im Team. Die User Stories dienen als Leitfaden für die Entwicklung und beschreiben die Funktionalität von Wall@ aus Benutzersicht. Der Entwicklungsprozess umfasst verschiedene Phasen und Praktiken, die sicherstellen, dass die App erfolgreich entwickelt, getestet und bereitgestellt werden kann. Zusätzlich wird der Einsatz von GitLab vorgestellt, einer kollaborativen Entwicklungsplattform, die Funktionen zur Versionskontrolle und zum Projektmanagement bietet.

2.1 Issues in GitLab

Für die Koordination und Verwaltung des Entwicklungsprozesses haben wir das Ticket-System in GitLab genutzt. Mithilfe dieses Systems konnten wir Tickets erstellen, Aufgaben zuweisen und die Entwicklung effektiv koordinieren. Wir haben Funktionen wie Ticketzuweisung (an Verantwortliche), Labeling und Kommentarfunktionen genutzt, um die Zusammenarbeit und den Fortschritt innerhalb des Teams zu erleichtern.

2.2 User Stories

Aufgrund der Anforderungen an die App haben wir verschiedene User Stories erstellt, die die Funktionalität von Wall@ aus Benutzersicht beschreiben. Eine Liste der User Stories spowie der aktuelle Entwicklungsstand der einzelnen Stories ist über ein seperates GitLab Board (siehe User Stories Board) einzusehen.

2.3 GitFlow

Für eine effiziente Verwaltung des Entwicklungsprozesses haben wir das GitFlow-Modell implementiert. Es ermöglichte uns, den Code in verschiedenen Branches zu organisieren und stabile sowie entwicklungsorientierte Umgebungen aufrechtzuerhalten. Wir haben den "master"-Branch für stabile Versionen, den "develop"-Branch für die Hauptentwicklung und Feature-Branches für neue Funktionen verwendet. Zusätzlich nutzten wir Hotfix-Branches, um kritische Fehler schnell zu beheben und in den "master" und "develop"-Branch zu überführen. Zudem wurden alle Commits mit einer entsprechenden Issue-Nummer versehen, um die Nachvollziehbarkeit zu gewährleisten.

Während der Entwicklung ergab sich recht schnell eine klare Arbeitsaufteilung: Simon beschäftigte sich mit Organisation und der Entwicklung der Scanner-Funktion; Lars entwickelte viele der View-Elemente und Marcel kümmerte sich um den Großteil des Backends.

3 Architektur

3.1 MVC+S

Jede Komponente der Anwendung unterliegt einer MVC-Struktur. Komponenten beziehen sich hierbei auf Haupt-Features der App, wie etwa die Ordner-Ansicht oder Detail-Ansicht eines Dokuments.

Dabei sind die Abhängigkeiten aller Komponenten entsprechend des Dependency-Inversion-Prinzips umgekehrt. Jede View definiert sich ihren Controller selbst und erhält die Implementation über einen Riverpod Provider.

3.1.1 Services

Um den Zugriff auf die Datenbank zu vereinfachen wurde ein Persistenz-Service eingerichtet, sodass der Zugriff transparent geschieht. Siehe Bild 3.1 für eine informelle Graphik der verwendeten Architektur. Die Persistenz-Service ist modular, sodass Komponenten einfach ausgetauscht werden können. Dies erreichen wir mithilfe des DAO-Pattern, um den tatsächlichen Zugriff auf den Speicher zu maskieren. Im Sinne des Dependency-Inversion-Prinzips definiert der `PersistenceService` die DAOs, die dieser benötigt. Für mehr Information zum Aufbau der Persistenz-Schicht, siehe Kapitel 4.

Zusätzlich wird ein `DbController` definiert, welcher dafür zuständig ist, die Datenbank zu laden und die DAOs zu instanziieren. Dessen Implementation - und damit die der DAOs - wird über Riverpod injected. Controller erhalten eine Instanz des `PersistenceService` über einen Provider, welcher wiederum einen Provider für den DbController aufruft. Letzterer bestimmt die tatsächliche Implementation.

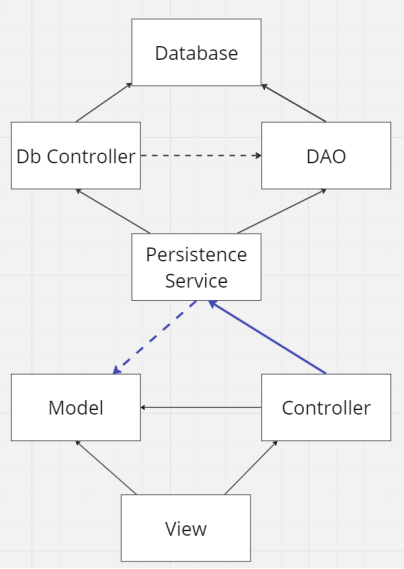

Figure 3.1: MVC+S Architektur

3.2 Compositum Pattern für Ordner

Die unterliegende Datenklasse für die Ordner-Struktur ist mittels des Compositum-Patterns umgesetzt. Dazu wird die abstrakte Klasse `FolderItem` genutzt, welche entweder ein `SingleItem` oder ein `Folder` ist, welcher weitere FolderItems enthält. Dadurch können Ordner sehr leicht wiederum andere Ordner enthalten und die Unterscheidung dieser dennoch einfach. So wird zum Besipiel in `FolderItem` auch definiert, wie zur Detail-Ansicht dieser Entität navigiert werden kann, sodass sich die View nicht um diese Logik kümmern muss.

3.3 State Management (Riverpod)

Die App nutzt Riverpod zur Verwaltung des App-States. In einer Klasse sind die zahlreichen Provider als statische Variablen definiert, die von jeglichen Komponenten genutzt werden können. Dabei gibt es für jede Komponente einen spezifischen Provider. Typischerweise ist das ein `StateNotifierProvider` auf dem Model und dem Controller der Komponente oder die Async-Variante davon, wenn etwa Daten asynchron über den PersistenceService geladen werden müssen.

3.3.1 Provider Families

Die Provider für Ordner und Dokumente sind speziell, da diese eine ganze Reihe an Entitäten abdecken und nicht nur eine einzelne Instanz. Dies lässt sich recht einfach über die Family-Provider von Riverpod lösen. Dabei muss beim Abruf des Provider ein Identifier – in diesem Fall die ID der Entität – übergeben werden und so erhält man einen neuen Provider für jede neue Entität.

Bei Dokumenten kommt noch dazu, dass diese über eine spezielle View editierbar sein sollen. Diese erlaubt, Änderungen zu puffern und bei Bedarf anzuwenden oder zu verwerfen. Aus diesem Grund gibt es einen zweiten Family-Provider, welcher für das Editieren der Einträge verantwortlich ist.

Der Provider für Ordner ist so aufgebaut, dass bei Übergabe von null als Identifier der Provider für das Root-Verzeichnis des Profils zurückgeliefert wird. Die `Folder`-Klasse nutzt ein mixin '`ExternalResource`', welches die Variablen `isLoading` und `hasError` definiert. Diese werden genutzt, wenn der Ordner asynchron geladen werden muss. Dadurch kann beim Aufruf von Unterordnern dieser eifnach aus den Inhalten des vorherigen gezogen werden und der View als initiale Daten übergeben werden, während die tatsächlichen Daten im Hintergrund neu geladen werden. Um diesen Aufruf zu vereinfachen wird eine selbst definierte Builder-Klasse genutzt; siehe Abschnitt 3.5.1 zu angepassten `FutureBuildern`.

3.4 Multi-Language Support

Um Text in der App in verschiedenen Sprachen anzeigen zu können, wird eine abstrakte `Language`-Klasse genutzt. Diese bietet Getter-Methoden für verschiedenste Strings, die von den Views benötigt werden. Die Methoden-Namen sind so gewählt, dass das erste Element des Namens den Verwendungszeck und der Rest den Kontext angibt. So werden zum Beispiel Seiten-Titeln ein 'title' vorangestellt; konstanten Label-Texten ein

'lbl', Knöpfen ein 'btn', Textfeldern ein 'txt', Info-Benachrichtigugen ein 'info', Fehlerbenachrichtigungen ein 'err'. Die Implementationen der Klasse bestimmen den genauen Text. Die korrekte Implementation erhält eine View wiederum durch den Riverpod-Provider für die App-Einstellungen.

3.5 ErrorHandling

Dadurch, dass die Views der App vorwiegend durch Slivers aufgebaut sind, können Overflow-Errors oder ähnliches in der Regel ausgeschlossen werden. Die einzige ernstzunehmende Fehlerquelle der App ist die Persistenz-Schicht. Dabei wird allerdings selten eine tatsächliche Exception geworfen, sondern einfach `null` zurückgeliefert. Aus diesem Grund werden die meisten Daten mittels Methoden zur Null-Safety verwendet. So liefert zum Beispiel der Provider für Dokumente ein `SingleItem?` und die `contents` der `Folder`-Klasse sind eine `List<FolderItem>?`.

3.5.1 Angepasste FutureBuilders

Um mit all diesen verschiedenen Zuständen und zusätzlich noch der Asynchronität umgehen zu können werden eigens geschriebene Varianten des `FutureBuilder`-Widgets verwendet. Diese kommen in 3 Varienten: einmal für die Standard `Future`s von Dart, dann für die `AsyncValue`s von Riverpod und letztlich für unsere eigenen `ExternalResource`s. Diese Builder werden dann mit dem entsprechenden asynchronen Wert und einer Methode initialisiert, die das Widget bei erfolgreichem Laden baut. Im Fall eines Fehlers oder wenn null zurückgeliefert wird, wird stattdessen ein entsprechendes `ErrorMessage`-Widget zurückgeliefert oder auch eine `InfoMessage` für die spezifischen ListBuilder-Varianten, falls die Liste leer ist.

4 Persistenz

4.1 Datenbank

Daten werden mithilfe von Isar persistiert. In Bild 4.1 ist ein ERM-Diagramm der Datenbank abgebildet. Auch wenn innerhalb der Zugriffe auf Isar mit speziellen Datenklassen gearbeitet wird, wandeln die DAOs diese beim Zugriff in die Model-Klassen der MVC-Schicht um. So werden zum Beispiel die Events direkt in die enthaltene Liste der `SingleItem`-Klasse eingefügt und nicht seperat angefragt.

Bilder werden im Dokumenten-Verzeichnis der Anwendung abgelegt und in der Datenbank nur der dazugehörige Pfad gespeichert.

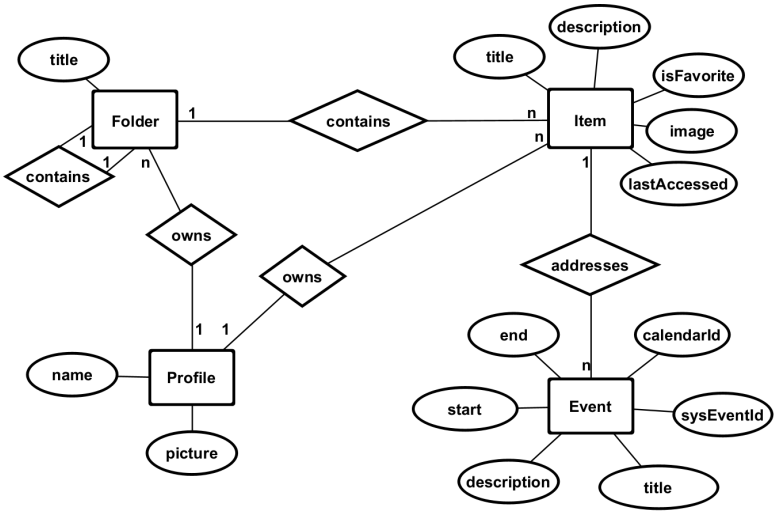

Figure 4.1: Persistenz ERM Diagram

Eine technische Besonderheit sind die Profile. Profile erlauben dem Nutzer, Daten noch einmal grober zu kategorisieren als Ordner. Um dies umzusetzen, verwaltet Isar ebenfalls eine Tabelle für Profile. Jeder andere Eintrag der Datenbank wird dann einem Profil zugeordnet. Die Besonderheit besteht darin, dass beim erstmaligen Starten der App ein Standard-Profil angelegt wird, welches zudem als eine globale Sicht fungiert. Das bedeutet, wenn ein Nutzer weitere Profile anlegt und mit Daten füllt, sind diese Daten auch über das Standard-Profil sichtbar. Zudem funktioniert die App auch ohne weitere Profile und es ändert sich nichts an der Nutzung, wenn ein Nutzer keine Profile erstellen möchte. Das derzeit ausgewählte Profil ist über den Provider für die App-Einstellungen erhältlich. Auf diesen wird beim erstellen des PersistenceProviders ein `watch` ausgeführt. Dadurch erhält jeder Controller automatisch einen PersistenceService, in dem das aktuelle Profil vermerkt ist, sodass die richtigen Daten zurückgeliefert werden.

Funktionalitäten zum Verschieben von Inhalten zwischen Profilen aber auch zwischen Ordnern sind im Backend bereits implementiert, allerdings sind diese noch nicht über die View zugreifbar.

4.2 Einstellungen

Um die Einstellungen, die ein Nutzer vornehmen kann, zu speichern, wird das Plugin shared_preferences genutzt. Dabei speichern wir zum Beispiel den ausgewählten System-Kalendar, das ausgewählte Farbschema, Profil und Sprache. Diese werden, wie üblich über einen per Riverpod verfügbaren Provider abgerufen.

5 Schnittstellen

5.1 Kamera via Package

Bei der Entwicklung von Wall@ war es erforderlich, ein Kamera-Paket in die App zu integrieren, um die Funktionalität des Dokumentenscanners zu ermöglichen. Nach einer gründlichen Evaluierung wurden die folgenden drei verschiedene Kamera-Pakete untersucht, um die Anforderungen unserer App zu erfüllen.

Package	Feature	Problem
flutter_document_scanner	Auto Crop, Manuel Crop, ScannLook	compiling errors
document_scanner_flutter	Auto Crop, Manuel Crop, ScannLook, pdf conv.	depends on photo_view package 0.12.0
cunning_document_scanner	Auto Crop, Manuel Crop	-

Ursprünglich hatten wir uns für das zweite Kamera-Paket entschieden, das unsere Anforderungen optimal erfüllte. Jedoch ist dieses Paket nicht mit Dart 3 kompatibel, was zu Kompatibilitätsproblemen und potenziellen Einschränkungen bei der Weiterentwicklung führen könnte.

Um dieses Problem zu lösen, wurde ein Issue auf dem Entwickler-Git des bevorzugten Kamera-Pakets erstellt (siehe Issue on GitHub) und das Problem kommuniziert. In der Zwischenzeit haben wir uns für ein alternatives Kamera-Paket entschieden, das mit Dart 3 kompatibel ist und unsere grundlegenden Anforderungen erfüllt. Durch die Nutzung des letzten Packet wurde die Funktionalität des Dokumentenscanners in Wall@ implementiert. Es ist somit möglich Bilder mittels der Kamera von Dokumenten aufzunehmen wobei die Ränder automatisch erkannt werden und das Bild entsprechend zugeschnitten wird.

Langfristig planen wir jedoch, eine eigene Lösung für den Dokumentenscanner in Wall@ zu entwickeln. Dies ermöglicht uns eine maßgeschneiderte Implementierung, die besser auf die spezifischen Anforderungen und zukünftigen Erweiterungen unserer App abgestimmt ist. Die Entwicklung einer eigenen Lösung wird es uns auch ermöglichen, volle Kontrolle über die Funktionalität und die Integration in den Rest der App zu haben.

In diesem Zuge soll auch die Funktionalität des Dokumentenscanners erweitert werden, so dass Bilder nicht nur als Bild, sondern auch als PDF gespeichert werden können und diese mittels Filter bearbeitet werden können so dass z.B.: Probleme mit der Belichtung oder dem Kontrast behoben werden können.

6 Generelles (Flutter)

6.1 Cupertino-Widgets

Die App verwendet vorwiegend Cupertino-Widgets, um eine Benutzeroberfläche zu erstellen, die den Design-Mustern von Apple-Anwendungen ähnelt.

6.2 Adaptives Design mit MediaQuery und Slivers

Um die App an verschiedene Bildschirmgrößen anzupassen, verwenden wir die MediaQuery-Klasse von Flutter.

Mit MediaQuery kann beispielsweise die verfügbare Bildschirmgröße ermittelt werden. Die App ist hiermit *adaptiv* und einzelne Elemente werden in ihrer Breite und Höhe an den Bildschirm angepasst. Das macht sie allerdings nicht *responsive* und die App ist derzeit nur für mobile Endgeräte im Hochformat ausgelegt.

Zusätzlich zur MediaQuery nutzen wir Slivers, um die App adaptiver zu gestalten. Slivers ermöglichen es, flexible und scrollbare Layouts zu erstellen, die sich automatisch an den verfügbaren Bildschirmplatz anpassen.

6.3 Animationen

6.3.1 Hero-Animation von Bildcontainern

Preview Bilder für Dokumente sind in ein `Hero`-Widget gewickelt. Dies erlaubt für einfache Animation des Bilds bei Navigation zu einem anderen Screen, auf dem das Bild ebenfalls vorhanden ist. Wenn ein Benutzer einen Eintrag in einer Liste antippt und in einen Einzeleintrag navigiert oder in der Einzelansicht das Bild im Vollbildmodus betrachtet, wird das Bild nahtlos von einem Container zum anderen animiert.

6.3.2 Animation des Kamera-Buttons

In der Home-Ansicht schwebt der Kamera-Button über der Navigationsleiste. In den anderen Seiten befindet sich dieser jedoch in der Leiste. Deswegen nutzt dieser eine individuelle Animation, um die neue Position anzuzeigen. Beim Navigieren, bewegt sich dieser in die Navigationsleiste hinein und wird letztlich transparent. Beim zurücknavigieren läuft die Animation verkehrt ab. Um dies zu implementieren wird das selbst definierte `CameraButtonHeroDestination`-Widget genutzt und auf den Seiten platziert, auf denen der Button in der Leiste zu sehen ist. Dort ist das Kamera-Icon in einem Hero-Widget mit einem individuellen `flightShuttleBuilder` platziert, welcher eine `FadeTransition` von opak zu transparent durchführt.

6.4 Widgets der Woche

Innerhalb des Projekts wurden eine Reihe von Flutter-Widgets verwendet, die als "Widgets der Woche" bekannt sind. Hier sind einige der Widgets, die genutzt werden; drei davon mit näher beschriebenem Nutzen:

1. **CupertinoSliverNavigationBar:** Dieser bietet eine elegante und visuell ansprechende Navigationsleiste im iOS-Stil, welche den Titel der aktuellen Seite, sowie einen 'Zurück'-Knopf und weitere Steuer-Elemente enthalten kann

2. **HeroMode:** Da die Hero-Animation der Preview-Bilder zwischen dem Home-, Favorites-, und Folder-Screen unerwünscht und nur bei Navigation zur Detail-Ansicht der Einträge ablaufen soll, wird `HeroMode` genutzt, um die Animation wenn nötig zu unterdrücken.

3. **FutureBuilder:** Mit diesem Widget macht asynchronen Zugriff auf Daten sehr viel einfacher. In Abschnitt 3.5.1 wird dessen Nutzung näher beschrieben.

SafeArea	LinearGradient	StatefulBuilder
GestureDetector	LayoutBuilder	Hero
SliverAppBar	CupertinoActivityIndicator	Divider
CupertinoActionSheet	DraggableScrollableSheet	CupertinoAlertDialog
Stack	AnimatedOpacity	MediaQuery
Flexible	Dismissible	SizedBox
SliverList	SliverGrid	

6.5 Themes

Unser Projekt implementiert ein flexibles Theme-System, das es uns ermöglicht, das Design unserer App einfach anzupassen oder neue Themes hinzuzufügen. Die Hauptfarben für sowohl das Dark- als auch das Light-Theme sind bewusst so gestaltet, dass sie den Apple-Farben ähneln jedoch unserem Color-CI entsprechen. Dadurch schaffen wir eine konsistente visuelle Ästhetik und sorgen dafür, dass unsere App in das Apple-Ökosystem passt.

6.6 Spezifische Packages

Wir nutzen verschiedene Packages in unserem Projekt, um spezifische Funktionalitäten zu implementieren. Dabei haben folgende Packages eine wichtige Rolle gespielt:

1. **social_share:** Das social_share Package wird genutzt, um den Benutzern die Möglichkeit zu geben, Einträge einfach und schnell über verschiedene soziale Medienplattformen zu teilen.

2. **device_calendar:** Das device_calendar Package spielte eine wichtige Rolle bei der Integration von Kalenderfunktionen. Mit diesem Package können Ereignisse zu den Systemkalendern der Benutzer hinzugefügt und entfert werden.

3. **isar:** Isar wurde verwendet, um eine effiziente Datenbank- und Persistenzverwaltung zu ermöglichen.

4. **beamer:** Beamer spielt eine wichtige Rolle bei der Implementierung des Routings innerhalb der App. Damit auch die komplexere und verschachtelte Navigation mit einer `BottomNavBar` umgesetzt.

6.7 Splash Screen

Neben dem Logo (siehe 1.1) wurde zudem ein ansprechender Splash Screen eingebaut, um einen visuell ansprechenden Start zu bieten, währenddessen die Datenbank geladen werden kann.