Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medienproduktion und -technik

Bachelorarbeit

von

Aida Kotschu

Multiplattform-App-Entwicklung am Beispiel einer Einkaufslisten-App

Multiplatform app development using the example of a shopping list app

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Medienproduktion und -technik

Bachelorarbeit

von

Aida Kotschu

Multiplattform-App-Entwicklung am Beispiel einer Einkaufslisten-App

Multiplatform app development using the example of a shopping list app

Bearbeitungszeitraum: von 9. Juni 2023
bis 29. November 2023

1.Prüfer: Prof. Dr. Dieter Meiller

2.Prüfer: Veit Stephan M. Eng.

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule Amberg-Weiden

Eigenständigkeitserklärung gemäß § 27 (8) ASPO

Name und Vorname

der Studentin/des Studenten: **Kotschu, Aida**

Studiengang: **Medienproduktion und -technik**

Ich bestätige, dass ich die Bachelorarbeit mit dem Titel:

Multiplattform-App-Entwicklung am Beispiel einer Einkaufslisten-App

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 27. November 2023

Unterschrift: 

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Bachelorarbeit Zusammenfassung

Studentin/Student (Name, Vorname):	**Kotschu, Aida**
Studiengang:	Medienproduktion und -technik
Aufgabensteller, Professor:	Prof. Dr. Dieter Meiller
Durchgeführt in (Firma/Behörde/Hochschule):	evidentmedia
Betreuer in Firma/Behörde:	Veit Stephan M. Eng.
Ausgabedatum:	9. Juni 2023
Abgabedatum:	28. November 2023

Titel:

Multiplattform-App-Entwicklung am Beispiel einer Einkaufslisten-App

Zusammenfassung:

Diese Bachelorarbeit erforscht und erläutert die Chancen und Hürden der Multiplattform-App-Entwicklung mit Dart und Flutter anhand eines Beispiels: eine Einkaufslisten-Anwendung, die als Besonderheit auf dem Multiplattform-Markt die Listeneinträge automatisch anhand vergangener Einkäufe sortiert.

Schlüsselwörter: Flutter, Dart, Multiplattform, App-Programmierung, Einkaufsliste

Inhaltsverzeichnis

Glossar

App	Anwendungssoftware, die vorwiegend für mobile Endgeräte entwickelt wurde
Bottom Sheet	Overlay-Oberfläche, die von unten auf dem Bildschirm erscheint und dem Benutzer zusätzliche Interaktionsmöglichkeiten bietet
Mock-Up	Visuelle prototypische Darstellung eines Designs oder einer Benutzeroberfläche, die in der Regel verwendet wird, um das Aussehen einer Anwendung zu präsentieren, bevor sie tatsächlich entwickelt oder umgesetzt wird
Service Worker	Web-Browser-Technologie, die im Hintergrund einer Website einen Proxy zwischen Browser des Nutzers und Website-Server bereitstellt. Erlaubt das Ausführen von JavaScript, ohne dass die dazugehörige Seite aktiv verwendet wird
Screen	Ansicht in einer Anwendung

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

1.1 Hintergrund und Motivation

Der Ursprung dieser Arbeit liegt in einer interessanten Marktlücke: Es gibt viele plattformübergreifende Apps, die als digitale Einkaufslisten konzipiert sind. Sie bieten verschiedene Konzepte für die Sortierung der Einträge einer Einkaufsliste. Die ideale Reihenfolge der Listeneinträge ist an die Route des Kunden im Laden angepasst. Das bedeutet konkret: die Produkte auf der Liste werden in der Reihenfolge angezeigt, in der sie im Geschäft angetroffen werden, sodass es nicht nötig ist, während eines Einkaufs die ganze Liste im Blick zu behalten. Gewiss ist es möglich, Produkte per Hand passend zu sortieren, jedoch stellt dies einen Arbeitsaufwand seitens des Nutzers dar, der automatisiert werden kann, indem die Reihenfolge der Produkte von der Anwendung anhand Daten aus vorhergehenden Einkaufsvorgängen prognostiziert wird.

Keine Einkaufsliste mit diesem Konzept der automatischen Sortierung ist auf dem Multiplattform-App-Markt vorhanden. Tatsächlich ergibt die Recherche nur eine einzige (native) Anwendung, die mit diesem Feature wirbt. Das bedeutet, Endnutzer auf anderen Plattformen haben keine Option zur Nutzung einer solchen App. Um diese Lücke zu schließen, wird die Multiplattform-App Listipede konzipiert, die Einkaufslisten automatisch so sortiert, dass sie mit der Route des Nutzers durch den Laden übereinstimmen.

Zur Realisierung dieser Anwendung wird sich für das Framework Flutter entschieden, das als eins der jüngsten Multiplattform-Frameworks weniger erforscht ist als etabliertere Frameworks wie React Native oder Xamarin.

1.2 Zielsetzung

Die Umsetzung der Anwendung Listipede stellt eine Möglichkeit dar, den Entwicklungsprozess einer Multiplattform-App aus wissenschaftlicher Sicht zu begleiten, indem er als Gegenstand der Forschung dient: konkret soll aus Entwicklersicht erforscht werden, welche Auffälligkeiten bei der Umsetzung einer Multiplattform-App in Flutter bestehen, sowohl positiv als auch negativ. Die übergeordnete Forschungsfrage der Arbeit lautet somit: Welche Hürden und Chancen bestehen bei der Multiplattform-App-Entwicklung mit Flutter und Dart?

Aus dieser Forschungsfrage ergeben sich Teilfragen: Wie können die Herausforderungen bei der Entwicklung gemeistert werden? Welche Limitierungen oder gar Erleichterungen im Hinblick auf Software-Bibliotheken ergeben sich bei der Verwendung von Flutter? Welche Unterschiede bestehen zur nativen App-Entwicklung?

1.3 Methodik

Um diese Fragen zu beantworten, werden zuerst einige Grundlagen rund um die Entwicklung einer Multiplattform-App mit Flutter geklärt, indem branchenübliche Literatur hinzugezogen wird. Danach werden existierende Anwendungen im Markt der Einkaufslisten-Apps analysiert und daraus funktionale Anforderungen an die App Listipede definiert. Basierend auf den Anforderungen wird ein Konzept für die App erarbeitet, die prototypisch entwickelt wird. Die Anwendung wird anhand ihrer Anforderungen evaluiert. Anhand der Erkenntnisse des gesamten Entwicklungsprozesses wird die Forschungsfrage beantwortet und ein Fazit aus der Arbeit gezogen.

2 Theoretischer Hintergrund

2.1 Usability und Appgestaltung

Dieser Abschnitt soll knapp die essenziellen Aspekte Appgestaltung und Usability erläutern, insbesondere im Hinblick auf die Entwicklung der App Listipede.

Die *International Organization for Standardization* (ISO) definiert in ISO-Norm 9241-11 den Begriff Usability als „Ausmaß, in dem ein System, ein Produkt oder eine Dienstleistung durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen“ (DIN, 2018, S. 9). Ziel der Usability im Kontext der Software-Entwicklung ist es, die Benutzung einer Anwendung so einfach wie möglich zu machen (Jacobsen & Meyer, 2019, S. 31). Der Begriff kann folglich sinngemäß als Benutzerfreundlichkeit oder Bedienbarkeit übersetzt werden und somit eine breite Palette von Konzepten beinhalten.

Dazu gehört eine Appgestaltung/App-Design, das die Bedürfnisse und Erwartungen der Nutzer erfüllt. Eine solche Gestaltung bedeutet unter anderem eine intuitive Benutzeroberfläche (UI), die es ermöglicht, die App ohne Schwierigkeiten zu navigieren. Konsistenz in der Gestaltung und Verwendung von UI-Elementen gewährleistet, dass Benutzer nicht verwirrt werden und sich leicht zurechtfinden können. Flexible Anpassung der Gestaltung im Rahmen des Responsive Design ist entscheidend, damit eine App auf verschiedenen Geräten und Bildschirmgrößen funktioniert und so eine optimale Benutzererfahrung geräteunabhängig gewährleistet wird. Barrierefreiheit bei der Gestaltung von Apps ist ein weiterer wichtiger Aspekt der Usability, der sicherstellt, dass Benutzer unabhängig von individuellen Bedürfnissen und Einschränkungen auf eine Anwendung zugreifen können.

Die Usability einer App ist ein Schlüsselfaktor, der großen Einfluss darauf hat, wie die Anwendung auf dem Markt von Nutzern angenommen wird (Peinert-Elger & Magerhans, 2023, S. 11–12). Usability-Konzepte sind demnach entscheidend für die Entwicklung erfolgreicher mobiler Apps. Die Anwendung in dieser Arbeit ist jedoch als technischer Prototyp konzipiert, der dazu dient, die Machbarkeit und die technischen Herausforderungen und Möglichkeiten der Multiplattform-App-Entwicklung mit Flutter zu untersuchen. Der Schwerpunkt dieser Bachelorarbeit liegt auf der technischen Umsetzung.

Für Leser, die sich eingehender mit den Themen Appgestaltung und Usability befassen möchten, wird die folgende Literatur empfohlen:

Nielsen, J. & Loranger, H. (2006). *Prioritizing Web Usability*. New Riders.

Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web (and Mobile) Usability* (Third edition). New Riders.

Semler, J. & Tschierschke, K. (2019). *App-Design* (2., aktualisierte und erweiterte Auflage). *Rheinwerk Design*. Rheinwerk Verlag.

Jacobsen, J. & Meyer, L. (2019). *Praxisbuch Usability und UX: Was jeder wissen sollte, der Websites und Apps entwickelt. Onleihe. E-Book*. Rheinwerk Verlag; divibib GmbH.

2.2 Multiplattform-App-Entwicklung

Die dynamische Entwicklung mobiler Technologien hat zu einer breiten Vielfalt an Geräten geführt, auf denen Apps ausgeführt werden können. In den Jahren vor diesem rasanten Fortschritt waren Mobiltelefone in erster Linie auf grundlegende Funktionen wie Anrufe und Kurznachrichten beschränkt. Die Transformation begann mit der Einführung von Smartphones, die erweiterte Möglichkeiten bieten und schließlich zu einem breiten Spektrum von Anwendungen führen.

Mit dem Debüt von Apples iOS etablierte sich die erste von zwei dominanten Plattformen in der Welt der mobilen Anwendungen. iOS wurde 2007 als Betriebssystem für das erste iPhone eingeführt und setzte den Grundstein für eine intuitive Touchscreen-Interaktion und das bis dahin fremde Konzept von mobilen Anwendungen – die Apps, zusammen mit dem ebenso neuen Konzept eines zentralisierten App Store, aus dem man diese neuartigen Anwendungen beziehen soll (Apple, 2007). Es ist erwähnenswert, dass Apple Software wie iOS ausschließlich auf Apple Geräten läuft und somit ein geschlossenes Ökosystem bildet.

Die zweite bahnbrechende Einführung ist das Betriebssystem Android, das von Google entwickelt wurde und direkt im Folgejahr 2008 auf den Markt kam, ebenfalls mit seinem eigenen Markt für Apps (Morrill, 2008). Es bot im Gegensatz zu iOS eine offene Plattform für Entwickler und Hersteller, was zu einer Vielzahl von Geräten führte, auf denen das Betriebssystem läuft. Neben iOS und Android gab es auch andere Plattformen, die versuchten, Fuß auf dem mobilen Markt zu fassen; jedoch ergeben iOS und Android zusammen heute weltweit etwa 99,9% des Marktanteils der mobilen Betriebssysteme und bilden damit die einzigen de facto Optionen für Smartphone-Plattformen (IDC, 2023).

Diese beiden Plattformen setzen unterschiedliche Programmiersprachen und Frameworks ein, um native Apps zu erstellen. Traditionell wurden mobile Anwendungen für jede Plattform separat entwickelt. Native Apps für Apples iOS werden beispielsweise mit Objective-C oder Swift und den Apple-eigenen Frameworks erstellt, während Android-Anwendungen hauptsächlich auf der Java-Plattform mit Java oder Kotlin basieren (Google, 2023).

Möchte man eine solche native Anwendung auf anderen Plattformen anbieten, um bspw. den potenziellen Absatzmarkt zu erhöhen, stellt der Prozess des Überführens von Code ein Problem dar: Ein manuelles Übersetzen zwischen den Programmiersprachen ist zeitaufwändig und fehleranfällig, was eine Erhöhung sowohl von Arbeitsbelastung als auch von Entwicklungskosten zur Folge hat. Da-

bei ist das Anbieten von Apps auf mehreren Geräten äußerst sinnvoll, da Erwachsene heutzutage nicht nur oft ein Gerät wie das Smartphone, Tablet oder den PC verwenden, sondern auch mehrere verschiedene Geräte täglich (Robinson, 2014).

Um also diese Herausforderungen zu bewältigen, entstand mit zunehmender Beliebtheit der beiden konkurrierenden Plattformen im Smartphone-Marktsegment ein Bedarf an plattformübergreifenden Lösungen. Es wurden verschiedene Technologien erarbeitet, die es ermöglichen, Apps für mehrere Plattformen gleichzeitig zu entwickeln. Grundsätzlich lassen sich hierbei drei Ansätze unterscheiden:

Die erste Möglichkeit, eine App plattformübergreifend zu gestalten, ist die sog. mobile Webanwendung. Diese ist streng genommen keine App an sich, sondern eine Client-Server-Anwendung, die entwickelt wurde, um die nativen Anwendungen des Host-Betriebssystems so genau wie möglich nachzuahmen, jedoch vom Gerät über einen Web-Browser aufgerufen wird und somit keine Installation erfordert (Jobe, 2013, S. 28; Rohr, 2018, S. 1–2). Folglich benutzt sie Web-Standards wie JavaScript oder HTML5 und kann theoretisch von jedem Gerät ausgeführt werden, dem ein Browser und eine ausreichende Internetverbindung zur Verfügung steht. Was eine solche Anwendung zu einer *mobilen* Web-App macht ist also vielmehr die Tatsache, dass sie (auch) zur Benutzung auf einem Mobilgerät konzipiert ist. Das bedeutet unter anderem, dass alle Elemente der Benutzeroberfläche optisch und ergonomisch an das mobile Gerät angepasst sind, um ein möglichst nahtloses Nutzererlebnis zu bieten (Serrano et al., 2013, S. 24). Diese Tatsache unterscheidet die Web-App ferner von einer herkömmlichen Webseite, welche bekanntlich eine vergleichbar große Bandbreite an reinen Funktionen liefern kann wie eine App.

Da eine solche Anwendung in den allermeisten Anwendungsbereichen eine Verbindung zum Internet voraussetzt, bildet diese einen technischen Flaschenhals, wenn beispielsweise der Empfang schlecht ist. Auch der Zugriff über einen Web-Browser stellt ein Problem dar, da dadurch nur jene Komponenten der jeweiligen Hardware durch die App ansteuerbar sind, auf die der Browser Zugriff hat (Jobe, 2013, S. 28). Somit sind die realistischen Anwendungsbereiche einer mobilen Web-App beschränkt.

Es gibt eine Unterkategorie der Web-App, die genug Alleinstellungsmerkmale bietet, um an dieser Stelle gesondert erwähnt zu werden: Die *Progressive Web App* (PWA).

Eine PWA enthält eine Manifest-Datei, die Informationen über die Website wie Icons und Farben enthält und verwendet wird, um das Hinzufügen der App zum Startbildschirm des Geräts zu erlauben. So kann auf PWAs genau wie auf installierte native Apps zugegriffen werden (Hume, 2018, S. 5).

Weiter zeichnen sich PWAs durch die Verwendung von JavaScript Service Worker aus, um Push-Benachrichtigungen zu erlauben und einmal abgerufene Inhalte

auch offline zur Verfügung zu stellen (Hume, 2018, 5–6). PWAs können also kurzum als verbesserte Web-Apps betrachtet werden; sie stellen keinen prinzipiell anderen Ansatz zur Multiplattform-App-Entwicklung dar.

Der zweite Ansatz ist die Hybrid-App: Sie basiert auf der Verwendung von Frameworks, die mit Web-Technologien arbeiten (also ein „Hybrid" zwischen nativer und Web-App). Ein bekanntes Framework, das auf dieser Technologie basiert, ist Apache Cordova. Bei der Entwicklung solcher plattformübergreifenden Apps werden Webseiten in die App eingebettet und über eine Browser-Komponente präsentiert (Q. Huynh et al., 2017, S. 53). Der tatsächliche Code wird in JavaScript geschrieben, während die Ausführung unbemerkt für den Nutzer innerhalb des nativen Webbrowsers der spezifischen Plattform erfolgt, etwa Safari (iOS) oder Google Chrome (Android). Über eine API können native Komponenten angesteuert werden, um auf die spezielle Hardware des Geräts zuzugreifen. Jedoch hat auch diese Vorgehensweise einige Nachteile, darunter die Einschränkung auf gemeinsam verfügbare Features beider Plattformen sowie eine eingeschränkte Performance aufgrund der Verwendung eines Browsers als Ausführungsumgebung. Dennoch kann dieser Ansatz nützlich sein, da er, ähnlich wie Webanwendungen, Web-Entwicklern einen Einstieg in die Welt der App-Entwicklung ermöglicht.

Der dritte Ansatz, die Cross-/Multiplattform-App, verfolgt eine plattformunabhängige Methode, bei der mit spezifischen Sprachen und Frameworks entwickelt wird, die anschließend in die gewünschte Plattform cross-kompiliert werden. Dies soll zu performanteren Apps führen, da diese direkt auf der Zielplattform ausgeführt werden können (Raj & Tolety, 2012). 2013 ist ein bis heute bekannter Vertreter dieser Technologie erschienen: Xamarin in der Version 2.0 von Microsoft (Friedman, 2013). Xamarin ermöglichte es mit diesem Update Entwicklern erstmalig, „echte" plattformübergreifende Apps (also solche, die nicht wie in den beiden vorangestellten Herangehensweisen auf die eine oder andere Weise Web-Browser-basiert sind) für iOS und Android zu erstellen, indem sie eine gemeinsame Codebasis verwenden.

Die Stärken von Xamarin liegen in der hervorragenden Performance, da die Apps direkt auf der Zielplattform ausgeführt werden können. Dies steht im Gegensatz zu den anderen Ansätzen, bei denen JavaScript und Webtechnologien genutzt werden, die eine zusätzliche Abstraktionsschicht einführen und folglich die Performance einschränken können. Ein weiterer Vorteil von Xamarin gegenüber den bisherigen Methoden der Crossplattform-Entwicklung ist die tiefe Integration in die jeweiligen Plattformen. Entwickler können native APIs und Funktionalitäten nutzen, um auf die Hardware des Geräts zuzugreifen und damit ein nahtloses Nutzererlebnis zu schaffen, ohne sich wie bei Hybrid-Apps auf gemeinsame Features beschränken zu müssen. Darüber hinaus bietet Xamarin die Möglichkeit, native Benutzeroberflächen zu erstellen, die sich in das Design der jeweiligen Plattform integrieren und somit ein für Nutzer vertrautes Erlebnis bieten.

Trotz dieser Stärken bringt die Verwendung von Xamarin auch einige Herausforderungen mit sich. Einer der Hauptkritikpunkte ist die mögliche Erzeugung von großen Dateien, da die gemeinsame Codebasis in C# geschrieben und dann in nativen Code kompiliert werden muss. Dies kann im Kontext der App-Entwicklung zu größeren Apps führen, die folglich den Speicherplatzbedarf erhöhen (Nawrocki et al., 2021, S. 26).

Ein weiterer prominenter Vertreter im Bereich der Crossplattform App-Entwicklung ist React Native, ein Open-Source-Framework, das von Meta (Ehemals Facebook) entwickelt wurde und 2015 erschien. Dieses Framework ermöglicht die Erstellung nativ wirkender mobiler Apps mithilfe von JavaScript und React, einer beliebten JavaScript-Bibliothek zur Erstellung von Benutzeroberflächen (Occhino, 2015). Die Idee hinter React Native ist es, die Effizienz und Performance nativer Apps mit den Entwicklungsprinzipien des Web-Entwicklungsprozesses zu kombinieren. Entwickler können dabei auf ggf. vorhandenes Wissen in JavaScript und React aufbauen, um plattformübergreifende Apps zu erstellen, die eine merklich bessere Performance aufweisen als Xamarin (Nawrocki et al., 2021, S. 23).

Ein besonderer Ansatz von React Native liegt in der Verwendung von sogenannten „Native Components“. Hierbei handelt es sich um vorgefertigte Bedienelement-Komponenten, die direkt auf die nativen Komponenten der Zielplattform abgebildet werden. Dadurch können Entwickler UI-Elemente erstellen, die sich nahtlos in die jeweilige Plattform integrieren und ein entsprechend gewohntes Nutzererlebnis bieten (Meta, 2023). Ein weiterer Vorteil von React Native ist die „Hot Reload“-Funktion, die es ermöglicht, Änderungen im Quellcode in Echtzeit zu sehen, ohne eine App neu starten zu müssen. Dies beschleunigt den Entwicklungsprozess und erleichtert das Debugging erheblich (Bigio, 2016).

Allerdings gibt es auch Herausforderungen bei der Verwendung von React Native. Beispielsweise kann die Abhängigkeit von JavaScript in manchen Fällen zu einer beachtlich langsameren Ausführung im Vergleich zu nativem Code führen (Nawrocki et al., 2021, S. 23).

Insgesamt bieten Frameworks mit plattformunabhängigem Ansatz eine vergleichsweise effiziente Möglichkeit, Apps für verschiedene Plattformen gleichzeitig zu entwickeln und Code-Wiederverwendung zu fördern. Deshalb sind Frameworks, die in diese Kategorie fallen, besonders interessant für Entwickler. Ein weiteres solches Framework ist Flutter, welches ein zentraler Gegenstand dieser Arbeit ist und markante Unterschiede gegenüber Xamarin und React Native aufweist. Bereits vorhandene Vergleiche von Flutter, React Native und Xamarin zeigen, dass Flutter unter Umständen sowohl bei der CPU-Auslastung als auch bei der App-Größe und Anlaufzeit von Apps vergleichbar gut bis besser als eine oder beide dieser Frameworks abschneidet (Nawrocki et al., 2021, S. 23–26).

2.3 Flutter und Dart

Um die Entwicklung von Flutter darzustellen ist es sinnvoll, zuerst die von Flutter benutzte Programmiersprache, Dart, zu erklären.

Dart wurde 2011 von Google veröffentlicht. Das Ziel hinter der Entwicklung war eine flexible, strukturierte und umgebungsunabhängig leistungsstarke Sprache für die Web-Programmierung, die für Entwickler leicht zu erlernen und benutzen ist (Bak, 2011). Insbesondere sollte Dart innerhalb der Branche mit dem damaligen Spitzenreiter JavaScript konkurrieren (Shankland, 2011). Tatsächlich wurde dieses Ziel 2015 aufgegeben, indem sich der Fokus auf das Kompilieren von Dart auf JavaScript wandelte (Bak & Lund, 2015). Im Kontext der plattformübergreifenden App-Entwicklung hat Dart als Programmiersprache aber eine besondere Bedeutung erlangt, indem sie die Grundlage für das Flutter-Framework bildet.

Flutter verwendet Dart und wird ebenfalls von Google entwickelt. Es wurde 2017 veröffentlicht (Bracken, 2017). Es ist ein Multiplattform-Framework mit demselben grundsätzlichen Crossplattform-Ansatz, den auch Xamarin und React Native zuvor verfolgten, mit dem Entwickler ansprechende, bei Bedarf nativ wirkende Benutzeroberflächen für mobile, Web- und Desktop-Plattformen erstellen können.

Im Vergleich zu Xamarin und React Native bietet Flutter einige signifikante Unterschiede. Während Xamarin C# als Programmiersprache verwendet und React Native auf JavaScript basiert, ist Flutter speziell für die Entwicklung mit Dart optimiert und ermöglicht eine performante Ausführung der Anwendungen. Ein weiterer Unterschied besteht in der Art und Weise, wie die Benutzeroberflächen erstellt werden. Während React Native auf nativen UI-Komponenten der jeweiligen Plattform aufbaut und Xamarin eine direkte Verbindung zu den nativen APIs herstellt, erstellt Flutter seine Benutzeroberflächen mithilfe einer eigenen Sammlung von Widgets. Diese Widgets sind anpassbare Bausteine, mit denen Entwickler komplexe Benutzeroberflächen erstellen können. Da Flutter auf Widgets basiert, bietet es eine hohe Flexibilität bei der Gestaltung der Benutzeroberfläche, ohne sich an die nativen Komponenten der Plattform binden zu müssen. Ein besonderes Merkmal von Flutter hierbei ist die Verwendung einer Grafik-Engine (Impeller bei iOS und Skia bei Android und Web), die eine schnelle und flexible Darstellung von UI-Elementen ermöglicht (Chisholm, 2022). Flutter bietet zusätzlich eine ‚Hot Reload"-Funktion ähnlich der in React Native.

Trotz dieser Vorteile bringt auch Flutter einige Herausforderungen mit sich. Da Dart nicht so weit verbreitet ist wie andere Programmiersprachen wie JavaScript oder C#, könnte die Einarbeitung in diese Sprache für einige Entwickler eine zusätzliche Hürde darstellen. Darüber hinaus ist die Verfügbarkeit von Drittanbieter-Bibliotheken und Modulen möglicherweise eingeschränkter als bei etablierteren Programmiersprachen.

2.4 Firebase

Firebase ist eine umfassende Entwicklungsplattform, die 2012 erschienen ist (Metz, 2012). Sie eignet sich bestens für die Integration mit Flutter und Dart, da sie ebenso von Google entwickelt wird. Firebase soll es App-Entwicklern ermöglichen, schnell und einfach Web- und Mobile-Anwendungen zu erstellen, zu verbessern und zu skalieren. Zu diesem Zweck bietet die Plattform eine Vielzahl von Werkzeugen und Diensten. Im Folgenden werden diejenigen Funktionen kurz umrissen, die für die vorliegende Arbeit von besonderer Relevanz sind:

Ein wichtiger Dienst von Firebase ist Firestore, eine flexible und skalierbare Cloud-Datenbanklösung, die eine leistungsstarke Grundlage für die Speicherung und Abfrage von Daten darstellt. Dies kann dazu genutzt werden, die Einkaufslisten und andere Daten der Benutzer zu speichern und sie auf einfache Weise abzurufen.

Ein besonders bedeutender Aspekt ist die Echtzeit-Datenbank, die es ermöglicht, Daten in Echtzeit zwischen den verschiedenen Nutzern einer App zu synchronisieren. Dies ist von besonderem Nutzen für Anwendungen wie die geplante Einkaufslisten-App, in der mehrere Benutzer gleichzeitig auf die Liste zugreifen und Änderungen vornehmen können sollen.

Des Weiteren stellt Firebase auch eine Authentifizierungsfunktion bereit, die es ermöglicht, Nutzer sicher in der App anzumelden und das Account-System von Firebase zu verwalten. Dies ist essenziell, um die Privatsphäre der Nutzer zu gewährleisten und den Zugriff auf persönliche Daten zu schützen.

2.5 Einkaufslisten-App-Marktanalyse

Um ein Bild über das bestehende Angebot der Einkaufslisten-Apps zu verschaffen, werden zunächst beliebte Anwendungen in diesem Marktsegment recherchiert (vgl. **Tabelle 1**).

App	Bring!	Die Einkaufsliste	Listonic	pon
Downloads (Google Play Store)	>5 Mio.	>1 Mio.	>10 Mio.	-
⌀ Bewertung (Google Play Store)	4,5/5	4,7/5	4,4/5	-
Anzahl Rezensionen (Google Play Store)	~128.000	~39.000	~260.000	-
⌀ Bewertung (iOS App Store)	4,8/5	4,8/5	4,7/5	4,7/5
Anzahl Rezensionen (iOS App Store)	~9000	~6000	~8000	~25.000

Tabelle 1: Beliebte Einkaufslisten-Apps

Diese Anwendungen werden heruntergeladen und verwendet. Ich untersuche ihre Funktionen sowie meine Eindrücke zu deren Benutzerfreundlichkeit. Diese Vorbereitung erleichtert zusätzlich die Formulierung des Anforderungskatalogs,

da hier bereits im Vorfeld sowohl auffällig hilfreiche Funktionalitäten als auch mögliche Probleme bei dieser Art von Anwendung erkennt werden können.

2.5.1 Bring! Einkaufsliste & Rezepte

Die Einkaufslisten-App „Bring!“ erfreut sich großer Beliebtheit sowohl auf iOS als auch auf Android-Plattformen. Die Startseite der App präsentiert die Auswahl der Einkaufslisten (**Abbildung 1**). Eine untere Navigationsleiste ermöglicht es, zwischen den Listen und anderen Bildschirmen zu wechseln. Das Hinzufügen von Artikeln erfolgt über eine Suchleiste am unteren Bildschirmrand, die bei Antippen erweitert wird und Produktempfehlungen anbietet. Die Artikel können in dieser Ansicht Details und Mengenangaben erhalten. Ein Artikel kann mehrfach als separate Einträge in der Liste auftauchen, wenn sich die Beschreibung (z.B. „vegan“ oder „Bio“) oder Menge unterscheidet. Die Darstellung der Artikel kann entweder in einer konventionellen Listenansicht (**Abbildung 2**) oder als Kacheln erfolgen. Die Sortierung der Listeneinträge kann anhand von Kategorien vorgenommen werden, wobei die Reihenfolge manuell angepasst werden kann. Ein Verschieben von Artikeln innerhalb einer Kategorie ist nicht möglich.

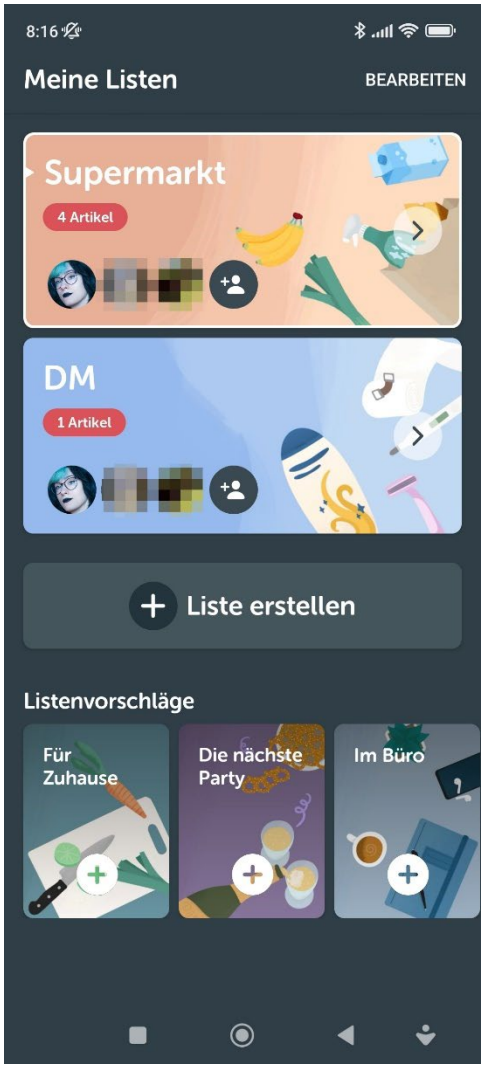

Abbildung 1: Bring! Startbildschirm

Abbildung 2: Bring! Einkaufsliste

Das Abhaken von Artikeln erfolgt standardmäßig durch Antippen. Abgehakte Artikel werden im Verlauf unterhalb der eigentlichen Liste angezeigt, wobei ältere Einträge gelöscht werden, sobald neue hinzugefügt werden. Die App ermöglicht auch kollaboratives Arbeiten, indem Einladungen zu Listen per Link versendet werden. Diese Funktion erfordert einen Benutzeraccount. Zusätzlich werden Benachrichtigungen an alle Mitwirkenden versendet, sobald eine Änderung an der Liste vorgenommen wird.

Ein besonderes Merkmal der App sind die Rezepte, die eher als Zutatenlisten betrachtet werden können. Diese können einfach zur Liste hinzugefügt werden, indem die benötigten Zutaten ausgewählt und anschließend in die Einkaufsliste überführt werden. Die App bietet sowohl eigene Rezepte als auch Vorschläge von verschiedenen Webseiten an. Zudem sind Prospekte in der App eingebunden.

Insgesamt betrachtet sind die grundlegenden Funktionen der „Bring!"-App zufriedenstellend. Es gibt jedoch einige Punkte, die potenziell verbessert werden könnten. Beispielsweise kann das Abhaken von Artikeln versehentlich durch einfaches Antippen geschehen. Die Sortierungsoptionen könnten optimiert werden, da das Verschieben von Artikeln innerhalb von Kategorien fehlt. Kategorien müssen manuell pro Liste sortiert werden, wodurch beim Einkauf in verschiedenen Märkten und deren unterschiedlichen Regalverteilungen separate Listen notwendig wären. Die Navigation durch die untere Navigationsleiste ist nicht immer intuitiv, da diese erst auftaucht, sobald man eine Liste angewählt hat.

Die Möglichkeit zur gemeinsamen Nutzung von Listen empfinde ich als nützlich, insbesondere die Benachrichtigungsfunktion über Änderungen. Die Rezepte bieten eine praktische Ergänzung.

2.5.2 Die Einkaufsliste

Eine weitere App ist „Die Einkaufsliste", die auf Android und iOS verfügbar ist.

Die App startet mit der zuletzt genutzten Einkaufsliste (**Abbildung 3**). Alle Listen tauchen direkt in einer Navigationsschublade auf, die durch Ziehen von der linken Bildschirmkante oder Tippen auf das Hamburger-Menü-Symbol links oben geöffnet werden kann (**Abbildung 4**). Die Einkaufsliste selbst zeigt die Artikel untereinander in einer Listenansicht an. Das Hinzufügen von Artikeln erfolgt durch eine Suchleiste oben auf dem Bildschirm, wobei die Eingabe von Details und Mengen erst nach dem Hinzufügen möglich ist. Mehrere Variationen eines Artikels können nicht mehrfach in derselben Liste vorhanden sein; es sind separate Artikel erforderlich.

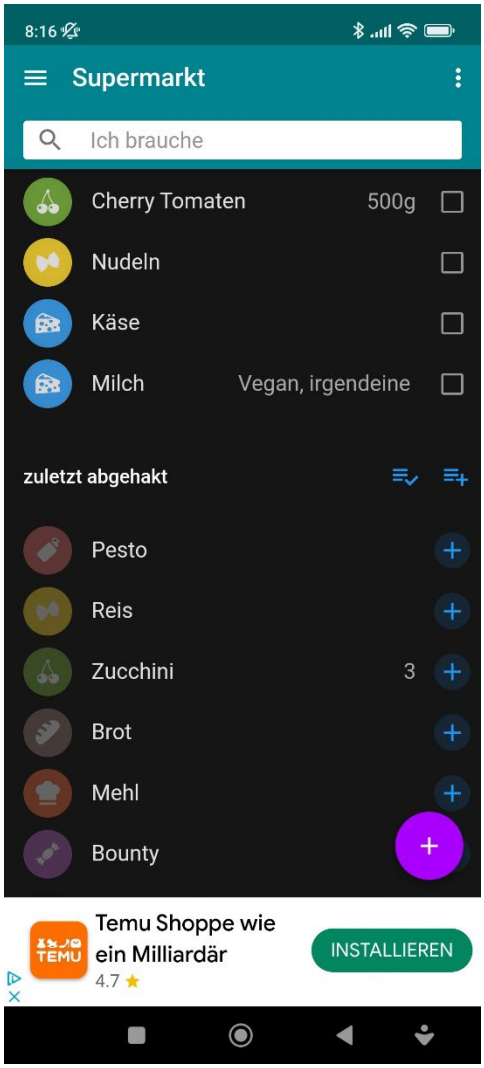

Abbildung 3: Die Einkaufsliste Listenansicht

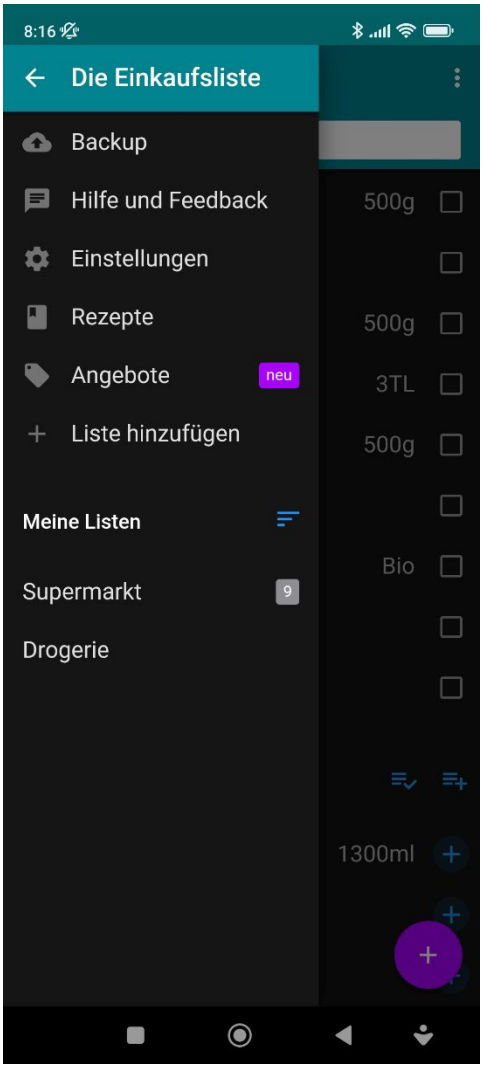

Abbildung 4: Die Einkaufsliste Schubladenmenü

Die Sortierung erfolgt auch hier anhand von Kategorien, die durch Symbole gekennzeichnet sind. Die Verschiebung von Kategorien ist möglich, jedoch können Einzelartikel auch hier nicht innerhalb Kategorien verschoben werden. Das Abhaken von Artikeln erfolgt durch Antippen eines kleinen Kontrollkästchens auf der rechten Seite. Abgehakte Artikel verbleiben wie bei „Bring!“ als Verlauf unter der Liste. Die Möglichkeit, Listen gemeinsam zu nutzen, beschränkt sich darauf, Die Einträge einer Liste als reinen Text zu versenden.

Rezepte bestehen neben einer Zutatenliste auch aus einer Anleitung mit Schritten; Zutaten können entweder als komplette Liste oder einzeln hinzugefügt werden. Prospekte sind ebenfalls in die App integriert.

Die subjektive Erfahrung mit dieser App zeigt, dass die Kästchen zum Abhaken etwas klein sind. Dies kann auch die Zuordnung zu den jeweiligen Artikeln erschweren. Die Eingabe von Details und Mengen wird als umständlich empfunden. Das Fehlen einer effektiven Funktion zur gemeinsamen Listenführung mindert die Nützlichkeit dieser App in diesem Bereich.

2.5.3 Listonic

Eine weitere App in der Marktanalyse dieser Arbeit ist „Listonic“, die auf Android, iOS und als Web-App verfügbar ist.

Die Startseite der App zeigt die Auswahl der Einkaufslisten, wobei Fortschrittsbalken angezeigt werden (**Abbildung 5**). Die Navigation erfolgt über eine ähnliche Schublade wie bei „Die Einkaufsliste“. Artikel werden ebenfalls in Listen untereinander angezeigt (**Abbildung 6**). Das Hinzufügen von Artikeln erfolgt über einen Floating Action Button, der einen Bildschirm mit einer Suchleiste öffnet. Die Mengenangabe eines hinzugefügten Artikels kann durch erneutes Drücken auf „+“ um eins erhöht werden, während die detaillierte Mengen- und Einheitenangabe erst nach dem Hinzufügen eines Artikels in der Liste möglich ist. Duplikate desselben Artikels sind nicht möglich.

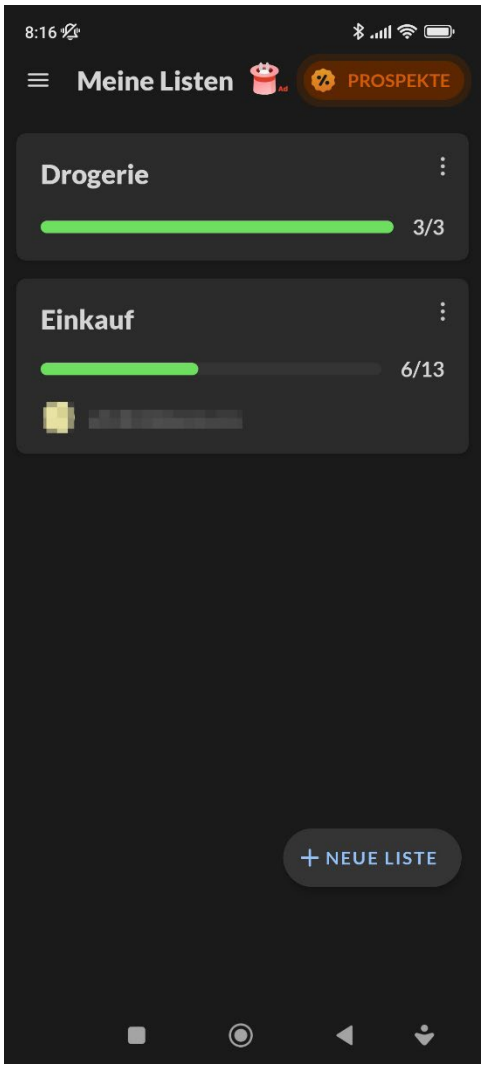

Abbildung 5: Listonic Startbildschirm

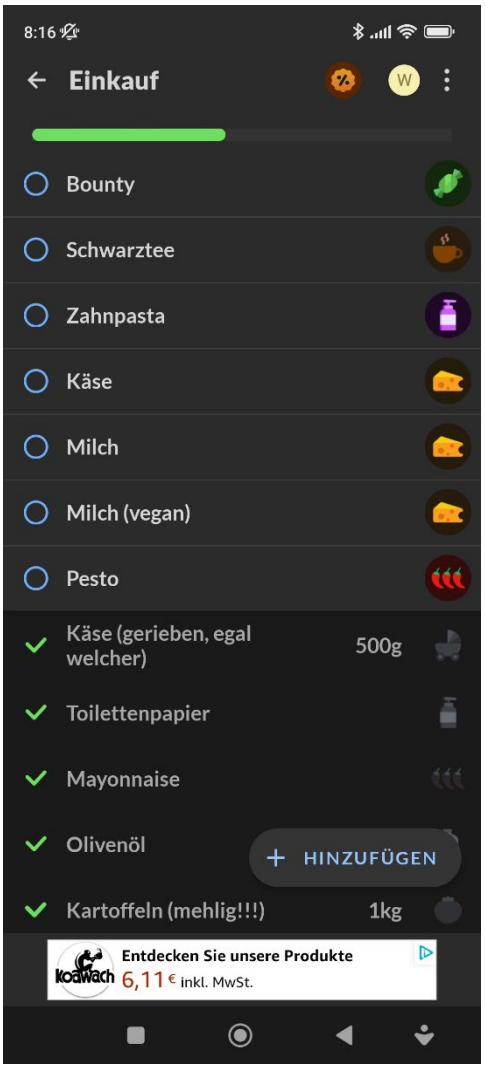

Abbildung 6: Listonic Einkaufsliste

Das Sortieren von Artikeln kann sowohl nach Kategorien als auch alphabetisch oder benutzerdefiniert per Drag-and-drop erfolgen. Abhaken ist entweder durch Tippen auf ein Kontrollkästchen oder durch Wischen nach rechts möglich. Abgehakte Artikel verbleiben in der Liste und werden als Fortschritt in der Listenan-

sicht angezeigt. Das Entfernen aller erledigten Artikel erfordert das Navigieren in ein Menü.

Die Funktion zur gemeinsamen Listenführung ist über Links möglich, wobei auch Benachrichtigungen an die Beteiligten versendet werden können. Zudem besteht die Option, die Liste als Klartext zu exportieren. Prospekte können ebenfalls in der App eingesehen werden.

Nach meiner subjektiven Einschätzung der „Listonic“-App ist die visuelle Zuordnung der Kontrollkästchen zu den jeweiligen Listeneinträgen durch klare Linientrennung gut gelöst. Die Aufbewahrung abgehakter Artikel in der Liste könnte im Vergleich zu anderen Apps als weniger praktisch empfunden werden, insgesamt bietet die Anwendung aber völlig akzeptable Nutzbarkeit.

2.5.4 pon - smarte Einkaufsliste

Exklusiv für iOS ist die App „pon - smarte Einkaufsliste“.

Die Startseite zeigt die Einkaufslisten und bietet einen Button für Einstellungen (**Abbildung 7**). Die Einkaufslistenansicht zeigt die Artikel untereinander (**Abbildung 8).** Das Hinzufügen neuer Artikel erfolgt über einen "+"-Button in der unteren Menüleiste.

Beim Hinzufügen eines Artikels ist standardmäßig eine produktabhängige Einheit ausgewählt (z. B. Wasser - Flasche, Gewürzgurken - Glas), die mit einem Cupertino-Picker angepasst werden kann. Zudem muss der Primäre Markt, in dem ein Artikel gekauft wird (oder „Beliebig”), festgelegt werden, und optional können auch sekundäre Märkte hinterlegt werden. Eigenschaften von Artikeln können durch An-/Ausschalter aktiviert oder deaktiviert werden. Hier können auch Preise, Fotos und Aktionen hinterlegt werden.

Abbildung 7: pon Startbildschirm

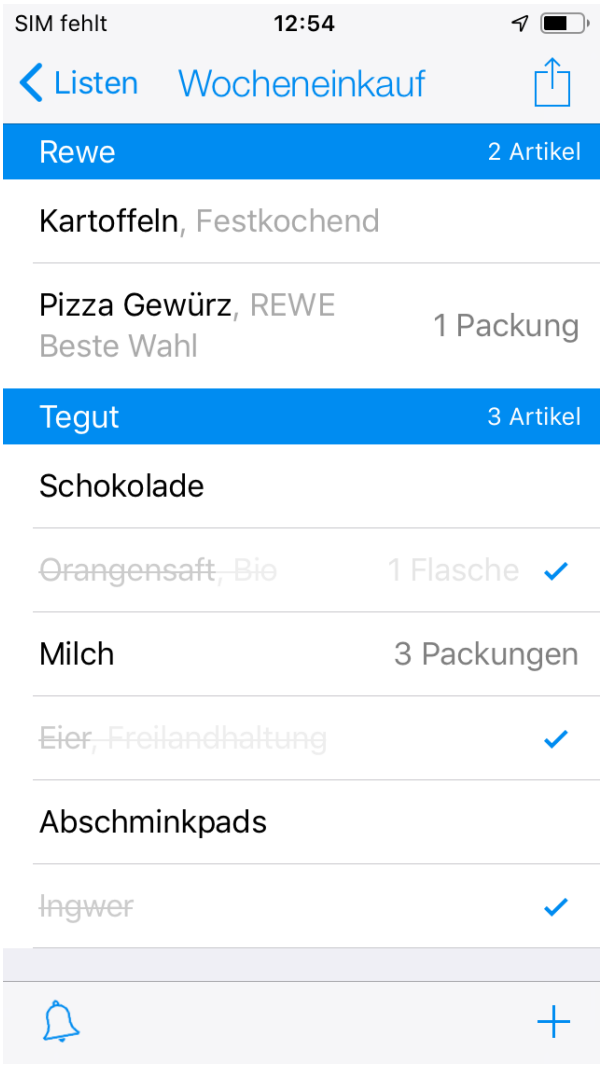

Abbildung 8: pon Einkaufsliste

Durch Wischen nach links werden Bearbeitungsschaltflächen für die Artikel aufgedeckt. Das Abhaken der Artikel erfolgt durch Antippen. Das Entfernen von abgehakten Artikeln erfordert das Herunterziehen der Einkaufsliste oder eine Schüttelgeste. Einmal entfernte Einträge werden nicht in einen Verlauf verschoben. Die Möglichkeit zur gemeinsamen Nutzung von Einkaufslisten mittels Accounts ist gegeben.

Diese Anwendung ist besonders relevant für meine geplante App, da sie eine „smarte“ Listenführung verspricht, die Einträge anhand der Reihenfolge, in der sie abgehakt werden, sortiert. Diese Funktion ist jedoch nur für Artikel verfügbar, die einem Markt zugeordnet sind. Alternativ können Artikel manuell per Drag-and-Drop bewegt oder gänzlich nach alphabetischer Reihenfolge oder Erstellungsdatum sortiert werden. Artikel in der Einkaufsliste werden nach den Primärmärkten gruppiert.

Ein weiteres besonderes Merkmal der App ist das Geo-Fencing-Feature, das recht simpel gestaltet ist: Beim Abhaken eines Artikels in einer Liste erscheint ein Popup-Fenster, ob sich der Nutzer gerade im Laden befindet, um den Ort mit dem Markt in Verbindung zu bringen. Wenn sich der Benutzer zu einem späte-

ren Zeitpunkt wieder in der Nähe des Geschäfts befindet, erscheint ein Popup-Fenster zur Erinnerung, wenn sich noch Produkte in einer Liste befinden, die mit dem entsprechenden Standort in Verbindung gebracht wurden. Artikel, die einen bestimmten Standort als sekundären Markt haben, werden automatisch in die Unterkategorie des entsprechenden Marktes verschoben, sobald an diesem Ort eingekauft wird.

Die subjektive Erfahrung mit dieser App zeigt, dass das Befüllen und Bearbeiten von Artikeln im Vergleich zu anderen Anwendungen weniger benutzerfreundlich ist. Das Hinterlegen von Märkten macht Sinn für die „smarte“ Sortierfunktion, jedoch kann es als umständlich empfunden werden, da für jeden Artikel das manuelle Hinterlegen von Primär- und Sekundärmärkten erforderlich ist. Auch die Benutzung während des Einkaufs wird als suboptimal eingestuft, da abgehakte Artikel in der Liste verbleiben und nicht wie in anderen Apps nach unten verschoben werden.

2.5.5 Fazit

Zusammenfassend ist im Rahmen der Marktanalyse festzuhalten, dass das Erstellen und Nutzen von Einkaufslisten bei allen untersuchten Anwendungen grundsätzlich zufriedenstellend funktioniert. Die Details spielen hierbei eine weniger entscheidende Rolle. Die Möglichkeit der gemeinsamen Nutzung von Listen durch mehrere Benutzer wird als besonders nützlich empfunden, sofern die Liste nicht alleine genutzt wird. Die Benachrichtigungsfunktion über Änderungen in einer Liste erscheint hier als eine sinnvolle Ergänzung.

Die automatische Artikel-Sortierung wird als wünschenswert erachtet und ist in diesem Marktsegment kaum vertreten. Eine Umsetzung dieser Funktion erfordert zwangsläufig die Einbeziehung verschiedener Märkte.

Die Integration von Rezepten bzw. Zutatenlisten wird als sinnvoller Mehrwert angesehen. Die Umsetzung dieses Features erscheint unkompliziert.

Die Einbindung von Prospekten in Einkaufslisten-Apps wird zwar häufig angeboten, wird jedoch bewusst ausgelassen, um den Arbeitsaufwand im Rahmen dieser Bachelorarbeit nicht übermäßig zu erhöhen.

2.6 Anforderungskatalog

Im folgenden Abschnitt werden die erarbeiteten Anforderungen an die App in Form eines Anforderungskatalogs konkret festgelegt. Er fungiert also im Fall dieser Arbeit als eine Art vereinfachtes kombiniertes Lasten- und Pflichtenheft. Diese Dokumente würden im Allgemeinen in der Planungsphase der Softwareentwicklung vom Auftraggeber (Lastenheft) bzw. in der Definitionsphase vom Auftragnehmer (Pflichtenheft) formuliert werden, um die genauen funktionellen Inhalte einer geplanten Anwendung zu konkretisieren (Metzner, 2020, 47-48, 62).

Da Listipede nicht aus einem Auftrag der freien Wirtschaft mit Auftraggeber entsteht, wird von der Formulierung eines tatsächlichen Lasten- bzw. Pflichtenheftes zum Zweck der Anforderungsdefinition abgesehen.

Die Anforderungen werden mit dem Kürzel „FA“ für Funktionale Anforderung und einer Zahl definiert. Nachdem die App entwickelt wurde, werden diese Anforderungen auf Vollständigkeit und Benutzerfreundlichkeit evaluiert, um die Anwendung zu beurteilen und mögliche Verbesserungen für die Zukunft zu identifizieren.

2.6.1 Anforderung FA-1

Ziffer	FA-1
Beschreibung	Die Anwendung muss das Anlegen und Verwalten von Einkaufslisten erlauben.
Zielsetzung	Einkaufslisten sollen mittels Datenbanken vom Nutzer erstellt und verwaltet werden können. Produkte/Artikel müssen zur Liste hinzugefügt, verwaltet und abgehakt werden können. Nutzer sollen neue Produkte definieren bzw. voreingestellte Produkte anpassen können.
Priorität	Hoch

2.6.2 Anforderung FA-2

Ziffer	FA-2
Beschreibung	Die Anwendung soll einen gemeinsamen Zugriff auf Listen von mehreren Personen erlauben.
Zielsetzung	Eine Nutzerverwaltung soll mittels Accounts erfolgen, welche Nutzer sich vor Benutzung der App einrichten. Das damit verbundene Authentifizierungssystem muss vorhanden sein. Verschiedene Nutzer sollen gemeinsam auf Listen zugreifen können, indem sie sich gegenseitig einladen.
Priorität	Hoch

2.6.3 Anforderung FA-3

Ziffer	FA-3
Beschreibung	Die Anwendung soll die Artikel auf einer Liste automatisch an die Reihenfolge der durchlaufenen Route im Markt anpassen.
Zielsetzung	Die Sortierung soll sich an vergangenen Einkäufen orientieren. Hierfür ist die Reihenfolge, in der Artikel von einer Liste gestrichen wurden, ausschlaggebend.
Priorität	Hoch

2.6.4 Anforderung FA-4

Ziffer	FA-4
Beschreibung	Die Anwendung soll das Einbeziehen verschiedener Märkte erlauben, um die Sortierung problemlos an die Anordnung der Regale beliebiger Filialen anzupassen und Produkte mit bestimmten Märkten zu assoziieren.
Zielsetzung	Märkte sollen vom Nutzer zu einer Datenbank hinzugefügt und bei Produkten hinterlegt werden können.
Priorität	Hoch

2.6.5 Anforderung FA-5

Ziffer	FA-5
Beschreibung	Die Anwendung soll sowohl auf Android als auch auf iOS funktionieren.
Zielsetzung	Mithilfe der Multiplattform-Funktionen von Flutter sollen alle Teile der App problemlos auf beiden Plattformen funktionieren, ohne die App für die Betriebssysteme separat zu programmieren.
Priorität	Hoch

2.6.6 Anforderung FA-6

Ziffer	FA-6
Beschreibung	Nutzer sollen Benachrichtigungen über Änderungen in einer Liste erhalten.
Zielsetzung	Wenn ein Nutzer Änderungen in einer Liste vornimmt, sollen optional die restlichen mitwirkenden Nutzer (falls vorhanden) eine Push-Benachrichtigung erhalten können.
Priorität	Mittel

2.6.7 Anforderung FA-7

Ziffer	FA-7
Beschreibung	Nutzer sollen Rezepte/Zutatenlisten verwalten können.
Zielsetzung	Es soll eine Datenbank an Rezepten/Zutatenlisten bereitstehen, in denen Zutaten als Artikel eingebunden sind. Diese sollen so mit wenig Aufwand komplett oder teilweise auf eine Einkaufsliste gesetzt werden können.
Priorität	Niedrig

3 Konzept der Anwendung

Bevor die tatsächliche Programmierung von Listipede stattfindet, sind einige theoretische Vorbereitungsschritte nötig, deren Hintergedanken und Hergang in diesem Kapitel erläutert werden.

Im Laufe der Konzeption und späteren Umsetzung werden viele Komponenten der App iterativ verändert oder gänzlich verworfen. Diese für das Endprodukt wenig relevanten Erarbeitungsschritte werden im Interesse der Lesefreundlichkeit und -effizienz nicht explizit dargelegt, wenn sie nicht für die Forschungsfrage als relevant erachtet werden, indem sie ein Problem oder einen Umstand der Multiplattform-Entwicklung mit Flutter und Dart aufzeigen. Dasselbe gilt für die Dokumentation der Implementierung in **Kapitel 4**.

Außerdem wird darauf hingewiesen, dass die englische Sprache bei der Namensgebung von Klassen, Objekten, Methoden etc. bewusst verwendet wird, da die Syntax der meisten Programmiersprachen und Frameworks – einschließlich Dart und Flutter – auf Englisch definiert ist. Dies wird bereits bei den Vorbereitungsschritten berücksichtigt und spiegelt sich unter anderem in entsprechenden Diagrammen wider.

3.1 Einkaufslisten-Funktionen

Um die grundlegende Funktion der App als nutzbare Einkaufsliste zu gewährleisten und somit **Anforderung FA-1** zu erfüllen muss zunächst definiert werden, mit welchen logischen Komponenten/Kategorien die Anwendung arbeiten soll. Die folgenden Kategorien werden hierfür als notwendig erachtet:

- Nutzer (USER)
- Einkaufsliste (SHOPPING_LIST)
- Produkt/Artikel (PRODUCT)
- Listeneintrag (LIST_ENTRY)

Diese logischen Kategorien sollen auf bestimmte Arten miteinander interagieren können, um die Einkaufslisten-Funktionalität der App zu gewährleisten. Zur einfachen Darstellung aller dieser Interaktionen wird ein vorläufiges Entity-Relation-Diagramm erstellt (**Abbildung 9**).

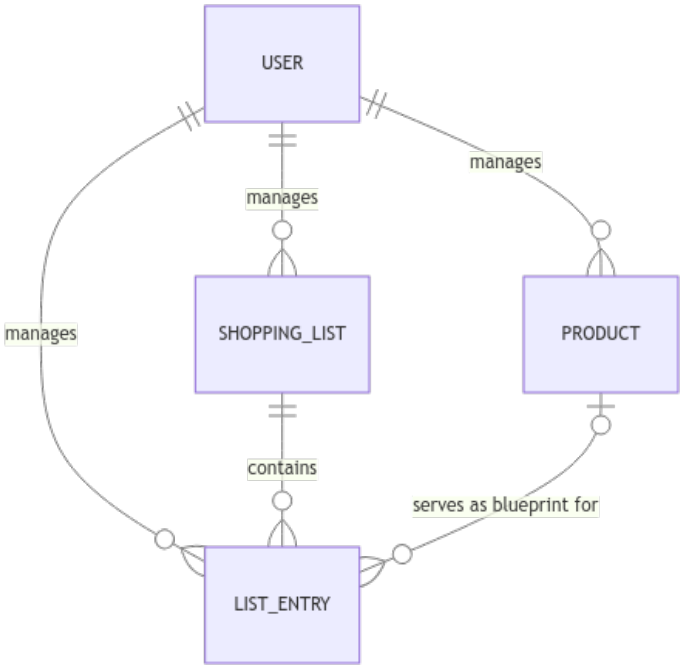

Abbildung 9: Entity-Relation-Diagramm der Einkaufslisten-Funktionen

Um die **Anforderung FA-2** zu erfüllen, müssen Nutzer Accounts erstellen können, die sie eindeutig identifizieren. Dies soll in Listipede durch die „Authentication“ Komponente von Googles Firebase gelöst werden.

Auch die anderen nötigen Datenbanken für die Anwendung sollen mit Firebase realisiert werden: Firestore soll für Daten der Nutzer in der App genutzt werden. Um sicherzustellen, dass der Zugriff auf Einkaufslisten und ihre Einträge von mehreren Nutzern gleichzeitig in Echtzeit erfolgt, sollen jene Daten in der Echtzeit-Datenbank von Firebase gespeichert werden. Die Entscheidung für Firebase als Lösung benannter Systeme erfolgt, weil die Nutzung mit Flutter und Dart als optimiert angenommen wird, da diese Komponenten alle von Google stammen. Zudem wird die Arbeitsentlastung, die Firebase mit seinen vorgefertigten Komponenten bietet, aus Entwicklersicht als sehr positiv empfunden. Auch wenn Firebase kein Bestandteil von Flutter ist, liegt seine Nutzung in Flutter-Apps nahe und wird deswegen an dieser Stelle als positive Möglichkeit in der Multiplattform-Entwicklung mit Flutter und Dart vermerkt.

Weiterhin sieht **FA-2** vor, dass mehrere Nutzer auf eine Einkaufsliste zugreifen können. Mittels der E-Mail-Adresse eines Nutzers soll man jenen Nutzer zu einer Einkaufsliste hinzufügen können. Es wird ein Ablaufdiagramm erstellt, das den Prozess darstellt (**Abbildung 10**). Die Nutzer einer Einkaufsliste sollen in Firestore hinterlegt werden.

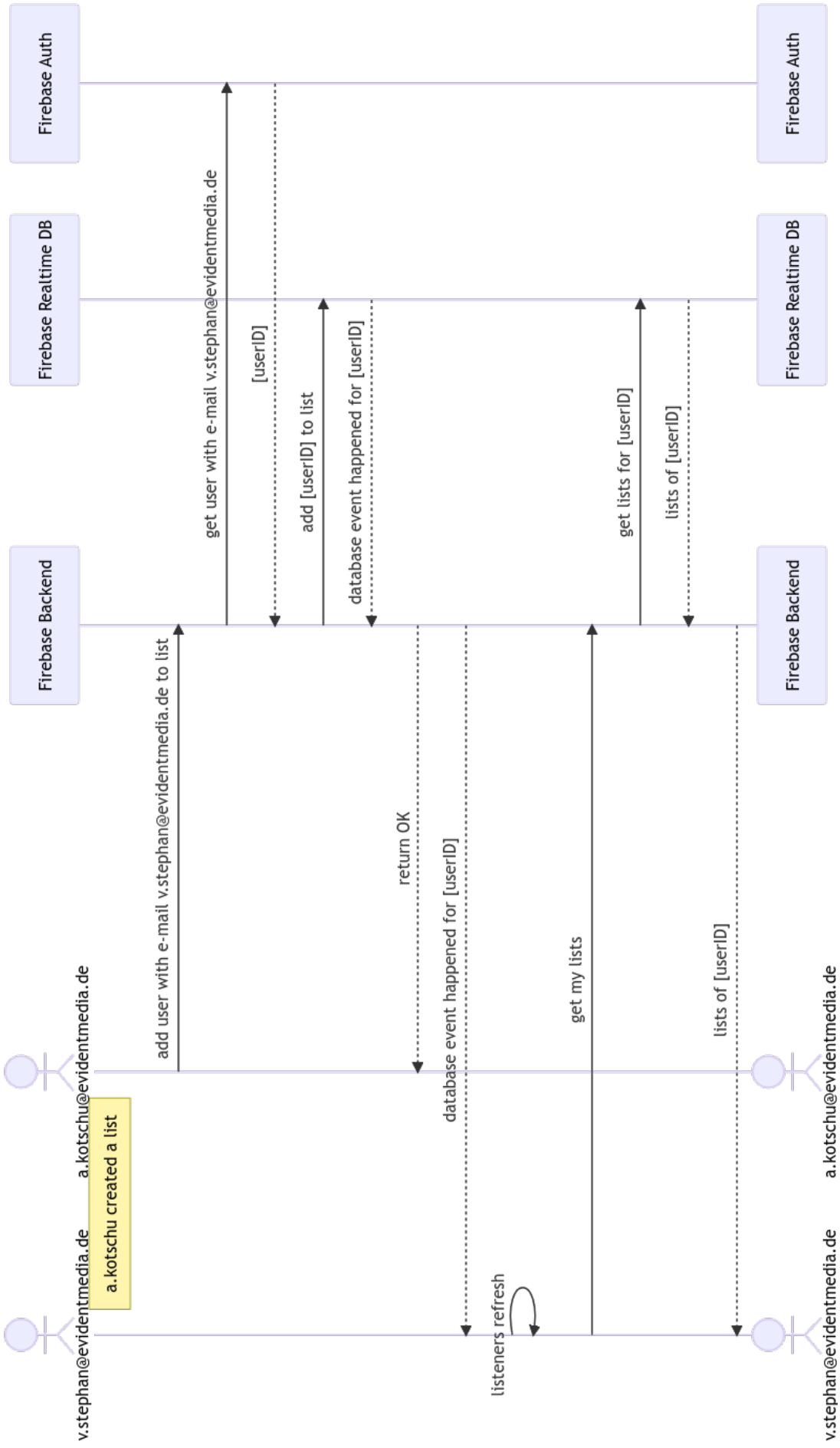

Abbildung 10: Ablaufdiagramm der Nutzerverwaltung bei Einkaufslisten

3.2 Automatisches Sortiersystem

Die Anwendung soll Einträge in einer Liste anhand der Reihenfolge sortieren, in der die mit den Einträgen verknüpften Produkte in vergangenen Einkäufen angetroffen wurden (**Anforderung FA-3**). Zu diesem Zweck müssen Produkte eine Art Sortier-Index (im Folgenden als SI abgekürzt) hinterlegt haben, anhand derer sie sortiert werden. Um der **Anforderung FA-4** gerecht zu werden, muss dieser SI eines Artikels gleichzeitig an einen Markt gebunden sein, da ein Artikel in mehreren Märkten vorhanden sein kann und die Sortier-Positionierung für separate Märkte gesondert berücksichtigt sein soll. Folglich stellt dieser SI die vermutete Position eines Artikels in der Route des Nutzers durch einen bestimmten Markt dar. Anders formuliert hinterlegt ein SI die Sortierposition einer einzigen Artikel/Markt-Kombination.

Um die Reihenfolge gekaufter Artikel nachvollziehen zu können wird davon ausgegangen, dass das Streichen eines Eintrags in der Regel bedeutet, dass jener Artikel aus dem Ladenregal entnommen wird. Diese Regel trifft nicht zu, wenn der Nutzer einen Artikel vergisst und erst später im Einkauf von der Liste streicht; hier würde das Abhaken des Artikels nicht mit seiner Position in der Route des Nutzers übereinstimmen. Ebenso kann ein Nutzer vergessen, einen entnommenen Artikel Dieser Faktor muss mit einbezogen werden, damit das Abhaken eines Artikels wirklich mit der Position dieses Artikels in der Route des Nutzers durch das Geschäft gleichzusetzen ist.

Aus diesen Umständen ergeben sich die ersten zwei Voraussetzungen, die bei der Konzeption der Sortierfunktion beachtet werden müssen: Erstens muss der Anfang und das Ende eines Einkaufsvorgangs zuverlässig erfasst werden. Das ist nötig, damit in der Sortierlogik die Reihenfolge der Artikel in einem Einkauf nachvollzogen werden kann. Zweitens muss gewährleistet werden, dass die Reihenfolge abgehakter Listeneinträge manuell vom Nutzer angepasst werden kann, damit sichergestellt wird, dass die erfasste Reihenfolge der abgehakten Produkte in der Anwendung tatsächlich mit der Reihenfolge jener Produkte in der Route des Nutzers durch den Laden übereinstimmt.

3.2.1 Einkaufsmodus

Zur Erfüllung der ersten Voraussetzung kann ein Einkauf als eine Art binärer Status der Liste für einen Nutzer verstanden werden: Ein Nutzer kauft zu einem gegebenen Zeitpunkt entweder mit einer Einkaufsliste ein oder nicht. Die Anwendung könnte also eine Art „Einkaufsmodus“ besitzen. Anfang und Ende eines Einkaufs wären somit als Beginn und Beendigung des Einkaufsmodus erfasst.

Eine solche Denkweise erweist sich auch für die Erfüllung der zweiten Voraussetzung als sinnvoll: Um die Reihenfolge der Produkte während eines Einkaufs ändern zu können, liegt es nahe, während des Einkaufsmodus den Platz eines abgehakten Produkts in der Reihenfolge aller abgehakten Produkte in der Einkaufsliste festzuhalten. Aus so einem Abhak-Index (Im Folgenden AI) der Artikel

könnte am Ende eines Einkaufs eine Anpassung des SI vorgenommen werden. Somit würde sich der SI anhand vergangener Einkäufe ändern und **Anforderung FA-3** wäre erfüllt.

Sowohl Beginn als auch Beendigung des Einkaufsmodus könnten auf verschiedene Arten erfolgen. Denkbar wäre eine zeitliche Eingrenzung, die den Einkaufsmodus beginnt, sobald ein Listeneintrag abgehakt wird, und beendet, sobald in einem vorgegebenen Zeitraum kein Eintrag abgehakt wurde. Es wird sich dafür entschieden, dass das Abhaken eines Artikels den Einkaufsmodus startet, aber das Beenden des Einkaufs aktiv vom Nutzer mit einem Button veranlasst werden soll, damit ein Einkauf nicht aus Versehen enden kann. Für diesen Prozess wird ein Ablaufdiagramm angefertigt (**Abbildung 11**).

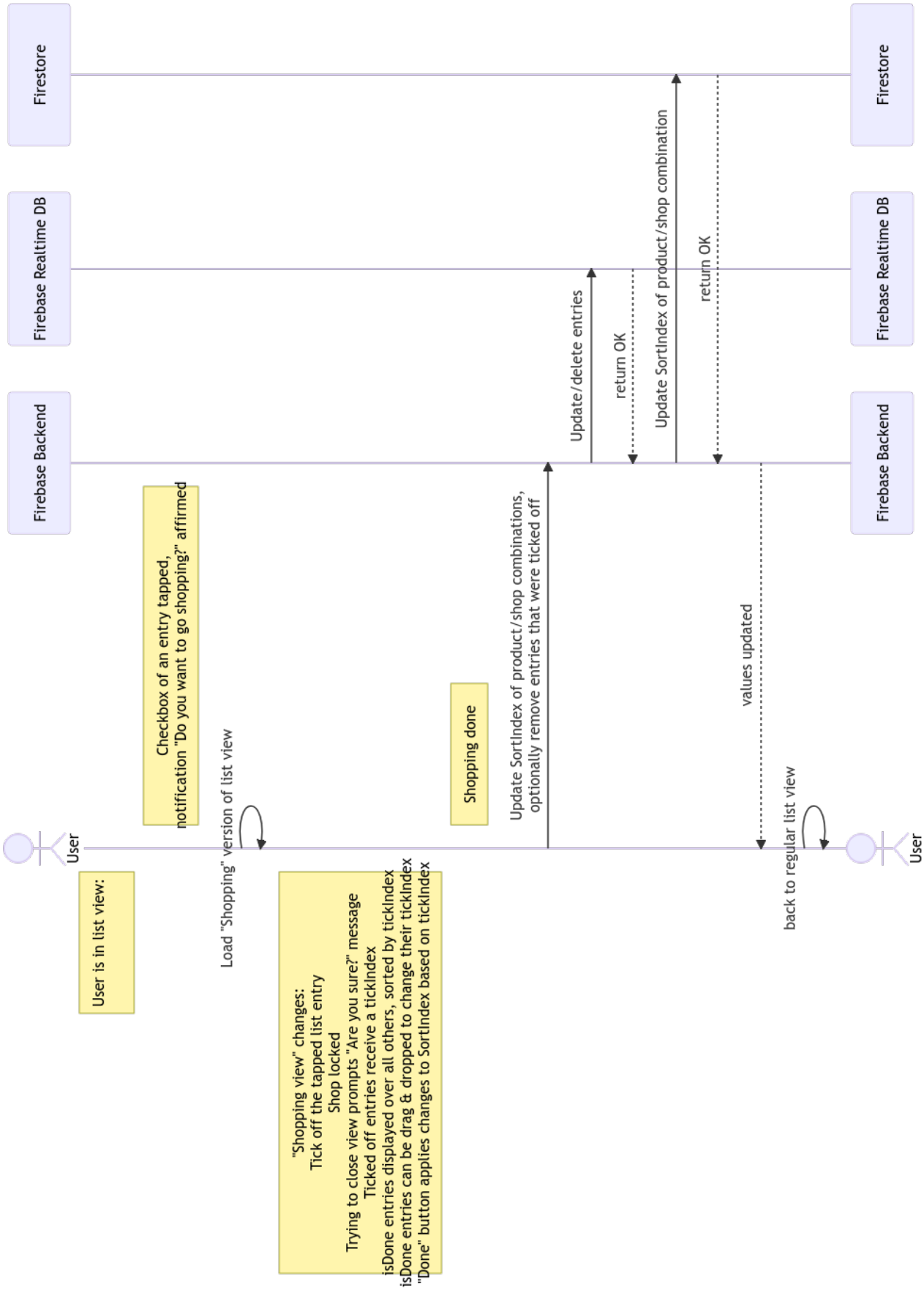

Abbildung 11: Ablaufdiagramm des Einkaufsmodus

3.2.2 Sortierlogik (Ansatz A)

Um dieses Modell umsetzen zu können, bedarf es einer Konkretisierung der Logik, die anhand des AI eines Artikels nach einem Einkauf den SI des Artikels anpassen soll. Zu diesem Zweck wird der Entwickler der Anwendung „pon - smarte Einkaufsliste“, Adrian Kühlewind, kontaktiert, da seine App dieselbe Art von smarter Listensortierung verspricht. In der E-Mail-Korrespondenz (**Anhang B**) erklärt er grob, wie die Sortier-Berechnung bei seiner Anwendung funktioniert:

> *„Bei pon haben alle Artikel einen [Sortier-]Index und keiner ist doppelt vergeben. […] [W]enn man einen Artikel verschiebt wird dieser zwischen den vorherigen und nachfolgenden gesetzt.“*

Als Beispiel hierfür werden zwei hypothetische Artikel herangezogen, welche SI 0 und 1 haben. Wenn nun ein dritter Artikel anhand seiner Sortierung zwischen diese beiden Artikel rutschen würde, würde er den SI mittig zwischen 0 und 1 erhalten, also 0,5. Dieser Prozess kann laut Kühlewind beliebig oft wiederholt werden. Zusammenfassend:

> *„Beim Lernen werden Indexe miteinander verglichen und ähnlich dem [genannten Beispiel] neue Positionen ermittelt.“*

Näher möchte der „pon“-Entwickler die Vorgehensweise in seiner App nicht preisgeben. Deshalb ist nicht nachvollziehbar, wie „pon“ damit umgeht, wenn ein Artikel nicht zwischen zwei andere Artikel „geschoben“ wird, sondern am Anfang oder Ende der Liste steht, oder ob neue SIs für eingekaufte Artikel wie bei Listipede geplant am Schluss einer Art Einkaufsmodus berechnet werden. Es ist denkbar, dass die Sortierlogik in „pon - smarte Einkaufsliste“ Änderungen direkt bei Abhaken eines Artikels vornimmt. Ferner ist nicht ersichtlich, zu welchem Zeitpunkt und wie genau einem Artikel das erste Mal ein SI zugewiesen wird.

Nichtsdestotrotz bildet das beschriebene Konzept, Artikeln einzigartige SIs zuzuweisen und diese je nach Platzierung des Produkts beim Einkauf „mittig“ zwischen die SIs anderer Artikel zu setzen, die Vorlage für unseren ersten Lösungsansatz der Sortierung. – im Folgenden „Ansatz A“. Ansatz A wird für die mögliche Verwendung in der App Listipede folgendermaßen konkretisiert, um in einem späteren Schritt getestet und verglichen zu werden:

Unter Ansatz A sollen SIs nur verändert werden, wenn sie für einen Einkauf eine falsche Reihenfolge ergeben. Bei Beendigung eines Einkaufs wird die Stimmigkeit des SI von jedem abgehakten Produkt nacheinander in der Reihenfolge der AIs überprüft, indem er mit dem SI des darauffolgenden abgehakten Produkts verglichen wird. Wenn der SI des darauffolgenden Produkts größer ist, ist er stimmig mit der Reihenfolge der AIs und es erfolgt keine Anpassung des SI. Falls der SI des nächsten abgehakten Produkts kleiner ist, wird der Wert des SI des überprüften Produkts „mittig“ zwischen die SIs des vorigen und nächsten

abgehakten Produkts gesetzt. Anders formuliert bedeutet das für ein zu überprüfendes Produkt p, das laut AI zwischen den Produkten o und q gekauft wurde:

$$SortIndex(p) = \frac{SortIndex(o) + SortIndex(q)}{2}$$

Wird ein SI in dieser Art angepasst, beginnt die Überprüfung der SIs im Einkauf erneut von vorne. Dieser iterative Prozess wird so lange wiederholt, bis in einem Durchlauf keine Anpassungen mehr vorgenommen werden. Im Sonderfall, dass o nicht existiert, weil p im Einkaufsvorgang als erstes abgehakt wurde, wird anstatt $SortIndex(o)$ der Wert 0 eingesetzt.

Der erste SI eines Produkts wird vergeben, sobald jenes Produkt das erste Mal gekauft wird. Falls mehrere abgehakte Produkte hintereinander keinen hinterlegten SI haben, werden diese analog zur obigen Rechnung gleichmäßig zwischen die Produkte mit vorhandenen SIs verteilt.

Falls keine Produkte eines Einkaufs einen bestehenden SI haben, wird der allerhöchste SI eines Produkts aus der Datenbank erfasst und die nächsthöhere natürliche Zahl als SI für das erste abgehakte Produkt des Einkaufs zugewiesen. Der SI jedes folgenden abgehakten Produkts wird um 1 erhöht. Wenn keine Produkte einen SI hinterlegt haben, wird bei der Zahl 1 begonnen. Wenn ein oder mehrere Produkte am Ende der Abhak-Reihenfolge keinen hinterlegten SI haben, werden für diese ebenso ganzzahlige Werte vergeben.

Es muss gewährleistet sein, dass der SI eines Produkts einzigartig ist; keine zwei Produkte dürfen denselben SI gleichzeitig haben. Zu diesem Zweck wird am Anschluss an den erläuterten Prozess überprüft, ob die berechneten SIs eines Einkaufs bereits in der Datenbank vorhanden sind und in jenem Fall der neu zu vergebende SI zwischen den berechneten Wert und den unmittelbar vorigen SI in der Datenbank gesetzt.

3.2.3 Sortierlogik (Ansatz B)

Ein zweiter Lösungsansatz wird separat eigenständig erarbeitet. Hierbei wird im Gegensatz zu Ansatz A die Spanne der SIs begrenzt. SIs müssen Teil einer Menge M sein, die folgendermaßen definiert wird:

$$M = [a, b]$$

wobei a und b theoretisch willkürlich wählbar sind. Folglich gilt für jeden beliebigen SI s aus M:

$$\{s | a \leq s \leq b\}$$

Diese Begrenzung des SI bringt eine andere Denkweise zum Ausdruck als Ansatz A: Der SI soll hier die vermutete Position eines Artikels widerspiegeln, die dieser in Relation zum Anfang (repräsentiert durch a) und zum Ende (repräsentiert durch b) der Einkaufsroute des Nutzers einnimmt. Im Gegensatz zu Ansatz A

wird die genaue Reihenfolge der abgehakten Listeneinträge für eine Anpassung des SI nicht beachtet.

Eine Anpassung des SI von abgehakten Produkten nach Beendigung eines Einkaufs soll prinzipiell stattfinden. Die Berechnung beginnt damit, dass die AIs umgerechnet werden, sodass sie gleichmäßig zwischen a und b verteilt sind, wobei der AI des ersten eingekauften Produkts immer den Wert a hat, während das letzte eingekaufte Produkt den Wert b erhält. Als nächstes wird die Differenz zwischen vorhandenem SI und AI des Produkts erfasst. Diese Differenz wird wiederum in eine Funktion $f(x)$ eingesetzt, deren Aufgabe es ist, den Wert zu berechnen, der mit Addition des bisherigen SI den neuen SI des Produkts ergibt.

Zu diesem Zweck soll $f(x)$ relativ kleine Änderungen bewirken, insbesondere wenn die Differenz zwischen AI und vorhandenem SI vergleichsweise trivial ist. Hintergedanke ist, Änderungen des SI in kleinen Schritten über mehrere Einkäufe iterativ vorzunehmen, anstatt wie bei Ansatz A bei Unstimmigkeiten in der Reihenfolge direkt neue SIs zu vergeben, die den bisherigen SI ignorieren. Wenn die Differenz jedoch sehr groß ist, weil sich die angenommene Position des Produkts sehr von der tatsächlichen Position im Einkauf unterscheidet, soll der SI aggressiver angepasst werden, ohne den Wert der Differenz zu überschreiten. Falls ein Produkt keinen hinterlegten SI hat, wird der Positions-Index unverändert als initialer SI des Produkts eingetragen.

Um die Formulierung von $f(x)$ unter Beachtung dieser Vorgaben zu erleichtern, werden an dieser Stelle für a und b konkret die Zahlen 0 und 1 gewählt. Dadurch kann $f(x)$ folgendermaßen definiert werden:

$$f(x) = \frac{|x|}{x} \cdot y \cdot x^2$$

wobei die Variable y einen Wert zwischen 0 und 1 repräsentiert. **Abbildung 10** zeigt den Graphen von $f(x)$ für $y = 0{,}7$.

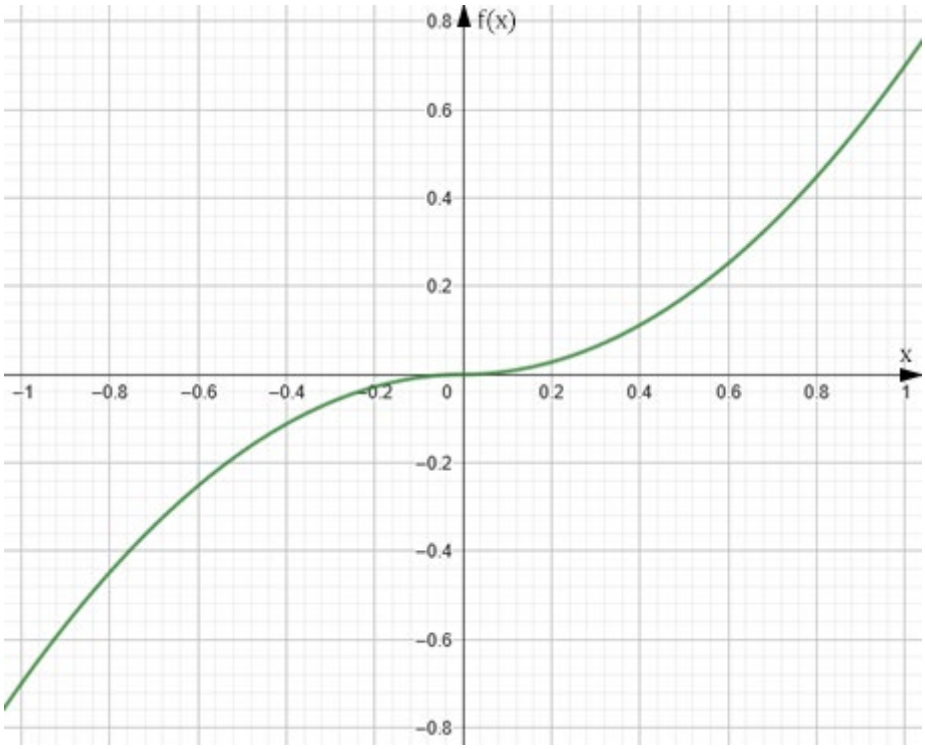

Abbildung 12: f(x) mit y=0,7

Anhand des Funktionsgraphen ist erkennbar, dass das Ergebnis von $f(x)$ das gewünschte Verhalten aufweist: Kleine Differenzen zwischen AI und SI führen zu unwesentlichen Veränderungen des SI, während mit größerem Wert x vergleichsweise größere Änderungen stattfinden. Gleichzeitig wird ein SI maximal um den Wert y verändert (bei dem größtmöglichen Unterschied zwischen AI und vorigem SI, also für $x = 1$).

Ansatz B bietet einige Möglichkeiten zur Verbesserung, die an dieser Stelle anerkannt werden. Die wohl offensichtlichste Möglichkeit stellt die Funktion $f(x)$ und insbesondere der tatsächliche Wert von y dar, welcher diktiert, wie aggressiv Änderungen am SI vorgenommen werden. Es ist unklar, welcher Wert ideal ist oder wie „ideal“ in diesem Kontext zu definieren ist, da in der Theorie jeder Wert $y > 0$ nach genug Einkäufen den SI eines Produkts an die tatsächliche Position des Produkts in der Route des Nutzers anpasst und gleichzeitig sichergestellt werden soll, dass Änderungen am SI nicht zu groß sind. Die Funktion $f(x)$ an sich könnte teilweise oder gänzlich anders gewählt werden.

Ferner wäre denkbar, bestimmte Werte als feste Schwellen zu definieren, indem alle Ergebnisse von $f(x)$ unter einem gewissen Wert ignoriert werden, damit kleine Werte x nicht nur unbeachtliche Änderungen am SI veranlassen, sondern der SI in diesen Fällen gänzlich unberührt bleibt. Dasselbe gilt für sehr hohe Werte x: Eine feste Obergrenze könnte nützlich sein.

Diese Anpassungsmöglichkeiten sind als Stellschrauben zur Optimierung des Grundprinzips von Ansatz B zu verstehen. Es wird als unwahrscheinlich bewertet, dass diese Faktoren das Grundprinzip von Ansatz B nennenswert verändern.

Folglich werden diese Möglichkeiten im weiteren Verlauf dieser Arbeit nicht ausgiebig berücksichtigt, aber als positives Potenzial von Ansatz B festgehalten.

3.2.4 Test der Ansätze

Um die Vorgehensweise der Ansätze A und B realitätsnah nachzuvollziehen und mögliche Probleme der Sortierlogik aufzudecken, wird ein Beispiel in Form eines fiktiven Geschäfts mit 12 Produkten herangezogen. Für diese Produkte wird eine Reihenfolge festgesetzt, die die Reihenfolge in der Route eines Nutzers durch das Geschäft darstellt. Gleichzeitig erhalten diese Produkte zufällige SIs, die eine zu verbessernde Ausgangssituation der Sortierung darstellen. Der resultierende Produktkatalog ist in **Tabelle 2** abgebildet.

Produkt	**Platzierung in Route des Nutzers**	**Ausgangs-SortIndex**
Kartoffel	1	1
Apfel	2	2
Semmel	3	4
Zahnseide	4	12
Müllbeutel	5	11
Milch	6	7
Hackfleisch	7	6
Kidneybohnen	8	9
Gewürzgurken	9	10
Schokolade	10	5
Makkaroni	11	8
Instant-Kaffee	12	3

Tabelle 2: Produktkatalog des fiktiven Geschäfts

Aus diesem Produktkatalog werden drei Einkäufe zusammengestellt, die nacheinander stattfinden (**Tabelle 3**).

Einkauf 1	**Einkauf 2**	**Einkauf 3**
Milch	Kartoffel	Apfel
Kidneybohnen	Apfel	Semmel
Gewürzgurken	Semmel	Müllbeutel
Schokolade	Müllbeutel	Milch
Instant-Kaffee	Hackfleisch	Makkaroni
	Schokolade	Instant-Kaffee

Tabelle 3: Simulierte Einkäufe in gekaufter Reihenfolge

Die SIs der Produkte nach den Einkäufen werden separat mit den Ansätzen A und B manuell kalkuliert. Zusätzlich wird ein Durchgang mit Ansatz A ohne Ausgangs-SIs berechnet, um denselben Prozess ohne bestehende Daten nachzuvollziehen. Einige aus diesen Berechnungen gewonnene Erkenntnisse (etwa das Handhaben einer Anpassung des ersten SI in einem Einkauf) sind in der Formu-

lierung und genauen Definition der Sortier-Ansätze bereits benannt; andere werden im Kontext der jeweiligen Berechnung dargelegt. Die komplette Dokumentation dieser Berechnungen ist **Anhang A** zu entnehmen.

Die Durchführung von Ansatz A erfolgt mit einer bestimmten Anzahl an Durchläufen der in **Kapitel 3.2.2** dargelegten Berechnung, wobei der letzte Durchlauf keine Anpassung der SIs vornimmt. **Tabelle 4** zeigt die benötigte Anzahl an Durchläufen für die Berechnungen.

Einkauf	Anzahl Durchläufe		
	mit Ausgangs-SI	ohne Ausgangs-SI (Variante 1)	ohne Ausgangs-SI (Variante 2)
#1	14	1	1
#2	15	42	1
#3	9	12	12

Tabelle 4: Anzahl der benötigten Durchläufe bei Ansatz A

Die Berechnung der Einkäufe mit dem Ausgangs-SI verläuft ohne nennenswerte Probleme. Bei der Durchführung ohne initialen SI tritt ein zuvor nicht kalkulierter Fall auf: Da in Einkauf 2 zufällig nur das letzte Produkt (Schokolade) einen hinterlegten SI hat, können die SIs nicht zwischen Produkten verteilt werden. Als erste Variante zur Lösung eines solchen Falls werden die SIs nach dem höchsten SI in der Produkt-Datenbank ganzzahlig weitergeführt. Diese Vorgehensweise erweist sich als sehr ungünstig, da dadurch für die Berechnung von Einkauf 2 42 Durchläufe benötigt werden.

Als zweite Variante dieser Berechnung werden die fehlenden SIs in der Einkaufsliste zwischen den niedrigsten SI in der Datenbank und den SI des letzten Produkts in der Einkaufsliste verteilt. Durch dieses Vorgehen ist keine Anpassung in Einkauf 2 nötig.

Stellenweise tritt bei den Berechnungen das Problem auf, dass einer oder mehrere SIs eines berechneten Einkaufs bereits als SI eines anderen Produkts im Katalog vertreten sind. Da dies in Konflikt mit der Definition von Ansatz A ist, werden in diesen Fällen die neuen SIs zwischen den eigentlich berechneten Index und den vorigen SI im Produktkatalog gesetzt.

Mehrfach vorkommende SIs stellen auch während der Berechnung ein Problem dar. Bei Variante 2 führt ein solcher Fall in Einkauf #3 dazu, dass die Berechnung nie abschließt, weil die Anpassung der SIs zwischen den beiden gleichen Werten (Apfel und Milch) eine Endlosschleife ergibt. **Tabelle 5** zeigt den Anfang dieser Berechnung und ist von links nach rechts zu lesen: Jede Spalte neben den Produkten stellt einen Durchlauf der Sortierlogik dar, der von oben nach unten abläuft. Gelbe Felder markieren die SIs, die von der Sortierlogik als Diskrepanz erkannt und angepasst werden. Die in dieser Weise angepassten SIs sind grün hinterlegt.

Produkt	Sort-Index										
Apfel	2.25	2.25	2.25	1	1	1	1	1	1	1	
Semmel	2.5	2.5	2	2	1.375	1.375	1.094	1.094	1.023	1.023	
Müllbeutel	3.25	1.75	1.75	1.75	1.75	1.188	1.188	1.047	1.047	1.012	...
Milch	1	1	1	1	1	1	1	1	1	1	
Makkaroni	3	3	3	3	3	3	3	3	3	3	
Instant-Kaffee	5	5	5	5	5	5	5	5	5	5	

Tabelle 5: Beginn der endlosen Berechnung von Einkauf 3 mit Variante 2 (Werte auf 3 Nachkommastellen gerundet).

Auch hier ist es nötig, eine Anpassung an der Logik vorzunehmen, um den Fehler zu vermeiden: Nach jeder Anpassung eines SI oder bevor ein Durchlauf der Sortierlogik startet, muss geprüft werden, ob der berechnete angepasste SI schon in der Einkaufsliste vertreten ist. In jenem Fall muss ein anderer SI vergeben werden. Es wird sich dafür entschieden, den angepassten SI zwischen das Produkt mit dem berechneten SI und das vorige Produkt im Produktkatalog zu setzen.

Die Berechnung der Einkäufe mit Ansatz B wird nach dem in **Kapitel 3.2.3** beschriebenen Vorgehen ausgeführt. Der Wert y in der Funktion $f(x)$ wird zu diesem Zweck als 0,7 festgelegt. Hierbei gibt es keine Auffälligkeiten.

3.2.5 Ergebnis

Die Berechnungen mit Ansatz A zeigen einige Fälle auf, für die spezielle Ausnahmeregelungen und Mechanismen zur Erkennung gefunden werden müssen. Es wird davon ausgegangen, dass zusätzliche Schritte dieser Art in der Sortierlogik mit zusätzlicher nötiger Rechenleistung/-dauer verbunden sind.

Außerdem wird aus den Berechnungen ersichtlich, dass Ansatz A unter bestimmten Umständen eine hohe Anzahl an Iterationen der Berechnungslogik benötigt. Auch wenn die markant hohe Anzahl an Durchläufen bei Einkauf 2 mit Variante 1 der Berechnung ohne Ausgangs-SI dadurch zu begründen ist, dass die Initial-SIs des Einkaufs ungünstig gewählt wurden, sind sie sehr aufschlussreich über das Verhalten von Ansatz A unter ungünstigen Bedingungen generell. Eine hohe Anzahl an nötigen Berechnungsdurchläufen wird ebenso wie die bereits dargelegten nötigen Ausnahmeregelungen im Rahmen der Effizienz negativ bewertet. Variante 2 dieser Berechnungen führt zu keiner nötigen Anpassung der SIs in Einkauf 2; jedoch ist markant, dass diese „Verteilung“ der SIs bis Anfang des Katalogs in gewisser Weise Ansatz B ähnelt.

Die Berechnung mit Ansatz B hingegen verläuft wie erwartet. Im Gegensatz zu Ansatz A gibt es rein prinzipiell bei Ansatz B kein Potential für lange/unendliche Verkettungen an Rechnungsdurchläufen. Ausnahmeregelungen in Ansatz B sind in Form der in **Kapitel 3.2.3** genannten Stellschrauben denkbar, aber nicht notwendig wie in Ansatz A. Aus diesen Gründen wird sich für Ansatz B entschieden.

3.3 Datenbank

Zur Skizzierung der Datenbankstruktur wird auf Basis des vorläufigen Entity-Relation-Diagramm aus **Kapitel 3.1** (**Abbildung 9**) das endgültige Entity-Relation-Diagramm (**Abbildung 10**) erstellt, das alle bisher erarbeiteten Kategorie-Komponenten der Datenbanken und ihre Beziehungen zueinander abbildet. Diese sollen dadurch zusammenfassend visualisiert werden, um die Implementierung zu erleichtern.

Im Diagramm werden darüber hinaus die voraussichtlichen Eigenschaften notiert, die ein Objekt in der jeweiligen Kategorie in der Datenbank besitzen soll. Die Eigenschaften sind unter den Objekten tabellarisch in folgendem Format vermerkt:

Typ (String, integer, boolean, …)	Bezeichnung	Art des Schlüssels (falls zutreffend): PK (Primary Key) = Primärschlüssel FK (Foreign Key) = Fremdschlüssel UK (Unique Key) = Eindeutiger Schlüssel PK/FK = Primärschlüssel aus zwei Fremdschlüsseln	Notizen (falls vorhanden)

Auch wenn das vorliegende Entity-Relation-Modell Konventionen aus SQL-Datenbanken befolgt, wird ausdrücklich darauf hingewiesen, dass es sich hierbei um Vorüberlegungen für die logische Struktur der Daten handelt. Die tatsächlich in der Implementierung verwendeten Firebase-Datenbankstrukturen sind keine SQL-Datenbanken.

Zusätzlich zu den bereits in **Kapitel 3.1** definierten Kategorien USER (Nutzer), PRODUCT (Produkt/Artikel), SHOPPING_LIST (Einkaufsliste) und LIST_ENTRY (Listeneintrag) wird in diesem finalen Entity-Relation-Diagramm die in **Kapitel 3.2** erarbeitete Kategorie des SHOP (Laden) abgebildet. Des Weiteren ist eine Abstraktionsebene zwischen USER und SHOPPING_LIST eingesetzt: SHOPPING_LIST_USERS_XREF. Diese bildet die Nutzer einer Einkaufsliste ab, die laut **Kapitel 3.1** zusätzlich hinterlegt werden müssen. In ähnlicher Weise wird PRODUCT_SHOP_XREF dem Diagramm hinzugefügt, um die in **Kapitel 3.2** eingeführte logische Komponente einer Produkt/Markt-Kombination zu repräsentieren, die den SI beinhaltet.

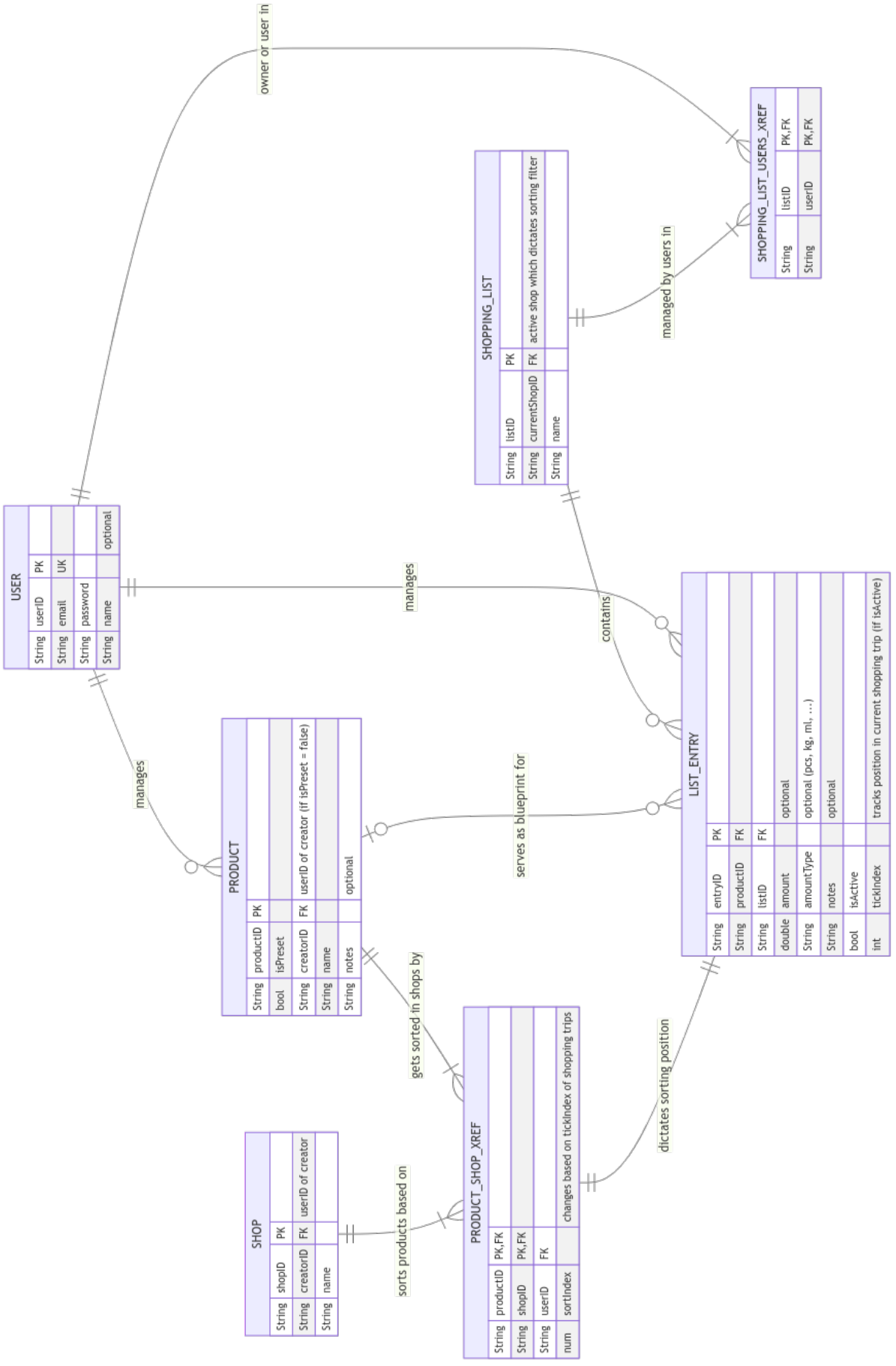

Abbildung 13: Entity-Relation-Diagramm

3.4 Design

Als erster Schritt für das Design der Anwendung muss entschieden werden, ob ein vorgefertigtes oder eigenes Gestaltungskonzept für die App verwendet wird. Diese Entscheidung ist insbesondere im Kontext der Crossplattform-Entwicklung interessant, da das Design der Nutzeroberfläche der **Anforderung FA-5** gerecht werden soll, indem das Design für die beiden Plattformen iOS und Android passend gewählt wird.

Zwar ist es technisch nicht notwendig, die Nutzeroberfläche an die Plattformen anzupassen, damit die Anwendung auf beiden Plattformen funktioniert, jedoch wird es für die Forschungsfrage durchaus als Mehrwert für die Multiplattform-App-Entwicklung angesehen, wenn die Nutzererfahrung gut an verschiedene Plattformen angepasst werden kann. „Gut“ bedeutet in diesem Kontext einerseits aus Sicht der Entwickler einfach und effizient umsetzbar, und andererseits aus Sicht der Nutzer performant und visuell nahtlos in die Plattform integriert.

Zudem stellt Flutter Bibliotheken bereit, die Komponenten aus Googles „Material Design“ für Android und Apples „Human Interface Guidelines“ für iOS enthalten. Eine Verwendung dieser Flutter-Bibliotheken liegt nahe, da der Fokus dieser Arbeit auf der Programmierung liegt. Die arbeitsaufwändige Entwicklung eines eigenen App-Designs hat für die Forschungsfrage wenig Nutzen, während die Benutzung der benannten Bibliotheken für die beiden Plattformen es ermöglicht, die Chancen und Hürden beim Aspekt App-Design zu untersuchen. Es werden folglich die von Flutter für Android und iOS bereitgestellten Designbibliotheken benutzt.

3.4.1 Mock-Ups

Um die bisher erarbeiteten grundlegenden Strukturen der Anwendung weiter zu konkretisieren und somit beim Programmieren zielgerichteter arbeiten zu können, sollen im Vorfeld grafische Mock-Ups für das Design der Nutzeroberfläche erstellt werden. Dies ist im Vorfeld nur für eine der beiden Designsprachen nötig, da es eher fundamental um Layout und Struktur der App geht als um die genauen Details einzelner Komponenten. Ebenso ist die Farbgebung in diesem Schritt noch willkürlich, da sie später beliebig gewählt werden kann.

Die Mock-Ups erfolgen in der Software Figma für Googles Material Design. Hierzu wird das von Google in Figma bereitgestellte „Material 3 Design Kit“ verwendet. Als erstes wird eine skizzierte Start-Ansicht bei Nutzung der App festgelegt (**Abbildung 14**). Ähnlich wie bei den zuvor untersuchten Einkaufslisten-Anwendungen sollen in dieser Ansicht die Einkaufslisten des Nutzers als größere Karten in einer Liste angezeigt werden. In der oberen Menüleiste können bis zu 2 etwaige sekundäre Menüs erreichbar sein (hier beispielsweise ein Hamburger-Menü und das Profil des Nutzers). Ein Antippen einer Einkaufsliste soll den Nutzer zu dieser Einkaufsliste navigieren.

Abbildung 15 zeigt jene geplante Ansicht einer Einkaufsliste. Einträge haben hier Platz für etwaige Notizen und Mengen. Besonders wichtig ist die Schaltfläche unter der oberen App-Leiste. Dort wird der Markt, nach dem die Liste sortiert wird, angezeigt.

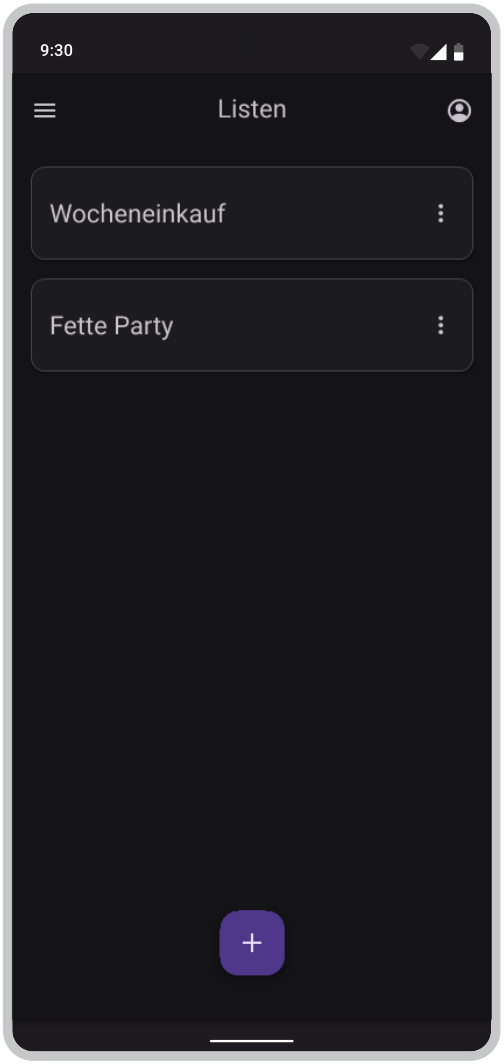

Abbildung 14: Mock-Up Startbildschirm

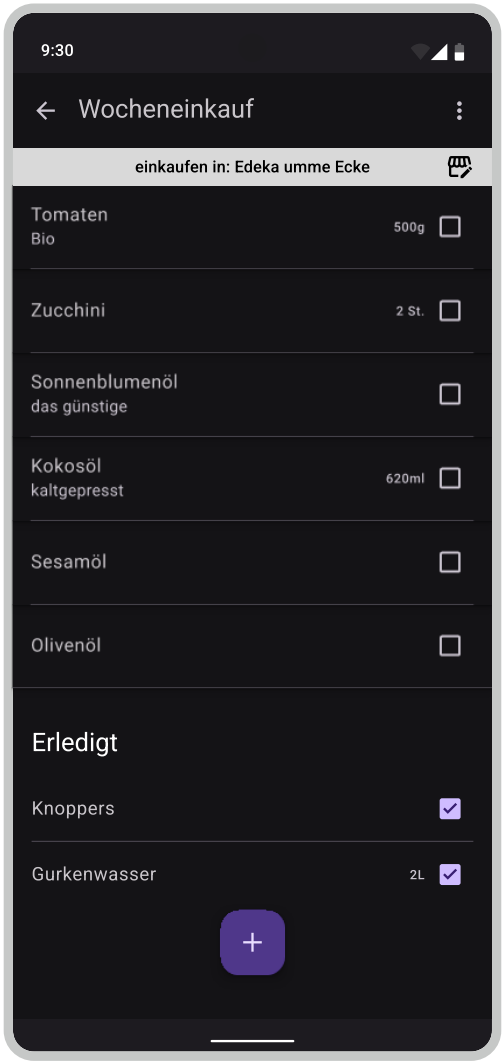

Abbildung 15: Mock-Up Einkaufslistenansicht

Ein Antippen dieses Laden-Felds soll den Nutzer zur Marktauswahl (**Abbildung 16**) führen. In dieser simplen Ansicht sollen die Läden als Liste dargestellt werden.

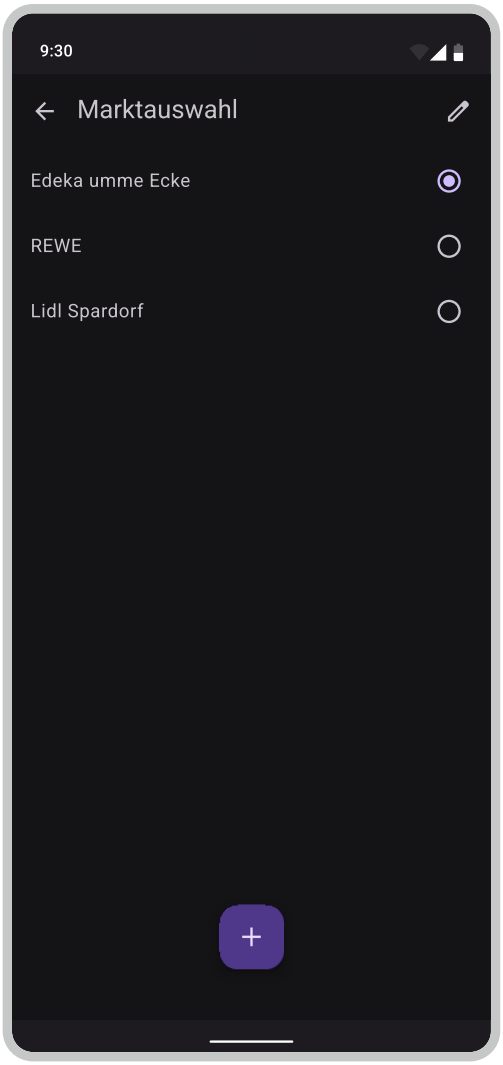

Abbildung 16: Mock-Up Ladenauswahl

4 Implementierung der Anwendung

Im folgenden Kapitel wird beschrieben, wie auf Basis der in **Kapitel 3** erarbeiteten Komponenten die Einkaufslisten-App Listipede realisiert wird. Hierfür werden die Schritte zur Programmierung der einzelnen Bestandteile erläutert. Erstmalig verwendete Konzepte und Schlüsselwörter werden ausführlicher dargelegt als solche, die bereits thematisiert wurden oder als selbsterklärend erachtet werden.

Während der gesamten Implementierung wird die Dokumentation von Flutter (`https://docs.flutter.dev`) als Hilfestellung verwendet. Die Dokumentation wird als Teil der Arbeit mit Flutter und Dart angesehen und folglich nach erfolgter Implementierung bewertet.

Um das Arbeiten mit Flutter zu ermöglichen, wird als erster Schritt der Implementierung die Entwicklungsumgebung initialisiert. Zu diesem Zweck wird das *Software Development Kit* (SDK) von Flutter bezogen. Danach wird Android Studio samt Android SDK installiert. Als nächstes wird in Android Studio ein virtuelles Android-Gerät erstellt, um die Anwendung während der Entwicklung in Android zu testen. Anschließend werden durch Ausführen von `flutter doctor -android-licenses` Android-Lizenzen angezeigt, für die das Einverständnis erteilt werden muss, ehe man das Android SDK nutzen kann. Zusätzlich wird auf einem separaten Apple-Gerät mit MacOS auf ähnliche Weise Flutter bezogen, um mit dem iOS Simulator in MacOS die Anwendung für iOS zu testen.

Für die Entwicklung wird *Visual Studio Code* (VS Code) als integrierte Entwicklungsumgebung (IDE) verwendet. VS Code bietet Erweiterungen zur Unterstützung von Dart und Flutter, die installiert werden.

Um das Flutter-Projekt zu initialisieren, wird in der Kommandozeile des IDE `flutter create listipede` ausgeführt. Dadurch entsteht das Stammverzeichnis des Projektes *listipede*, und darin einige Dateien und Ordner, darunter der Quellcode-Ordner *lib* mit der automatisch generierten *main.dart*: der Einstiegspunkt (oder das Hauptskript) einer Flutter-Anwendung; hier wird die Ausführung der App gestartet. Alle *.dart*-Dateien einer Flutter-Anwendung gehen von *lib* als Stammverzeichnis aus; somit wird für alle im Verlauf dieser Arbeit entstehenden *.dart*-Dateien und Unterordner vom wiederholten Nennen des Verzeichnisses abgesehen. Diese *.dart*-Dateien bilden den Quellcode einer Flutter-App. Die Architektur der Nutzeroberfläche von Flutter-Anwendungen setzt sich aus Widgets (Teilkomponenten) zusammen, die in diesen *.dart*-Dateien definiert und verwendet werden.

Eine wichtige generierte Datei im Projekt-Stammverzeichnis ist *pubspec.yaml:* Sie enthält Informationen über das Projekt (Name, Beschreibung) und die Pakete/Bibliotheken, die es benötigt (Abhängigkeiten/dependencies). Es ist wichtig, *pubspec.yaml* regelmäßig zu aktualisieren, um sicherzustellen, dass alle Abhän-

gigkeiten korrekt verwaltet werden. Außerdem werden Ordner für den Code der einzelnen Plattformen generiert.

Das Einrichten des Projekts in Firebase beginnt mit dem Anlegen eines neuen Projekts auf `https://console.firebase.google.com/`. Hier wird der Projektname *listipede* festgelegt. Als nächstes wird die Firebase-CLI über Node.js bezogen und mit dem Befehl `dart pub global activate flutterfire_cli` aktiviert. Im Stammverzeichnis des Flutter-Projekts wird `flutterfire configure --project=listipede` ausgeführt und die Plattformen Android und iOS zur Registrierung in Firebase ausgewählt. Gleichzeitig wird dadurch die Konfigurationsdatei *firebase_options.dart* generiert. Diese wird in ein neues Verzeichnis *config* verschoben.

Insgesamt verläuft die Initialisierung ohne größere Hürden. Die `doctor` Funktion von Flutter wird aus Entwicklersicht als hilfreich bewertet, weil sie das Arbeiten mit Flutter erleichtert, indem sie die Voraussetzungen zur Nutzung des Frameworks verständlich darstellt und Schritte zur Behebung von Fehlern kommuniziert.

4.1 Plattform-spezifischer Aufbau

Um das in **Kapitel 3.4** benannte Multiplattform-Design-Konzept zu realisieren, ist es nötig, die Struktur der Nutzeroberfläche von Grund auf darauf auszurichten, dass je nach Betriebssystem verschiedene Widgets angezeigt werden. Das in Dart enthaltene Paket `io` liefert die Möglichkeit, eine Kondition `Platform.isAndroid` bzw. `Platform.isIOS` in if-statements zu verwenden. Diese Herangehensweise erfordert die separate Definition der entsprechenden Widgets. Da sehr viele Widgets mit vergleichbarem Inhalt und Funktion aber unterschiedlicher visueller Aufmachung angezeigt werden sollen, ist abzusehen, dass eine solche Architektur zu einer hohen Menge an Code führt, der in ähnlicher Form doppelt vorhanden ist.

Eine weitere Möglichkeit für den plattformspezifischen Aufbau, die dieses Problem lösen soll, besteht in Form einer externen Bibliothek für Flutter. Solche Bibliotheken und Pakete können über `https://pub.dev` recherchiert werden, wo auch die Dokumentationen bereitstehen. Das tatsächliche Hinzufügen erfolgt mit dem Befehl `flutter pub add` gefolgt vom Namen des Paketes. Dieser Befehl fügt ein Paket den Abhängigkeiten in *pubspec.yaml* hinzu. Alternativ wäre es möglich, in jener Datei den Namen eines Paketes manuell im Format `paketname: ^version` unter `dependencies:` einzugeben und danach den Befehl `flutter pub get` auszuführen, mit dem Flutter die Abhängigkeiten überprüft und fehlende Pakete bezieht.

Das erwähnte Paket hat den Namen *flutter_platform_widgets*. Es definiert eine Reihe an Widgets, die die Code-Verdoppelung minimieren sollen, indem sie unter einmaliger Definition der Argumente eines Widgets die Anzeige plattform-

spezifischer Widgets veranlassen. Vergleichbare Argumente der Material- und Cupertino-Versionen bestimmter Widgets (z.B. Text in einem Button, Inhalt eines Screen-Grundgerüsts, usw.) können nach diesem Prinzip einmalig angegeben werden. Dort, wo es nötig ist, können Abweichungen zwischen den Widgets der beiden Plattformen oder Material-/Cupertino-eigene Argumente in einem gesonderten Argument hinterlegt sein.

Das Paket *flutter_platform_widgets* wird bezogen. Die genaue Implementierung der spezifischen Widgets wird an den geeigneten Stellen in der restlichen Dokumentation aufgezeigt.

4.2 Account-System

Als erste Funktion wird die Struktur der Benutzerverwaltung implementiert, da ein Account Voraussetzung zur Nutzung der Anwendung ist. Die Erstellung und Verwaltung von Accounts mit Firebase Authentication beginnt mit der Konfiguration dieser Komponente über die Firebase-Konsole. Diese ist unter `https://console.firebase.google.com` erreichbar. Nach Anwählen der Funktion „Authentication“ muss ausgewählt werden, welche Arten von Authentifikation möglich sein sollen, da Firebase Authentication neben der Account-Erstellung via E-Mail auch die Authentifizierung mittels Telefonnummer oder bestehender Accounts diverser Dienste wie Twitter, Google oder Facebook anbietet. Es wird die Nutzerverwaltung mittels E-Mail-Adresse gewählt.

Als nächstes ist es nötig, der App die entsprechenden Firebase-Bibliotheken hinzuzufügen. Das erste nötige Paket ist *firebase_core,* das „Grundpaket“ von Firebase. Das zweite Paket ist *firebase_auth,* das die Authentication-Funktionen von Firebase beinhaltet. Zusätzlich wird das Paket *firebase_ui_auth* bezogen, welches einige vorgefertigte Flutter-Widgets und -Dienstprogramme beinhaltet, um Firebase Authentication in die Benutzeroberfläche der App zu integrieren.

Nun können die Pakete in der App verwendet werden. Um Pakete in einer *.dart*-Datei zu verwenden, müssen sie am Anfang jener Datei importiert werden.

```
1 import 'config/firebase_options.dart';
2 import 'package:firebase_auth/firebase_auth.dart' hide
  EmailAuthProvider;
3 import 'package:firebase_core/firebase_core.dart';
4 import 'package:firebase_ui_auth/firebase_ui_auth.dart';
5 import 'package:flutter/material.dart';
6 import 'package:flutter/cupertino.dart';
7 import 'package:flutter_platform_widgets/flutter_platform_widgets.dart';
```

Listing 1: Import von Paketen in main.dart

Listing 1 zeigt den Import der Pakete am Anfang von *main.dart.* In Z. 1-4 wird die zuvor angelegte Firebase-Konfigurationsdatei zusammen mit den Firebase-Paketen importiert, wobei in Z. 3 die Klasse `EmailAuthProvider` aus dem Paket

firebase_auth ignoriert wird, um die gleichnamige Klasse aus *firebase_ui_auth* zu verwenden und einen Konflikt der beiden Pakete zu vermeiden. In den Z. 5-7 werden die Flutter-eigenen Material/Cupertino Widget-Pakete und *flutter_platform_widgets* initialisiert. Der Import nötiger Pakete wird in der restlichen Implementierung nicht näher dokumentiert.

Nun kann Firebase Auth in der Funktion `main` initialisiert werden, welche beim Start einer Flutter-Anwendung ausgeführt wird.

```
1 void main() async {
2   WidgetsFlutterBinding.ensureInitialized();
3   await Firebase.initializeApp(options:
  DefaultFirebaseOptions.currentPlatform);
4   FirebaseUIAuth.configureProviders([
5     EmailAuthProvider(),
6   ]);
7   runApp(MyApp());
8 }
```

Listing 2: Die Funktion main in main.dart

Die Funktion `main()` in Z. 1 soll keinen Rückgabewert liefern und erhält demnach den Marker `void`. Zusätzlich wird sie als `async` definiert, da die Funktion `Firebase.initializeApp` in Z. 3 abgeschlossen muss (Schlüsselwort `await`), ehe die Authentifikations-Anbieter in Z. 4-6 initialisiert werden. In Z. 2 wird `WidgetsFlutterBinding.ensureInitialized()` ausgeführt. Diese Anweisung stellt sicher, dass Flutter-Widgets korrekt initialisiert werden. Schließlich wird in Z. 7 ein Widget der Klasse `MyApp` mit der Funktion `runApp()` ausgeführt.

4.3 Navigation

`MyApp` (**Listing 3**) stellt somit die Anwendung an sich dar, die startet, nachdem die Flutter-Bindung und Firebase Authentication initialisiert sind.

```
1  class MyApp extends StatelessWidget {
2    @override
3    Widget build(BuildContext context) {
4      final providers = [EmailAuthProvider()];
5      return PlatformProvider(
6        builder: (context) => PlatformTheme(
7          builder: (context) => PlatformApp(
8            localizationsDelegates: <LocalizationsDelegate<dynamic>>[
9              DefaultMaterialLocalizations.delegate,
10             DefaultWidgetsLocalizations.delegate,
11             DefaultCupertinoLocalizations.delegate,
12           ],
```

```
13            title: 'Listipede',
14            material: (_, __) => MaterialAppData(
15              theme: listiTheme,
16            ),
17            initialRoute: FirebaseAuth.instance.currentUser == null
18                ? '/sign-in'
19                : '/listoflists',
20            routes: {
21              '/': (context) => const ListOfListsScreen(),
22              '/sign-in': (context) => SignInScreen(
23                    providers: providers,
24                    actions: [
25                      AuthStateChangeAction<SignedIn>((context, state) {
26                        Navigator.pushReplacementNamed(context,
    '/listoflists');
27              '/listoflists': (context) => const ListOfListsScreen(),
28            },
29          ),
30        ),
31      );
32    }
33  }
```

Listing 3: MyApp in main.dart (vorläufig)

In Z. 1 wird **MyApp** als Widget definiert. Ein **StatelessWidget** ist eine statische Ansicht, deren Darstellung nicht von veränderlichen Daten abhängt. In Z. 2-3 wird die Methode **build** überschrieben, um die Benutzeroberfläche des Widgets zu erstellen. Als Parameter für **build** wird ein **BuildContext** benötigt. Der **BuildContext** ermöglicht es der **build**-Methode, auf Ressourcen und Eigenschaften zuzugreifen, die für das Layout und die Darstellung des Widgets relevant sind. Dieser Kontext dient als Kommunikationsmittel zwischen verschiedenen Teilen der Benutzeroberfläche und ermöglicht es, das Widget entsprechend seiner Position und Umgebung anzupassen. Somit spielt der **BuildContext** eine zentrale Rolle bei der Erstellung und Anordnung von Widgets in einer Flutter-Anwendung. In diesem Fall wird `context` als **BuildContext** in der **build**-Methode angegeben. `context` wird automatisch von Flutter bereitgestellt und ist ein Objekt, das Informationen über die Position des aktuellen Widgets in der Widget-Hierarchie enthält.

build gibt ein Objekt des Typs **Widget** zurück (s. Z. 3). Das Widget, das zurückgegeben wird (=`return`) ist ein **PlatformProvider** (Z. 5): Dies ist das erste verwendete Widget aus *flutter_platform_widgets,* das zusammen mit dem **PlatformTheme** und **PlatformApp** genutzt wird, um plattformspezifische Funktionalitäten und Designs zu ermöglichen. So wird gewährleistet, dass die App von

Grund auf für die beiden Plattformen als eine `MaterialApp` bzw. `CupertinoApp` ausgegeben wird. Der wichtigste Parameter, der in der `PlatformApp` eingerichtet wird, ist die Routenführung der Anwendung:

In Z. 20-28 werden die Ansichten für die Routenführung der Anwendung definiert. Es gibt Routen für die Hauptansicht der Einkaufslisten (`'/listoflists'`), die Anmeldeansicht (`'/sign-in'`), und da eine `CupertinoApp` eine Ansicht braucht, auf die bei Fehlern zurückgegriffen wird, wird eine „leere“ Startseite (`'/'`) aufgeführt.

Die Route `'/sign-in'` führt zum vorgefertigten `SignInScreen`-Widget aus *firebase_ui_auth*, um die Anmeldeansicht zu handhaben, und es gibt eine Aktion für den Fall, dass der Benutzer erfolgreich angemeldet ist (`AuthStateChangeAction`), um den Nutzer zur Listenansicht weiterzuleiten. Die Listenansicht wird gleichermaßen für die Routen `'/listoflists'` und `'/'` definiert: `ListOfListsScreen` wird in **Kapitel 4.3** implementiert. Alle neuen Ansichten in der Anwendung werden im Folgenden als Teil ihrer Implementierung auf ähnliche Weise in *main.dart* hinterlegt, ohne dass darauf explizit hingewiesen wird.

Die Startseite der App (`initialRoute`) wird abhängig vom Authentifizierungsstatus des Benutzers festgelegt (Z. 17-19). In Z. 14-16 wird für `material` (also Android) als Design (=`theme`) `listiTheme` angegeben. `listiTheme` wird in einer neuen Datei *style.dart* definiert, die im Ordner *config* platziert wird. Dort werden zudem Farbpaletten für die Anwendung vergeben. **Abbildung 17** zeigt die fertige Ansicht des Log-ins, nachdem Authentifikation und Flutter-interne Navigation eingerichtet sind.

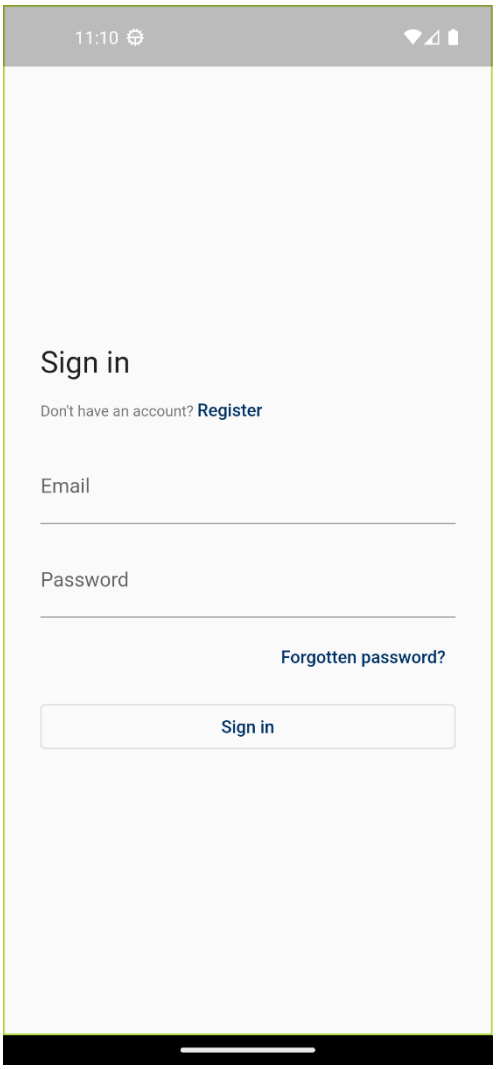

Abbildung 17: Ansicht des Log-ins

4.4 Home-Ansicht

Bevor die Home-Ansicht `ListOfListsScreen` dargelegt wird, werden einige ihrer Bestandteile implementiert. Die Klassen `ShoppingList`, `Product`, `ListEntry` und `Shop` werden mit Attributen gemäß des Entity-Relation-Diagramm (**Abbildung 13**) unter *domain/entities* in einer entsprechend benannten *.dart*-Datei erstellt.

4.4.1 App-Leiste

Unter *presentation/widgets* wird die erste sichtbare Widget-Klasse der Anwendung eingeführt: Die obere App-Leiste in der Einkaufslistenansicht, `MainAppBar` (**Listing 4**), welche standardmäßig ein großes Design mit Logo hat und bei Scrollen zu einer herkömmlichen Material/Cupertino App-Leiste wird.

```
1 class MainAppBar extends StatefulWidget {
2   MainAppBar({super.key});
3   @override
4   State<MainAppBar> createState() => _MainAppBarState();
```

```
}

class _MainAppBarState extends State<MainAppBar> {
  bool _pinned = true;
  bool _snap = false;
  bool _floating = false;
  @override
  Widget build(BuildContext context) {
    if (Platform.isAndroid) {
      return SliverAppBar(
        automaticallyImplyLeading: false,
        pinned: _pinned,
        snap: _snap,
        floating: _floating,
        expandedHeight: MediaQuery.of(context).size.height / 3,
        actions: [MainPopupMenu()],
        flexibleSpace: FlexibleSpaceBar(
          expandedTitleScale: 2.5,
          collapseMode: CollapseMode.pin,
          centerTitle: true,
          title: LayoutBuilder(builder: (context, constraints) {
            return Text(
              'Listipede',
              style: TextStyle(
                 fontWeight: constraints.maxHeight > kToolbarHeight * 2
                      ? FontWeight.w200
                      : FontWeight.normal),
            );
          }),
          background: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              SvgPicture.string(
                evidentmediawhite,
                allowDrawingOutsideViewBox: true,
                height: MediaQuery.of(context).size.width / 3,
              ),
              SizedBox(
                height: 20,
              ),
            ],
          ),
```

```
        ),
      );
    } else {
      return CupertinoSliverNavigationBar(
        automaticallyImplyLeading: false,
        largeTitle: Row(
          children: [
            SvgPicture.string(
              evidentmediablue,
              allowDrawingOutsideViewBox: true,
              height: 35,
            ),
            Text(' Listipede')
          ],
        ),
        middle: Text('Listipede'),
        alwaysShowMiddle: false,
        trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            PlatformIconButton(
              cupertino: (_, __) => CupertinoIconButtonData(
                padding: EdgeInsets.zero,
                icon: Icon(CupertinoIcons.add),
                onPressed: () {
                  showPlatformDialog(
                    context: context,
                    builder: (context) => ListCreationDialog(),
                  );
                },
              ),
            ),
            MainPopupMenu(),
          ],
        ),
        backgroundColor: Colors.white,
        border: Border(bottom: BorderSide(color: Colors.grey)),
      );
    }
  }
}
```

Listing 4: MainAppBar in presentation/widgets/main_app_bar.dart

Anders als bisherige Widgets ist `MainAppBar` ein `StatefulWidget`, weil es aufgrund seines Verhaltens verschiedene Zustände benötigt. Die Klasse selbst wird in Z. 1-5 definiert. Sie verweist auf den `_MainAppBarState` als Anfangszustand, der ab Z. 7 implementiert wird. Da das Plattform-Widget-Paket kein vorgefertigtes Widget zu diesem Zweck liefert, wird das Widget für Android als `SliverAppBar` (Z. 13-48) und für iOS als `CupertinoSliverNavigationBar` (Z. 49-87) separat definiert. Beide Widgets bieten das gewünschte Scroll-Verhalten. Die `MainAppBar` zeigt somit die quasi-Verdoppelung des Codes mit der if-Kondition `Platform.isAndroid` (Z. 13), um ein funktionell ähnliches Widget angemessen an die Plattformen anzupassen.

Die in den App-Leisten enthaltenen Logos im SVG-Format werden als String an anderer Stelle definiert und unter Verwendung des Pakets *flutter_svg* als `SvgPicture.string` in den Z. 37-41 bzw. 54-58 eingebunden.

Beide Versionen der Navigationsleiste beinhalten einen Button, der `MainPopupMenu` ausführt, welches als plattformabhängiges `PlatformPopupMenu` eingeführt wird. In diesem Menü soll die Navigation zu Menüs für das Profil (`'/profile'`), die Verwaltung erstellter Produkte (`'/my-products'`) und ein Platzhalter für ein einzusetzendes Impressum (`'/about'`) stattfinden.

4.4.2 Erstellung von Einkaufslisten

Für iOS wird in Z. 67-78 zusätzlich ein weiterer Button als Ersatz für den geplanten Floating Action Button der Android-Version eingesetzt, der einen Dialog zum Hinzufügen einer neuen Einkaufsliste öffnen soll (**Listing 5**).

```
1  class ListCreationDialog extends StatelessWidget {
2    final TextEditingController _listNameController =
   TextEditingController();
3
4    ListCreationDialog({super.key});
5    @override
6    Widget build(BuildContext context) {
7      return PlatformAlertDialog(
8        title: const Text('Neue Einkaufsliste erstellen'),
9        content: Column(
10         mainAxisSize: MainAxisSize.min,
11         children: [
12           PlatformTextFormField(
13             controller: _listNameController,
14             hintText: 'Name der Liste',
15           ),
16         ],
17       ),
```

```
18        actions: [
19          PlatformDialogAction(
20            onPressed: () {
21              Navigator.of(context).pop();
22            },
23            child: const Text('Zurück'),
24          ),
25          PlatformDialogAction(
26            onPressed: () async {
27              final listId = const Uuid().v4();
28              final listName = _listNameController.text;
29              final user = FirebaseAuth.instance.currentUser;
30              if (listName.isEmpty) {
31                return;
32              }
33              if (user == null) {
34                return;
35              }
36              final newShoppingList = ShoppingList(
37                id: listId,
38                name: listName,
39                userIds: [user.uid],
40              );
41              await ShoppingListRepository()
    .createShoppingList(newShoppingList);
42              Navigator.of(context).pop();
43            },
44            child: const Text('OK'),
45          ),
46        ],
47      );
48    }
49  }
```

Listing 5: ListCreationDialog

Der plattformabhängige Dialog `PlatformAlertDialog` (Z. 7) enthält ein Textfeld (`PlatformTextFormField`, Z. 12-15), in dem der Benutzer den Namen der neuen Liste eingeben kann. Zwei Schaltflächen ("Zurück" und "OK"), werden durch `PlatformDialogAction`-Widgets dargestellt. Beim Tippen auf "OK" werden einige Aktionen durchgeführt:

Unter Zuhilfenahme des Pakets *uuid* wird eine eindeutige `id` für die neue Liste generiert, der eingegebene Name der Liste und der aktuelle Firebase-Benutzer

werden abgerufen und eine neue `ShoppingList` wird mit diesen Daten initialisiert.

Die erstellte Einkaufsliste wird an eine Funktion `createShoppingList` aus `ShoppingListRepository` weitergegeben. Diese Funktion soll die Daten der Liste in die Firebase Echtzeit-Datenbank speichern. `ShoppingListRepository` wird in *domain/repositories/shopping_list_repository.dart* erstellt (**Listing 6**).

```
1  class ShoppingListRepository {
2    Future<void> createShoppingList(ShoppingList shoppingList) async {
3      print('Creating shopping list...');
4      final listId = shoppingList.id;
5      final userId = FirebaseAuth.instance.currentUser?.uid;
6      final DatabaseReference listsRef =
           FirebaseDatabase.instance.ref().child('shoppinglists/$listId/');
7      print('List ID: $listId, User ID: $userId');
8      try {
9        await listsRef.set({
10         'name': shoppingList.name,
11         'userIds': {'$userId': true},
12       });
13       print('Shopping list created successfully!');
14     } catch (e) {
15       print('Error creating shopping list: $e');
16     }
17   }
18 }
```

Listing 6: ShoppingListRepository

Für das Nutzen der Echtzeitdatenbank ist das Paket *firebase_database* nötig; die darin enthaltene Funktion `set` in Z. 9-12 erstellt im Datenbankpfad `'shopping-lists/'` aus Z. 6 einen Knoten im JSON-Format für die Echtzeit-Datenbank, dessen Bezeichnung als die Listen-ID `$listId` festgelegt ist. In Unterknoten wird der Name der Liste und der Nutzer hinterlegt. Genauer wird für den Nutzer ein boolean-Wert in den Unterknoten `'userIds'` eingesetzt, dessen Bezeichnung die Nutzer-ID ist (Z. 11). Dieses Vorgehen wird als unverhältnismäßig kompliziert wahrgenommen, jedoch unterstützt die Firebase Echtzeit-Datenbank keine Gruppierung durch Arrays o.Ä.

Außerdem muss die Echtzeit-Datenbank in Firebase eingerichtet werden. Dies geschieht in der Firebase-Konsole nach Auswahl einer Region für die Datenbank und einer Einrichtung von Sicherheitsregeln.

4.4.3 Darstellung von Einkaufslisten

Die mit `createShoppingList` erstellten Listen müssen aus der Firebase Echtzeit-Datenbank gelesen werden, damit sie dargestellt werden können. Dafür wird die Funktion `shoppingListsStreamForUser` in einem neuen Repository `ListOfListsRepository` erstellt (**Listing 7**).

```
1  Stream<List<ShoppingList>> shoppingListsStreamForUser(String userId) {
2    return shoppingListsReference.onValue.map((event) {
3      final dynamic data = event.snapshot.value;
4      final List<ShoppingList> shoppingLists = [];
5      if (data is Map<dynamic, dynamic>) {
6        data.forEach((key, value) {
7          if (value is Map<dynamic, dynamic> &&
8              value.containsKey('name') &&
9              value.containsKey('userIds') &&
10             value['userIds'][userId] == true) {
11           shoppingLists.add(
12             ShoppingList(
13               id: key,
14               name: value['name'],
15               userIds: List<String>.from(value['userIds'].keys),
16             ),
17           );
18         }
19       });
20     }
21     return shoppingLists;
22   });
23 }
```

Listing 7: shoppingListsStreamForUser in ListOfListsRepository

`shoppingListsStreamForUser` gibt einen `Stream` aus (Z. 1): Das ist ein asynchroner Datenstrom, der kontinuierlich Daten liefert. Konkret liefert dieser `Stream` unter Verwendung der Firebase-Funktion `onValue.map` in Z. 2 eine Liste von Objekten der Klasse `ShoppingList`, die sich aus allen Einkaufslisten in der Datenbankreferenz `shoppingListsReference` zusammensetzt, die den momentanen Nutzer im Feld `'userIds'` hinterlegt haben. `onValue.map` aktualisiert die Liste vom `Stream` immer dann, wenn ein relevantes Datenbank-Ereignis stattfindet.

Die Einkaufslisten in der Datenbank können nun als Liste mit Objekten der Klasse `ShoppingList` an ein Widget weitergegeben werden, das die optische Liste aus den Objekten erstellt.

```dart
class ShoppingListView extends StatelessWidget {
 final List<ShoppingList> shoppingLists;
 const ShoppingListView({super.key, required this.shoppingLists});
 @override
 Widget build(BuildContext context) {
   return ListView.builder(
     itemCount: shoppingLists.length,
     itemBuilder: (context, index) {
       final shoppingList = shoppingLists[index];
       return Card(
         shape: RoundedRectangleBorder(
           borderRadius: BorderRadius.circular(15.0),
         ),
        margin: EdgeInsets.symmetric(horizontal: 10.0, vertical: 5.0),
         child: SizedBox(
           height: 100,
           child: ClipRRect(
             borderRadius: BorderRadius.circular(15.0),
             child: Center(
               child: ListTile(
                 title: Text(
                   shoppingList.name,
                   style: TextStyle(fontSize: 20),
                   maxLines: 2,
                   overflow: TextOverflow.ellipsis,
                 ),
                 trailing: PlatformIconButton(
                   materialIcon: const Icon(Icons.more_vert),
                   cupertinoIcon: const
Icon(CupertinoIcons.ellipsis_vertical),
                   onPressed: () {
                     shoppingListBottomSheet(context, shoppingList);
                   },
                 ),
                 onTap: () {
                   Navigator.pushNamed(context, '/shopping-list',
                       arguments: shoppingList);
                 },
               ),
             ),
           ),
         ),
```

```
42          );
43        },
44      );
45    }
46  }
```

Listing 8: ShoppingListView in presentation/widgets/shopping_list_view.dart

`ShoppingListView` stellt die Einkaufslisten mit einem `ListView.builder` (Z. 6) dar. Ein `ListView` ist in Flutter eine scrollbare Liste. Konkret ist jedes Objekt der optischen Liste ein `Card` Widget (Z. 10-42), das den Namen der Einkaufsliste und einen Button für Einstellungen anzeigt. Dieser `PlatformIconButton` (Z. 27-33) führt die Funktion `shoppingListBottomSheet` aus, die ein plattformabhängiges Bottom Sheet anzeigt. Dieses Bottom Sheet muss ähnlich wie die `MainAppBar` aus **Listing 4** für beide Plattformen separat programmiert werden.

Das Bottom Sheet enthält Optionen zur Verwaltung einer Einkaufsliste, die in `ListOfListsRepository` als Funktionen `listRenameDialog`, `listDeleteDialog` und `listLeaveDialog` erstellt werden. Diese Funktionen zeigen analog zum `listCreationDialog` aus **Listing 5** Dialogfenster an und ändern anhand Input vom Nutzer die Daten der entsprechenden Einkaufsliste in der Datenbank. Die Option `listLeaveDialog` ist nur verfügbar, wenn die Einkaufsliste mehr als einen Nutzer hinterlegt hat, um zu vermeiden, dass Einkaufslisten ohne Nutzer entstehen.

Die `onTap`-Funktion der Einkaufslisten-`Card` (Z. 34-37) navigiert in die neue Route `'/shopping-list'`, sobald der Nutzer die Einkaufsliste antippt. Hierfür wird das `ShoppingList`-Objekt als Parameter für die Ansicht weitergegeben. Die Implementierung dieser Ansicht erfolgt in **Kapitel 4.5**.

4.4.4 ListOfListsScreen

Nun kann die finale Ansicht in Form von `ListOfListsScreen` implementiert werden. `ListOfListsScreen` fügt den Stream der Einkaufslisten als `_shoppingListsStream` in einen `PlatformScaffold` aus *flutter_platform_widgets* ein (**Listing 9**). Scaffolds in Flutter bilden das „Grundgerüst“ einer Material-/Cupertino-Ansicht.

```
1  PlatformScaffold(
2      body: NestedScrollView(
3        headerSliverBuilder: (BuildContext context, bool
   innerBoxIsScrolled) {
4          return <Widget>[
5            MainAppBar(),
6          ];
7        },
```

```
8         body: StreamBuilder<List<ShoppingList>>(
9           stream: _shoppingListsStream,
10          builder: (context, snapshot) {
11            if (snapshot.hasError) {
12              return Text('Error: ${snapshot.error}');
13            }
14            if (snapshot.connectionState == ConnectionState.waiting) {
15              return Center(child:
    PlatformCircularProgressIndicator());
16            }
17            final shoppingLists = snapshot.data;
18            if (shoppingLists == null || shoppingLists.isEmpty) {
19              return const Center(
20                  child: Text(
21                      "Keine Einkaufslisten vorhanden... Mach doch
    eine!"));
22            }
23            return ShoppingListView(shoppingLists: shoppingLists);
24          },
25        ),
26      ),
27      material: (_, __) => MaterialScaffoldData(
28        floatingActionButtonLocation:
    FloatingActionButtonLocation.centerFloat,
29        floatingActionButton: FloatingActionButton(
30          onPressed: () {
31            showPlatformDialog(
32              context: context,
33              builder: (context) => ListCreationDialog(),
34            );
35          },
36          child: const Icon(Icons.add),
37        ),
38      ),
39    );
```

Listing 9: PlatformScaffold in ListOfListsScreen

Der `NestedScrollView` in Z. 2-26 ist nötig, um ein korrektes Scroll-Verhalten von „Sliver"-Widgets wie der `MainAppBar` zu gewährleisten. Mithilfe des `StreamBuilder` wird die Liste der Einkaufslisten via `ShoppingListView` angezeigt, sofern kein Fehler vorliegt oder der Nutzer in keinen Listen hinterlegt ist (Z. 8-25). In Z. 27-38 wird für die Android-Version der `FloatingActionButton` zur Erstel-

lung einer Einkaufsliste eingesetzt. Somit ist die Ansicht der Einkaufslisten für Android (**Abbildung 18**) und iOS (**Abbildung 19**) implementiert.

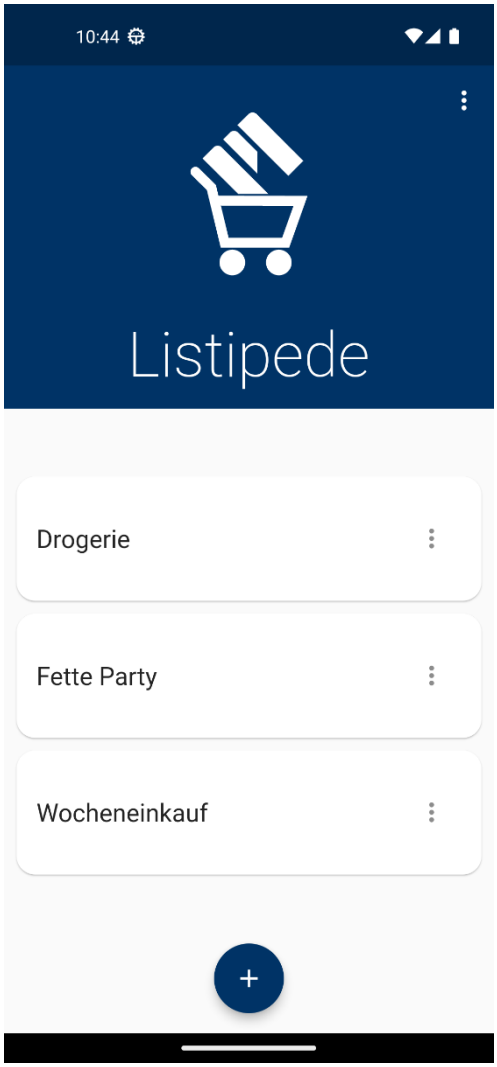

Abbildung 18: Home-Ansicht (Android)

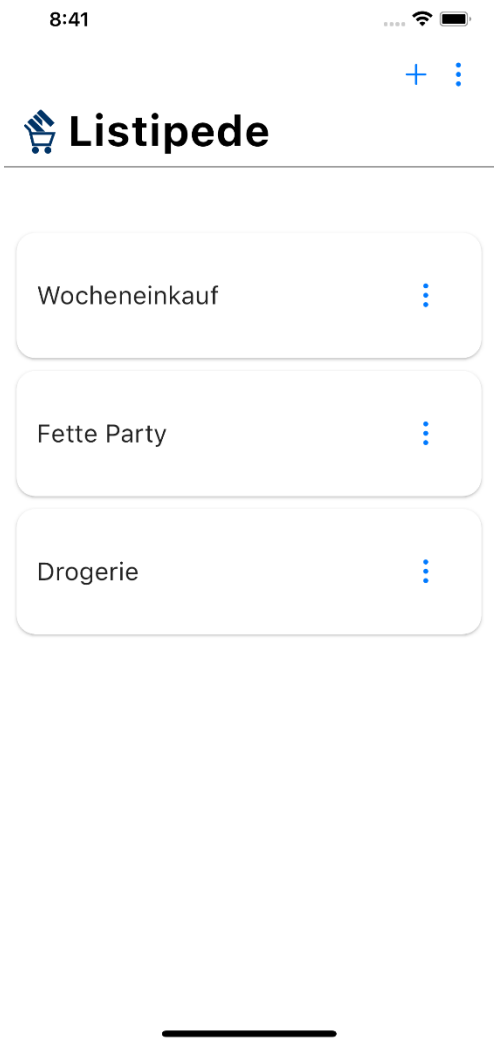

Abbildung 19: Home-Ansicht (iOS)

4.5 Ansicht einer Einkaufsliste

Im Folgenden wird die Implementierung der Einkaufslisten-Ansicht dokumentiert. Wie bei der Home-Ansicht (**Kapitel 4.4**) werden zunächst die einzelnen Komponenten erläutert, bevor sie im letzten Schritt zusammengesetzt werden. Außerdem wird darauf hingewiesen, dass der Einkaufsmodus aufgrund seiner Architektur eng mit der regulären Ansicht einer Einkaufsliste verwoben ist; jedoch werden alle Funktionalitäten des Einkaufsmodus nach Möglichkeit separat in **Kapitel 4.6** behandelt, um die Nachvollziehbarkeit der Dokumentation zu wahren.

4.5.1 Läden

Läden sollen – im Gegensatz zu den bisherigen Kategorien in der Datenbank – mit Firebase Firestore gehandhabt werden. Anders als bei der Echtzeit-

Datenbank handelt es sich bei Firestore um eine NoSQL-Datenbank, die dokumentenbasiert ist. Firestore-Dokumente sind Sammlungen von Feld-Wert-Paaren und können in Unterdokumente verschachtelt werden. Eine gleichzeitige Verwendung der beiden Datenbanken ist möglich, auch wenn die Verwendung in Flutter sich unterscheidet.

Die Initialisierung von Firestore erfolgt analog zur Echtzeit-Datenbank. Das Flutter-Paket *cloud_firestore* wird bezogen. Um den Namen des aktiven Shops einer Liste anzuzeigen, wird das `ShopRepository` erstellt und darin die `Stream`-Funktion `getCurrentShopNameStream` programmiert (**Listing 10**).

```
1  Stream<String> getCurrentShopNameStream(String userId, String listId) {
2      return FirebaseFirestore.instance
3          .collection('users')
4          .doc(userId)
5          .collection('usedLists')
6          .doc(listId)
7          .snapshots()
8          .asyncMap((snapshot) async {
9        final currentShopId = snapshot.data()?['currentShopId'] ?? '';
10       if (currentShopId.isNotEmpty) {
11         return await getShopNameById(userId, currentShopId);
12       } else {
13         return '';
14       }
15     });
16   }
```

Listing 10: getCurrentShopNameStream

`getCurrentShopNameStream` erhält durch den Befehl `snapshots` (Z.7) Änderungen im Firestore-Dokument `'/users/$userId/usedLists/$listId'` (vgl. Z. 2-6). Nutzer sollen also ein Dokument in `'/users'` erhalten, wo wiederum ihre benutzten Einkaufslisten und deren aktiver Laden als Attribut `'currentShopId'` vermerkt sind. Zusätzlich soll das Firestore-Dokument eines Nutzers alle Läden dieses Nutzers enthalten; dort wird der Name des aktiven Ladens anhand der ID `'currentShopId'` von `getShopNameById` ermittelt (**Listing 11**).

```
1    Future<String> getShopNameById(String userId, String shopId) async {
2      DocumentSnapshot<Map<String, dynamic>> shopDoc =
   await FirebaseFirestore
3          .instance
4          .collection('users')
5          .doc(userId)
6          .collection('shops')
7          .doc(shopId)
```

```
8         .get();
9     return shopDoc.data()?['name'] ?? '';
10  }
```

Listing 11: getShopNameById

Die einmalige Datenbankabfrage erfolgt nach ähnlichem Schema wie bereits beim Stream in `getCurrentShopNameStream`; nur wird der Befehl `get` genutzt (Z. 8).

Die Schaltfläche zur Anzeige des Ladens `ShopIndicator` wird erstellt. Sie besteht aus einem `StreamBuilder`, der den Namen des aktiven Shops mit `getCurrentShopNameStream` in einem simplen `Container` anzeigt, dessen Farbe und Design mit if-checks an die beiden Plattformen angepasst wird. Ein Antippen dieses `ShopIndicator` soll die Ladenauswahl öffnen, doch ein `Container` ermöglicht an sich keine solche Interkation. Deshalb wird der `Container` als `child` eines `GestureDetector`-Widgets gesetzt, das beim Antippen in die entsprechende Route navigiert. Diese Art von Vererbung und Verschachtelung ist eine häufige Begleiterscheinung der Widget-Architektur in Flutter.

4.5.2 Produktsuche

Um einen neuen Listeneintrag zu erstellen, wird eine Suchleiste benutzt. Darunter werden beim Suchen bestehende Produkte vorgeschlagen, die entweder voreingestellt oder vom Nutzer erstellt wurden. Dieses Suchleisten-Fenster wird unter dem Namen `EntryCreationDialog` erstellt, der hauptsächlich aus einem plattformabhängigen Textfeld besteht. Darunter werden Suchvorschläge von einem `StreamBuilder` in einer Liste dargestellt. **Listing 12** zeigt, wie der dazugehörige Stream `searchSuggestionsStream` den Firestore-Stream filtert, sodass nur voreingestellte oder vom Nutzer erstellte Produkte angezeigt werden.

```
1  .where(Filter.or(
2       Filter.and(
3           Filter("isCustom", isEqualTo: false),
4           Filter('searchName', isGreaterThanOrEqualTo: lowercaseQuery),
5           Filter('searchName',
6               isLessThanOrEqualTo: lowercaseQuery + '\uf8ff')),
7       Filter.and(
8         Filter('searchName', isGreaterThanOrEqualTo: lowercaseQuery),
9         Filter('searchName',
10            isLessThanOrEqualTo: lowercaseQuery + '\uf8ff'),
11        Filter("isCustom", isEqualTo: true),
12        Filter("createdBy",
13            isEqualTo: FirebaseAuth.instance.currentUser?.uid),
14          )))
```

Listing 12: Auszug aus der Datenbankabfrage von searchSuggestionsStream

Eine derartige Datenabfrage in Firestore stellt eine komplexe Suche dar, die zudem in Firebase eingerichtet werden muss. Die einzelnen kombinierten `and`-Suchfilter (Z. 2-6 und 7-14), die im `or`-Operator verwendet werden, müssen dort als Zusammengesetzter Such-Index erstellt werden, damit sie in dieser Weise benutzt werden können.

Wird bei der Abfrage kein bestehendes Produkt gefunden, das den Suchkriterien entspricht, dann wird automatisch ein neues Produkt in Firestore erstellt, das dem entsprechenden Nutzer zugewiesen wird. So ist in jedem Fall ein Produkt vorhanden, auf dessen Basis ein Listeneintrag in der Liste erscheint, indem es analog zu Einkaufslisten in der Echtzeit-Datenbank erstellt wird.

4.5.3 Listeneinträge

Nachdem Listeneinträge in einer Liste erstellt wurden, werden sie mit einer neuen `Stream`-Funktion `entriesForList` erfasst. Da der SI eines Eintrags von seinem Produkt (in Firestore) bestimmt werden soll, aber die anderen Attribute im Listeneintrag (in der Echtzeit-Datenbank) hinterlegt sind, gestaltet sich die Bereitstellung dieses `Stream` anspruchsvoll, da er einen weiteren `Stream` für den SI eines Eintrags benötigt und intern verwenden soll und eine derartige Verschachtelung asynchroner Operationen nicht ohne weiteres funktioniert.

Dieses Problem wird mit dem Paket *rxdart* gelöst. Genauer beinhaltet dieses Paket mit der Funktion `switchMap` die Möglichkeit, zwei Streams zu kombinieren. So wird der neue `sortIndexForProductInShopStream` in `entriesForList` eingesetzt.

Bevor die Listeneinträge von `entriesForList` zurückgegeben werden, werden sie mit der Dart-Funktion `sort` gemäß den Anforderungen sortiert (**Listing 13**).

```
1   listEntries.sort((a, b) {
2    if (a.isDone && b.isDone) {
3      return a. tickIndex!.compareTo(b.tickIndex!);
4    } else if (a.isDone) {
5      return -1;
6    } else if (b.isDone) {
7      return 1;
8    } else {
9      return a.sortIndex.compareTo(b.sortIndex);
10   }
```

Listing 13: Sortierung der Listeneinträge anhand tickIndex und sortIndex in entriesForList

Analog zu den Einkaufslisten werden diese Listeneinträge in einem neuen `ListView` namens `ListEntriesView` angezeigt, der sie in Form eines neuen Widgets darstellt: `ListEntryCard`. Dieses Widget ist im Grunde ein `PlatformListTile`, das die Informationen des Listeneintrags zusammen mit einem Kontrollkästchen

anzeigt. Ein Antippen der `ListEntryCard` öffnet einen Dialog zur Verwaltung von Menge, Mengentyp und Notiz eines Eintrags. Wird das Kontrollkästchen angetippt, informiert ein Dialog den Nutzer über das Starten des Einkaufsmodus. Sofern dieser Dialog bestätigt wird, startet der Einkaufsmodus (siehe **Kapitel 4.6**).

4.5.4 ShoppingListScreen

`ShoppingListScreen` wird als `PlatformScaffold` implementiert, das eine reguläre `PlatformAppBar` als App-Leiste verwendet. Dieser Aufbau ist bei allen restlichen Screens der App gleich. Der eigentliche Inhalt der Ansicht ist der `ListEntriesView` mit dem `ShopIndicator` darunter. Der Floating Action Button (Android) bzw. die Aktion in der App-Leiste (iOS) öffnet den `EntryCreationDialog`. Die Abbildungen **20** und **21** zeigen die Ansicht für Android/iOS.

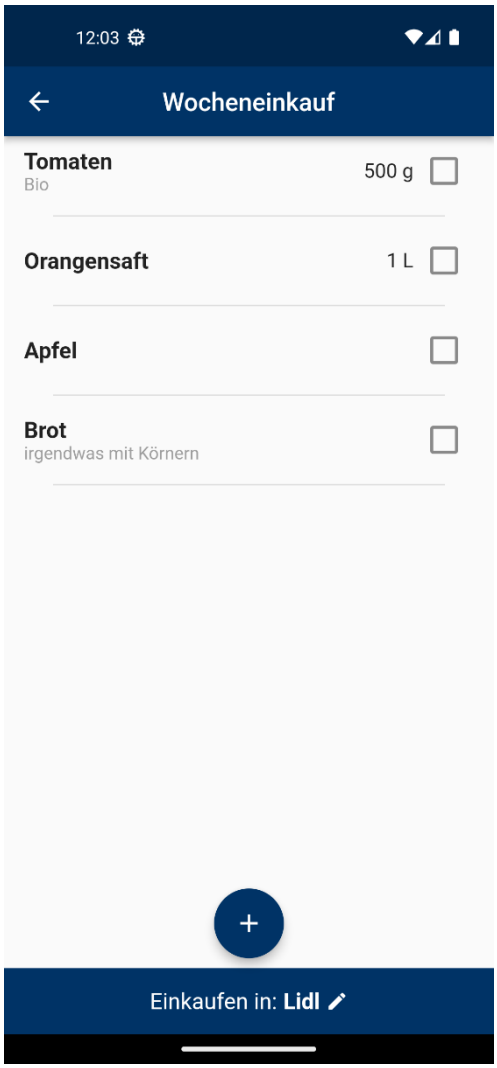

Abbildung 20: Einkaufslisten-Ansicht (Android)

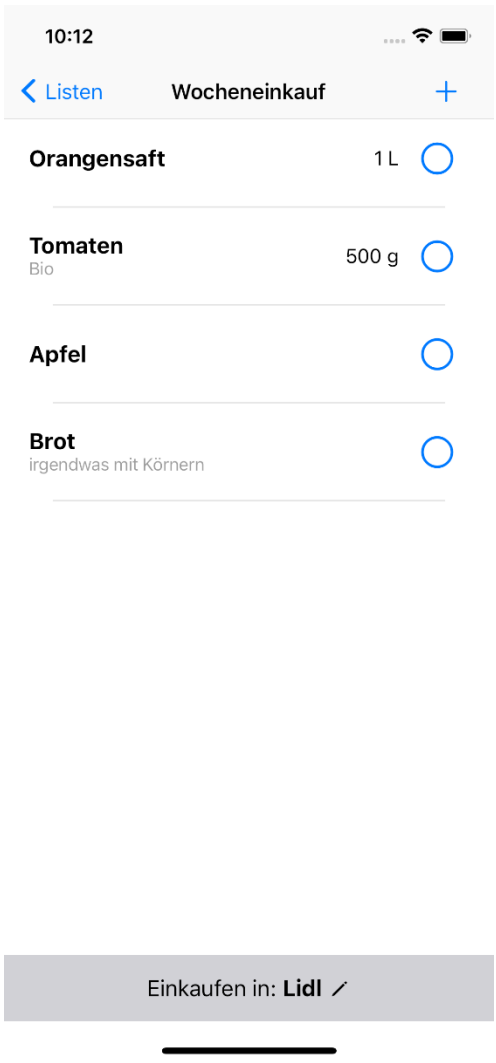

Abbildung 21: Einkaufslisten-Ansicht (iOS)

4.6 Einkaufsmodus

Die Einkaufsmodus-Ansicht stellt einen neuen Screen dar, der aber in vielen Teilen mit der regulären Ansicht einer Einkaufsliste übereinstimmt. Einige bestehende Widgets werden ergänzt und angepasst, indem sie ihr Verhalten mit if-Abfragen anhand eines neuen Boolean `shoppingMode` ausrichten. Dieser Boolean wird den Widgets als Parameter vom Screen gegeben, in dem sie sich befinden.

4.6.1 Änderungen an bestehenden Widgets

Der **`ShopIndicator`** zeigt im `shoppingMode` das aktive Geschäft nur an, anstatt eine Änderung zu ermöglichen, da die Sortierlogik nicht darauf ausgelegt ist, einen Einkaufsvorgang über mehrere Geschäfte zu tätigen.

Das Abhaken eines Listeneintrags findet als Antippen des Kontrollkästchens in **`ListEntryCard`** statt. Hierfür wird die Funktion **`toggleDone`** eingeführt, die in der Datenbank den Wert `isDone` eines Eintrags umkehrt und den `'tickIndex'` des Eintrags in der Datenbank einträgt bzw. löscht.

Diejenigen Einträge, die abgehakt sind, sollen per Drag-and-Drop verschiebbar sein. Hierzu wird eine grundsätzlich neue Definition von **`ListEntryCard`** vorgenommen, da ein regulärer **`ListView`** hierfür nicht reicht. Stattdessen wird im Einkaufsmodus ein **`ReorderableListView`** angezeigt, der die gewünschte Funktionalität bietet: genauer erhalten die **`ListEntryCard`**-Listeneinträge mit `isDone` eine Schaltfläche links, die die Möglichkeit des Drag-and-Drop signalisiert und die `'tickIndex'`-Werte der abgehakten Listeneinträge in der Datenbank neu vergibt, wenn ein Eintrag an eine andere Position verschoben wird. Zusätzlich wird ans Ende der Liste im Einkaufsmodus der neue Button **`EndShoppingButton`** platziert, der den Einkaufsvorgang beendet.

4.6.2 Neue Widgets

Im **`EndShoppingButton`** wird mit der Funktion **`assignNewSortIndexes`** die Berechnung der neuen SIs für die Produkte in Firestore vorgenommen.

```
1  for (int i = 0; i < doneEntries.length; i++) {
2    final entry = doneEntries[i];
3    final productId = entry.value['productId'];
4    final placementInTrip = i / (totalDoneEntries - 1);
5    final docRef = userDocRef
6        .collection('shops')
7        .doc(currentShopId)
8        .collection('sortedProducts')
9        .doc(productId);
10   final previousSortIndex = await docRef.get().then((value) {
11     if (value.exists &&
```

```
12          value.data() != null &&
13          value.data()!['sortIndex'] != null) {
14        return value.data()?['sortIndex'];
15      } else {
16        return placementInTrip;
17      }
18    });
19    final delta = placementInTrip - previousSortIndex;
20    if (delta == 0) {
21      await docRef
22          .set({'sortIndex': placementInTrip}, SetOptions(merge: true));
23     } else {
24
25    final deltaWithSortMagic =
26        (delta.abs() / delta) * 0.6 * (delta * delta);
27    final newSortIndex = previousSortIndex + deltaWithSortMagic;
28    await docRef
29        .set({'sortIndex': newSortIndex}, SetOptions(merge: true));
30  }}
```

Listing 14: Zuweisung der neuen SIs für die Einträge doneEntries in assignNewSortIndexes

Listing 14 zeigt den letzten Schritt in `assignNewSortIndexes`, in dem als erstes aus einer Produkt-/Laden-Kombination in Form vom Produkt als Firestore-Dokument in `'/shops/$currentShopId/sortedProducts'` der bestehende SI als `previousSortIndex` ausgelesen wird (Z. 5-14). Die Berechnung von `deltaWithSortMagic` in Z. 25-26 entspricht der Funktion $f(x)$ aus Sortier-Ansatz B mit dem Wert 0,6 als y. Schließlich wird der Wert auf `previousSortIndex` addiert (Z. 27) und in Z.28-29 in der Datenbank überschrieben.

War kein SI hinterlegt, wird stattdessen die Platzierung des Eintrags im Einkauf als `previousSortIndex` verwendet (Z. 15-17) und als SI gespeichert (Z. 20-24).

4.6.3 ShoppingModeScreen

Die aufgeführten Veränderungen und Ergänzungen werden im `ShoppingModeScreen` zusammengefügt. Dieser ist analog zum regulären `ShoppingListScreen` aufgebaut. Zusätzlich wird das `PlattformScaffold` als `child` in ein `WillPopScope`-Widget eingebettet, das beim Versuch, aus dem Screen zu navigieren, eine Funktion ausführt. Konkret wird ein plattform-abhängiges Dialogfenster angezeigt, das den Nutzer fragt, ob er den Einkauf abbrechen möchte. Wird es bestätigt, setzt die neue Funktion `resetShoppingMode` alle abgehakten Artikel zurück und navigiert aus dem Screen. **Abbildung 22** zeigt die Implementierung in Android, **Abbildung 23** in iOS.

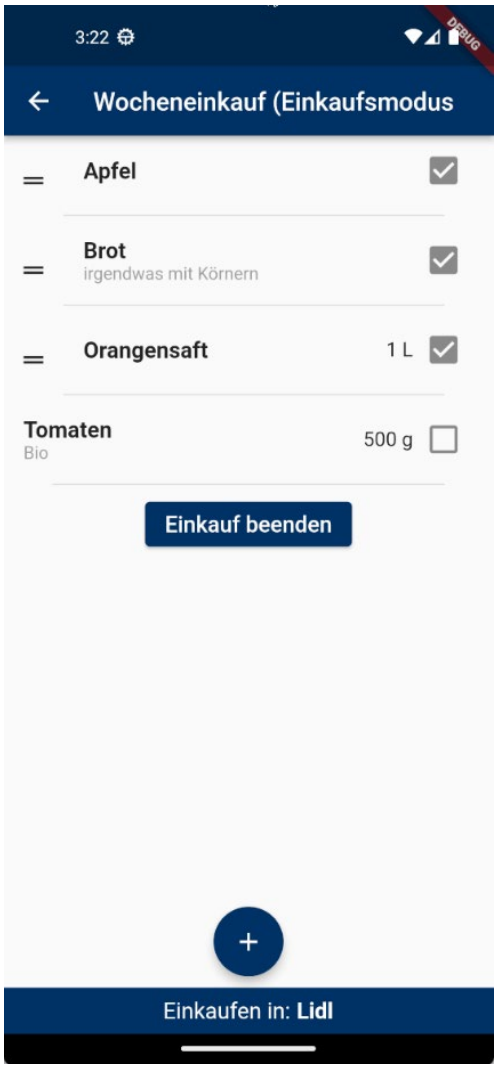

Abbildung 22: Einkaufsmodus (Android)

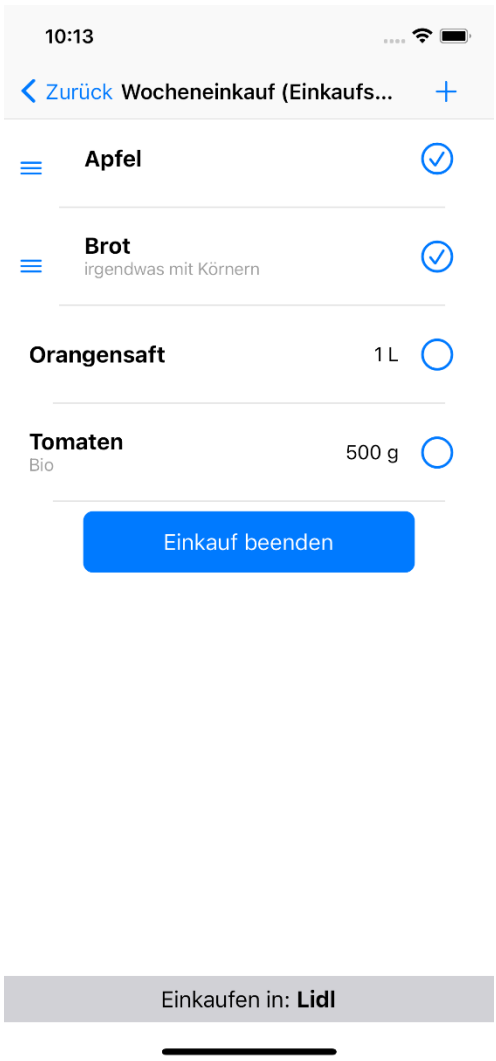

Abbildung 23: Einkaufsmodus (iOS)

4.7 Benutzermanagement

Benutzerdaten wie die E-Mail-Adresse können von der App nicht direkt aus Firebase Authentication ausgelesen werden. Dies stellt ein Problem für das Nutzermanagement der Einkaufslisten dar, wo z.B. das Einladen eines Nutzers mittels seiner E-Mail-Adresse vorgesehen ist. Um sicherzustellen zu können, dass alle Benutzer mit ID und E-Mail-Adresse in der Firestore-Datenbank hinterlegt sind, damit die Daten dort verwaltet werden können, beinhaltet Firebase Hintergrund-Funktionen, die bei Firebase-Ereignissen automatisierte Aktionen ausführen. Hierfür wird die Firebase CLI bezogen. Mit dem Befehl `firebase init functions` wird die Funktion definiert, die Nutzerdaten bei Registrierung in Firestore hinterlegt (**Listing 15**).

```
1 exports.addUserData = functions.auth.user().onCreate((user) => {
2   const uid = user.uid;
3   const email = user.email;
4   const userRef = admin.firestore().collection("users").doc(uid);
```

```
5   return userRef.set({
6     email: email,
7   });
8 });
```

Listing 15: Firebase Cloud-Funktion zum Hinterlegen der Nutzerdaten in Firestore

Analog dazu wird eine Funktion zur Entfernung der Daten bei Account-Löschung mit `onDelete` erstellt.

Ein neuer Screen namens `ListUsersScreen` wird erstellt.

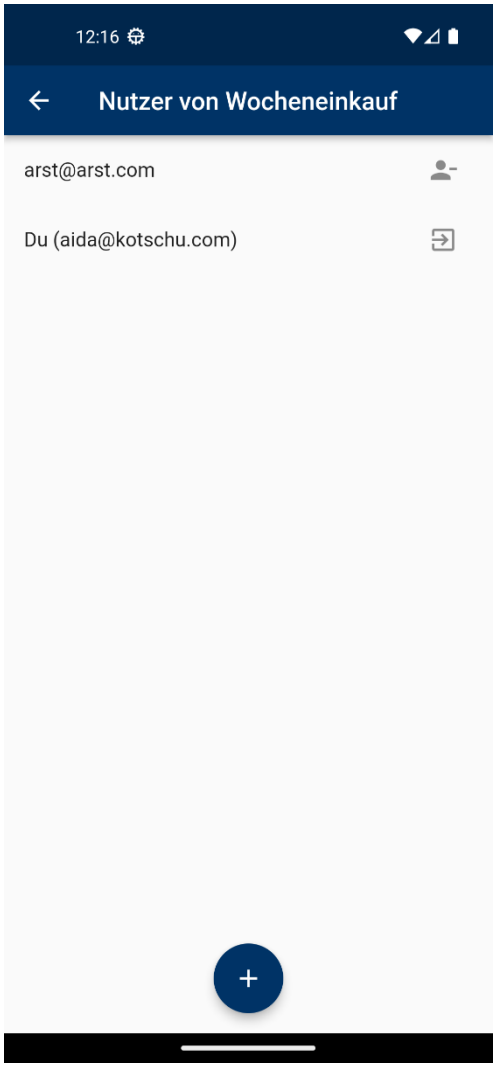

Abbildung 24: Nutzermanagement-Ansicht (Android)

Abbildung 25: Nutzermanagement-Ansicht (iOS)

Dieser Screen zeigt Nutzer in einem simplen `ListView` an, der Daten aus einem Firestore-Daten-`Stream` bezieht. Für das Hinzufügen eines Nutzers ist ein Dialogfenster im Floating Action Button (Android) bzw. `AppBar`-Button zuständig, in das die E-Mail-Adresse eingegeben werden soll und bei vorhandenem Nutzer die Datenbank aktualisiert. Zusätzlich werden Funktionen zum Entfernen von Gruppenmitgliedern und Verlassen der Liste eingesetzt.

Eine Navigation in diesen Screen wird als Menüpunkt im `shoppingListBottomSheet` aus **Kapitel 4.4.3** eingesetzt.

4.8 Sonstiges

Im Folgenden wird die Implementierung restlicher Komponenten erläutert, die aufgrund ihres niedrigen Umfangs und/oder Relevanz nur kurz beschrieben werden.

Um zu gewährleisten, dass nicht mehrere Personen gleichzeitig einen Einkauf mit einer Liste tätigen, wird das Dialogfenster `someoneShoppingDialog` eingeführt, das einen Nutzer bittet, entweder zu warten oder die Haken einer Liste zurückzusetzen, wenn man in der regulären Ansicht einer Einkaufsliste versucht in den Einkaufsmodus zu wechseln, während schon Artikel abgehakt sind.

Die Ansicht `ListUsersScreen` wird angezeigt, wenn der `ShopIndicator` angetippt wird. Hier werden Läden eines Nutzers als simple Liste angezeigt, ausgewählt und verwaltet (also erstellt/gelöscht/bearbeitet).

In `MyProductsScreen` werden alle Produkte eines Nutzers aufgelistet und verwaltet. `UserProfileScreen` ist ein Screen, in den die Ansicht des eigenen Profils aus *firebase_ui_auth* eingebettet ist. `AboutScreen` soll ein Impressum/Information über die App beinhalten. Diese drei Ansichten werden über den Button der App-Leiste in der Hauptansicht erreicht.

Der gesamte Quellcode der Anwendung Listipede ist **Anhang A** zu entnehmen.

5 Evaluation der Anwendung

Im folgenden Kapitel wird die in **Kapitel 3** konzipierte und in **Kapitel 4** implementierte App anhand des Anforderungskatalogs aus **Kapitel 2.6** bewertet.

5.1 Anforderung FA-1

Alle Punkte dieser Anforderung sind erfüllt: Einkaufslisten (**Kapitel 4.4**) mit abhakbaren Listeneinträgen (**Kapitel 4.5** und **4.6**), die auf Produkten (**Kapitel 4.5.2**) basieren, gewährleisten die nötige Funktion.

5.2 Anforderung FA-2

Das Account-System (**Kapitel 4.2**) und die Nutzerverwaltung (**Kapitel 4.7**) erfüllen diese Voraussetzung.

5.3 Anforderung FA-3

SIs, die anhand Einkäufen angepasst werden (**Kapitel 4.6.2**), geben die Reihenfolge von Listeneinträgen in einer Liste vor (**Kapitel 4.5.3**). Die Funktionalität der Sortierungsfunktion wird getestet, indem der fiktive Produktkatalog aus **Tabelle 2** in der Anwendung erstellt wird. Mithilfe einer Funktion `toggleDoneRandomly` wird eine zufällige Anzahl und Auswahl dieser Produkte in der richtigen Reihenfolge abgehakt, indem jeder Listeneintrag eine bestimmte Chance hat, abgehakt zu werden. In dieser Art werden mit einer 33%-Chance zum Abhaken 20 Einkäufe simuliert (Test 1). Anschließend werden die SIs der Artikel erfasst und zurückgesetzt. Der Vorgang wird mit der Hälfte der Einkäufe (10) aber doppelter Chance (66%) wiederholt (Test 2). Diese unterschiedlichen Chancen sollen unterschiedliche durchschnittliche Einkaufslisten-Längen simulieren, da eine höhere Abhak-Chance im Durchschnitt mehr abgehakte Produkte pro Einkauf bedeutet.

Produkt	SI		
	verteilt	Test 1	Test 2
Kartoffel	0	0	0
Apfel	0.091	0.022	0.143
Semmel	0.182	0.222	0.284
Zahnseide	0.273	0.304	0.360
Müllbeutel	0.364	0.381	0.499
Milch	0.455	0.650	0.618
Hackfleisch	0.545	0.521	0.659
Bohnen	0.636	0.818	0.754
Gurke	0.727	0.517	0.667
Schokolade	0.818	0.977	0.849
Nudeln	0.909	0.933	0.993
Kaffee	1	1	1

Tabelle 6: Ergebnis des Sortier-Tests (auf 3 Nachkommastellen gerundet)

Die **Tabelle 6** zeigt die Produkte in ihrer Reihenfolge von oben nach unten. Neben den Produkten sind ihre gleichmäßig zwischen 0 und 1 verteilten „idealen" SIs aufgeführt, sowie die SIs nach den Tests. SIs, deren Abfolge mit der Reihenfolge der Produkte übereinstimmt, sind grün hinterlegt.

Auch wenn Test 1 weniger Produkte in ihre exakte Reihenfolge bringt als Test 2, ist anhand der Werte erkennbar, dass die SIs nach beiden Tests nur minimale Abweichungen zu den ideal verteilten Werten aufweisen. Zur Quantifizierung dieses Umstands wird die durchschnittliche Abweichung der Testergebnisse von den gleichmäßig verteilten SIs berechnet. Um einen Vergleichswert für die Abweichungen der Tests zu erhalten, wird zusätzlich mittels 1,000,000 Durchläufen eines simplen Codes die durchschnittliche Abweichung von zufälligen Werten zu den gleichmäßig verteilten SIs ermittelt. Die Ergebnisse sind aus **Tabelle 7** ersichtlich.

Verfahren	∅ Abweichung
Test 1	0.0793
Test 2	0.0788
Zufällige Werte	0,3485

Tabelle 7: Durchschnitts-Abweichung der Tests und zufälliger Werte (auf 4 Nachkommastellen gerundet)

Da die Sortierlogik die Einträge erkennbar anhand der Einkäufe anordnet, wird die Anforderung als erfüllt bewertet.

5.4 Anforderung FA-4

Der Einbezug der Märkte ist gegeben (**Kapitel 4.5.1**). Die Anforderung ist erfüllt.

5.5 Anforderung FA-5

Die Anwendung ist sowohl technisch als auch optisch an die beiden Plattformen angepasst (Siehe **Kapitel 4.1**) und erfüllt somit diese Anforderung.

5.6 Anforderung FA-6

Die Anforderung **FA-6** ist nicht erfüllt. Die angegebene Priorität „mittel“ wird bereits früh in der Konzeption von Listipede als fehlerhaft eingestuft, da ein Ausbleiben von Push-Benachrichtigungen nicht als Mangel der Anwendung erachtet wird. Zugunsten einer möglichst guten Umsetzung der hoch priorisierten Anforderungen wird **FA-6** folglich für den Rahmen dieser Arbeit verworfen.

5.7 Anforderung FA-7

Die Rezept-Verwaltung aus **FA-7** ist im vorliegenden Prototyp nicht vorhanden. Ähnlich wie bei **FA-6** wird im Laufe der Implementierung die Optimierung der Kern-Anforderungen einem weiteren Feature vorgezogen.

5.8 Fazit

Alle Anforderungen an die Anwendung, deren Priorität als „hoch“ eingestuft wurden (**FA-1**, **FA-2**, **FA-3**, **FA-4**, **FA-5**) sind erfüllt. Die Priorisierung der fehlenden Features in **FA-6** und **FA-7** wurde bereits im Vorhinein als sekundär erachtet. Insgesamt wird die Realisierung vom Prototyp der Multiplattform-Anwendung Listipede als erfolgreich bewertet.

6 Hürden und Chancen der Multiplattform-Entwicklung mit Flutter und Dart

6.1 Hürden

Eine prinzipielle Hürde bei der Multiplattform-Entwicklung unabhängig vom Framework ist das Design der Nutzeroberfläche. Konkret soll eine Multiplattform-UI unter Umständen plattformspezifische Design-Richtlinien einbeziehen. Zur Handhabung einer Multiplattform-UI gibt es drei Ansätze (Barea et al., 2013, S. 11–12):

- Plattformabhängiges Design, das vorhandene Design-Richtlinien der Plattformen befolgt
- Eigenes, plattformübergreifendes Design
- Hybrid der beiden anderen Ansätze: Ein plattformübergreifendes Aussehen mit plattformspezifischen Interaktionsmöglichkeiten

In der Anwendung Listipede wird sich in **Kapitel 3.4** für ein plattformabhängiges Design der UI entschieden, da die Erstellung einer eigenen Nutzeroberfläche als sehr arbeitsintensiv eingeschätzt wird. Selbst die verwendete UI-Lösung stellt einen beachtlichen Arbeitsaufwand dar, da ein Äquivalent mancher Komponenten auf einer Plattform nicht existiert, beispielsweise der Floating Action Button in Googles Material Design, den es in iOS in keiner Form gibt. Solche Bestandteile benötigen auf der anderen Plattform eine Abhilfe. Zusätzlich wird die nötige quasi-Verdoppelung des Codes bei manchen plattform-spezifischen Widgets (z.B. die App-Leiste in **Kapitel 4.4.1**) als unvorteilhaft angesehen.

Die Widget-lastige Flutter-Architektur allgemein führt an vielen Stellen im Programmcode zu Verschachtelungen von Widgets, die eine Verkettung von Vererbungen bildet. Solche mehrfach-verpackten Widgets werden aus Entwicklersicht mit zunehmender Komplexität der Komponenten als schwer lesbar empfunden.

Die Arbeit mit Firebase in der App ist mit Komplikationen verbunden, da die verschiedenen Datenbanken in Firebase eine unterschiedliche Struktur und Programmiersyntax aufweisen. Insbesondere die simultane Nutzung von Firestore und Echtzeit-Datenbank in einer einzigen Komponente (**Kapitel 4.5.3**) benötigt aus Entwicklersicht eine komplexe Lösung. Die Verwendung von Firebase ist kein fester Bestandteil der Programmierung mit Flutter und Dart, wird aber von Google in der Flutter-Dokumentation erleichtert und somit gewissermaßen nahegelegt.

6.2 Chancen

Die Nutzung von Firebase bei der Multiplattform-Programmierung mit Flutter und Dart ist keineswegs nur als Hürde zu verstehen: Die vielen Möglichkeiten

der Einzelkomponenten und insbesondere die Auslagerung der Authentifikation ersparen viel Arbeit. Die Ordnerstruktur von Firestore wird als leicht verständlich bewertet. Das triviale Einrichten der Datenbanken und Module von Firebase wird ebenso als sehr zuvorkommend betrachtet.

Auch die bereits thematisierte Widget-Architektur stellt zugleich eine Chance dar, da die logische Struktur bei der Implementierung von Listipede dort positiv bewertet wird, wo große Verschachtelungen von Widgets vermieden werden, indem Unterkomponenten soweit möglich als kleinere Widgets in separaten Dateien definiert werden.

Flutter bietet viele Pakete, die im Verlauf der kompletten Implementierung hilfreich sind. Am markantesten für diese Arbeit ist *flutter_plattform_widgets*, welches in Listipede die Verdoppelung vom Code vieler quasi-identischer Komponenten verhindert. So kann eine kompakte Codebasis für mehrere Plattformen gepflegt werden.

Sofern eine kompakter Programmcode erfolgreich realisiert wird, zeigt dieser eine weitere Stärke von Multiplattform-Frameworks wie Flutter. Das Pflegen eines einzigen Quellcode ist im Vergleich mit nativer App-Entwicklung sehr effizient und senkt die Zugangsbarriere, da für Entwickler mit wenig Erfahrung nur das Erlernen des einzigen Frameworks nötig ist, anstatt den Umgang mit den Frameworks jeder Plattform separat zu erlernen. Dieser Punkt trifft selbst für Komponenten zu, die für die Plattformen einzeln erstellt werden: In der nativen App-Entwicklung wäre es nicht nur nötig, Teile der Anwendung doppelt zu programmieren, sondern dies müsste in den jeweiligen verschiedenen Frameworks und Programmiersprachen erfolgen.

Die einfache Initialisierung von Flutter wird genauso als Mehrwert für eine niedrige Zutrittsschranke in die App-Programmierung erachtet. Besonders positiv fällt der Befehl `doctor` auf. Die Dokumentation von Flutter wird als zufriedenstellend hilfreich für die Multiplattform-Programmierung angesehen. Best Practices aus der Dokumentation werden teilweise als Hinweise im Code (Lints) angezeigt, was für unerfahrene Entwickler positiv bewertet wird.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung der Ergebnisse

Die in **Kapitel 1.3** formulierte Forschungsfrage wird ausführlich in **Kapitel 6** beantwortet. Zusammenfassend bietet Flutter viele Hilfestellungen für Entwickler und Möglichkeiten für die kompakte Realisierung einer App auf mehreren Plattformen, besonders im Vergleich zur separaten nativen Realisierung einer Multiplattform-App. Gleichzeitig ist das Programmieren einer Multiplattform-Anwendung zwangsweise mit Mehraufwand im Gegensatz zu einer nativen App auf nur einer Plattform verbunden. Die Widget-basierte Architektur der Nutzeroberfläche in Flutter fördert vorteilhafte Code-Kompartimentierung, doch kann aus Entwicklerperspektive unübersichtlich werden. Firebase bietet viele Möglichkeiten als naheliegende Datenbank-Lösung in Flutter, auch wenn es mit Nachteilen verbunden ist.

7.2 Ausblick

Als weiterführende Erforschung plattformübergreifender Nutzeroberflächen wäre ein umfassender Vergleich verschiedener Multiplattform-Design-Ansätze aus Entwicklersicht interessant.

Bezüglich der Sortierfunktion der Anwendung wird angenommen, dass Möglichkeiten zur Verbesserung existieren. Für eine Optimierung des Systems wäre es nötig, handfestere messbare Metriken für die genaue Effizienz und Effektivität der Sortierlogik zu erarbeiten und anhand diesem Maß ausgiebige Tests verschiedener Varianten des Systems durchzuführen, die sinnvolle Stellschrauben aufzeigen. Eine solche Erarbeitung wird als wünschenswert erachtet.

Literaturverzeichnis

Apple (2007). *Apple erfindet mit dem iPhone das Mobiltelefon neu: Pressemitteilung vom 9. Januar 2007*. Zugriff am 12. August 2023, verfügbar unter https://web.archive.org/web/20080109043736/http://www.apple.com/de/pr/pr-infos2007/januar/iphone.html

Bak, L. (2011). *Dart: a language for structured web programming - The official Google Code blog*. Zugriff am 15. April 2023, verfügbar unter http://googlecode.blogspot.com/2011/10/dart-language-for-structured-web.html

Bak, L. & Lund, K. (2015). *Dart for the Entire Web*. Zugriff am 12. August 2023, verfügbar unter https://news.dartlang.org/2015/03/dart-for-entire-web.html

Barea, A., Ferre, X. & Villarroel, L. (2013). Android vs. iOS Interaction Design Study for a Student Multiplatform App. In C. Stephanidis (Hrsg.), *Communications in Computer and Information Science: Bd. 374, <>* (S. 8–12). Springer Berlin Heidelberg; Imprint: Springer. https://doi.org/10.1007/978-3-642-39476-8_2

Bigio, M. (2016). *Introducing Hot Reloading*. Meta. Zugriff am 12. August 2023, verfügbar unter https://reactnative.dev/blog/2016/03/24/introducing-hot-reloading

Bracken, C. (2017). *Release v0.0.6: Rev alpha branch version to 0.0.6, flutter 0.0.26 (#10010) · flutter/flutter*. Zugriff am 15. April 2023, verfügbar unter https://github.com/flutter/flutter/releases/tag/v0.0.6

Chisholm, K. (2022). *What's new in Flutter 3.3*. Zugriff am 12. August 2023, verfügbar unter https://medium.com/flutter/whats-new-in-flutter-3-3-893c7b9af1ff

DIN. (2018). *DIN EN ISO 9241-11:2018, Ergonomie der Mensch-System-Interaktion - Teil 11: Gebrauchstauglichkeit: Begriffe und Konzepte*. Beuth Verlag GmbH.

Friedman, N. (2013). *Announcing Xamarin 2.0*. Zugriff am 12. Juli 2023, verfügbar unter https://web.archive.org/web/20130627074458/http://blog.xamarin.com/announcing-xamarin-2.0/

Google (2023). *Application fundamentals*. Google. Zugriff am 12. August 2023, verfügbar unter https://developer.android.com/guide/components/fundamentals

Hume, D. A. (2018). *Progressive Web Apps*. Manning.

IDC (2023). *Marktanteile der führenden Betriebssysteme am Absatz von Smartphones weltweit vom 1. Quartal 2009 bis zum 2. Quartal 2023*. Zugriff am 12. August 2023, verfügbar unter https://de.statista.com/statistik/daten/studie/73662/umfrage/marktanteil-der-smartphone-betriebssysteme-nach-quartalen/

Jacobsen, J. & Meyer, L. (2019). *Praxisbuch Usability und UX: Was jeder wissen sollte, der Websites und Apps entwickelt. Onleihe. E-Book*. Rheinwerk Verlag; divibib GmbH.

Jobe, W. (2013). Native Apps Vs. Mobile Web Apps. *International Journal of Interactive Mobile Technologies (iJIM)*, *7*(4), 27. https://doi.org/10.3991/ijim.v7i4.3226

Meta (2023). *Core Components and Native Components*. Zugriff am 12. August 2023, verfügbar unter https://reactnative.dev/docs/intro-react-native-components

Metz, C. (2012). *'Firebase' Does for Apps What Dropbox Did for Docs*. Zugriff am 12. August 2023, verfügbar unter https://www.wired.com/2012/04/firebase/

Metzner, A. (2020). *Software Engineering - kompakt. Hanser eLibrary*. Carl Hanser Verlag GmbH & Co. KG. https://www.hanser-elibrary.com/doi/book/10.3139/9783446463653 https://doi.org/10.3139/9783446463653

Morrill, D. (2008). *Announcing the Android 1.0 SDK, release 1.* Google. Zugriff am 12. August 2023, verfügbar unter https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html

Nawrocki, P., Wrona, K., Marczak, M. & Sniezynski, B. (2021). A Comparison of Native and Cross-Platform Frameworks for Mobile Applications. *Computer*, *54*(3), 18–27. https://doi.org/10.1109/MC.2020.2983893

Occhino, T. (2015). *React Native: Bringing modern web techniques to mobile*. Zugriff am 12. August 2023, verfügbar unter https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/

Peinert-Elger, C. & Magerhans, A. (2023). *Quick Guide Usability: Wie Sie Produktflops vermeiden und eine nutzergerechte User Experience schaffen. Quick Guide*. Springer Fachmedien Wiesbaden; Imprint Springer Gabler. https://doi.org/10.1007/978-3-658-41469-6

Q. Huynh, M., Ghimire, P. & Truong, D. (2017). Hybrid App Approach: Could It Mark the End of Native App Domination? *Issues in Informing Science and Information Technology*, *14*, 49–65. https://doi.org/10.28945/3723

Raj, R. & Tolety, S. B. (2012, 9. Dezember). A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In I. S. C. Author (Hrsg.), *2012 Annual IEEE India Conference* (S. 625–629). IEEE. https://doi.org/10.1109/INDCON.2012.6420693

Robinson, O. (2014). *Finding simplicity in a multi-device world*. GfK. Zugriff am 12. Juli 2023, verfügbar unter https://www.gfk.com/blog/2014/03/finding-simplicity-in-a-multi-device-world

Rohr, M. (2018). *Sicherheit von Webanwendungen in der Praxis: Wie sich Unternehmen schützen können : Hintergründe, Maßnahmen, Prüfverfahren und Prozesse / Matthias Rohr* (2. Auflage). *Edition <Kes>*. Springer Vieweg. https://doi.org/10.1007/978-3-658-20145-6

Serrano, N., Hernantes, J. & Gallardo, G. (2013). Mobile Web Apps. *IEEE Software*, *30*(5), 22–27. https://doi.org/10.1109/MS.2013.111

Shankland, S. (2011). *Google debuts Dart, a JavaScript alternative*. Zugriff am 12. August 2023, verfügbar unter https://www.cnet.com/culture/google-debuts-dart-a-javascript-alternative/

Anhang A – Elektronischer Anhang

Übersicht elektronischer Anhänge:

- Quellcode der Anwendung
- Bachelorarbeit in elektronischer Form (PDF)
- Vollständige Dokumentation der Berechnungen aus **Kapitel 3.3.4**

zu finden unter https://github.com/aidusya/Bachelorarbeit

Anhang B – E-Mail-Korrespondenz mit Adrian Kühlewind

Von: a.kotschu@oth-aw.de

Gesendet: Wednesday, 31 May 2023 16:46

An: 'feedback@ponlist.de'

Betreff: Anfage Bachelorarbeit Sortierverfahren

Guten Tag Herr Kühlewind!

Ich programmiere im Rahmen meiner Bachelorarbeit (Multiplatform-Programmierung mit Dart und Flutter) eine Einkaufsliste. Grundidee für die Anwendung war eine sich selbst sortierende Einkaufsliste; bei der Recherche bin ich natürlich über ihre pon App gestoßen. Da ich an der Stelle sehr rätsle, wie genau die Sortierung funktionieren sollte, kontaktiere ich sie im Einvernehmen mit meinem Betreuer, ob Sie vielleicht verraten können, wie Sie die smarte Sortierungslogik gelöst haben?

Viele Grüße,

Aida Kotschu

Von: pon Support <feedback@ponlist.de>

Gesendet: Wednesday, 31 May 2023 16:53

An: a.kotschu@oth-aw.de

Betreff: [EXTERN] Re: Anfage Bachelorarbeit Sortierverfahren

Hallo Aida,

das ist nicht viel Magie:

pon - Häufige Fragen

my.ponlist.de

Viel Spaß beim Einkaufen mit pon,

Adrian Kühlewind

adrian@ponlist.de

Von: a.kotschu@oth-aw.de

Gesendet: Friday, 9 June 2023 15:40

An: 'pon Support'

Betreff: AW: [EXTERN] Re: Anfage Bachelorarbeit Sortierverfahren

Hallo nochmal Herr Kühlewind!

Vielen Dank für die Rückmeldung! Allerdings Beantwortet das FAQ die Frage nicht ganz; das grundsätzliche Prinzip ist verständlich, es ging mir eher um die tatsächliche Umsetzung einer solchen Sortierungslogik.

Mein Ansatz soweit ist, dass jeder Artikel eine Gewichtung zwischen 0 und 1 hinterlegt bekommt, anhand derer die Einträge in der Liste sortiert werden. Diese Gewichtungen sollen nach einem Einkaufs-„vorgang“ angepasst werden, aber wie genau man die Gewichtungen anhand der Reihenfolge der „Abhakungen“ eines Einkaufs verändert ist mir noch nicht ganz klar.

Es scheint mir sehr schwer anhand der Reihenfolge eines Einkaufs genau zu erkennen, in welche Richtung und wie weit die Gewichtung eines Produkts verschoben werden muss (oder auch nicht!), da man ja aus einem gegebenen Einkauf nur die Reihenfolge der Produkte zueinander weiß, ohne den Gesamtkontext aller anderen Produkte.

Vielleicht bin ich mit dem Ansatz aber auch grundsätzlich auf dem Holzweg?

Viele Grüße,

Aida Kotschu

Von: pon Support <feedback@ponlist.de>

Gesendet: Friday, 9 June 2023 21:01

An: a.kotschu@oth-aw.de

Betreff: [EXTERN] Re: [EXTERN] Re: Anfage Bachelorarbeit Sortierverfahren

Hallo Aida,

der Ansatz ist richtig. Es muss nicht zwangsweise eine Skala sein von 0-1 aber so in etwa funktioniert es. Bei pon haben alle Artikel einen Index und keiner ist doppelt vergeben.

Artikel haben einen SI und wenn man einen Artikel verschiebt wird dieser zwischen den vorherigen und nachfolgenden gesetzt.

Beispiel:

Artikel 1: 0

Artikel 2: 1

Nun wird Artikel 3 zwischen Artikel 1 und 2 geschoben, bekommt dann bspw. Index: 0.5. Das Spiel kann man beliebig oft wiederholen und endet in Kommawerten.

Beim Lernen werden Indexe miteinander verglichen und ähnlich dem Oben neue Positionen ermittelt. Mehr werde ich nicht offenlegen. ;-)

Viel Spaß beim Einkaufen mit pon,

Adrian Kühlewind

adrian@ponlist.de