Dokumentation zur Projektarbeit „Digitale Kaffeekasse“

Jan Müller

21. September 2020

Inhaltsverzeichnis

Einleitung

Die Entwicklung von Android Apps zeichnet sich unter anderem dadurch aus, dass die *API Level*[1] des *Android Frameworks* unterschiedliche Funktionalitäten bereitstellen. Die Nutzerverteilung erstreckt sich über mehrere Versionen [1], weshalb es zur Maximierung der potenziellen Nutzer notwendig ist auch ältere API Level zu unterstützen. Damit Entwickler trotzdem neue Funktionen einheitlich verwenden können, wurden die *Support Libraries* eingeführt [2]. Aus diesen wurde *Android Jetpack* entwickelt [3], welches Thema dieser Projektarbeit ist.

Ziel dieses Projektes war die Entwicklung einer Kaffeekassen App und einer dazugehörigen Serveranwendung mit der Programmiersprache Kotlin. Für die Entwicklung der App sollten Android Jetpack Bibliotheken verwendet werden, um ihre Vorzüge und Eigenschaften kennenzulernen.

Diese Dokumentation stellt Funktionalität, Architektur, verwendete Bibliotheken und Eigenschaften der digitalen Kaffeekassen App sowie der zugehörigen Serveranwendung vor. Anschließend werden Entstehungsprozess beider Programme erläutert, Ergebnisse vorgestellt und ein Fazit zu Android Jetpack gezogen.

[1] Zum Zeitpunkt dieser Ausarbeitung existieren 29 API Level.

Kapitel 1

Android App

Kern dieser Projektarbeit ist die Android App, die als Benutzeroberfläche der digitalen Kaffeekasse dient. Sie basiert auf Grundlagen, welche durch Android Jetpack gelegt wurden. In diesem Kapitel werden Funktionalität, Architektur, verwendete Android Jetpack Komponenten und Bibliotheken sowie Konfiguration der App vorgestellt.

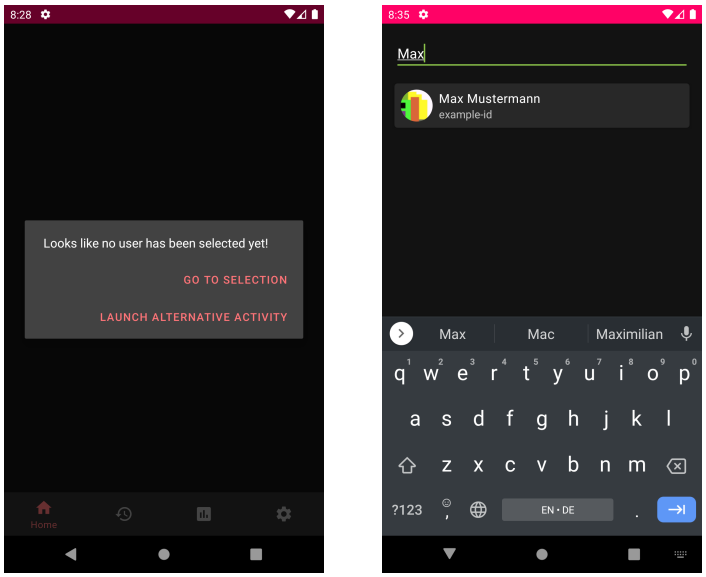
1.1 Funktionalität

Beim erstmaligen Starten der App werden Nutzer gebeten ihren Account auszuwählen. Alternativ können sie den Modus für eine geteilte Verwendung starten, welcher am Ende diese Abschnittes erläutert wird. Der zugehörige Dialog ist in Abbildung 1.1a zu sehen.

Die Nutzerauswahl zeigt eine Liste aller existierenden Nutzer an und ermöglicht es diese Interaktiv zu filtern. Suchergebnisse werden unmittelbar angezeigt und benötigen keine zusätzlichen Eingaben (siehe Abbildung 1.1b). Nachdem ein Nutzer durch Klicken ausgewählt wurde, öffnet sich die Hauptseite der App. Dort werden Informationen des ausgewählten Nutzers und verfügbare Artikel angezeigt. Auch diese lassen sich analog zu Nutzern filtern und durch klicken auswählen; Falls eine stornierbare Transaktion vorliegt wird diese ebenfalls hier angezeigt (siehe Abbildung 1.2a). Durch Klicken auf das Profilbild, beziehungsweise dessen Platzhalter, kann dieses bearbeitet oder entfernt werden.

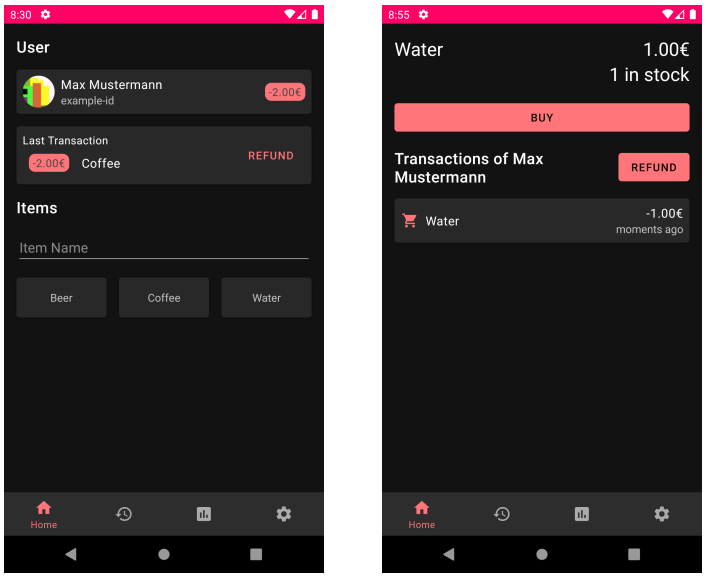
Die Artikelseite zeigt neben Preis und Name des Artikels auch alle Transaktionen des aktiven Nutzers an, die zu einem Artikel gehören. Falls es sich um einen Artikel mit begrenztem Bestand handelt, wird auch dieser angezeigt und bei Transaktionen aktualisiert. Zudem können Stornierungen auch von der Seite des betroffenen Artikels durchgeführt werden, wie in Abbildung 1.2b zu sehen ist. Wie auch bei der Hauptseite, ist die Option nur während des gültigen Zeitrahmens von 60 Sekunden verfügbar und verschwindet selbstständig nach Ablaufen der Zeit.

Eine Übersicht über alle Transaktionen können sich Nutzer mit der Historie verschaffen. Diese ist, wie auch Haupt-, Statistik- und Einstellungsseiten, über eine Navigationsleiste am unteren Bildschirmrand erreichbar. Die Historie zeigt eine vollständige Liste aller Transaktionen, die der ausgewählte Nutzer getätigt hat. Sie sind nach Transakti-



(a) Begrüßungsdialog (b) Nutzerauswahl

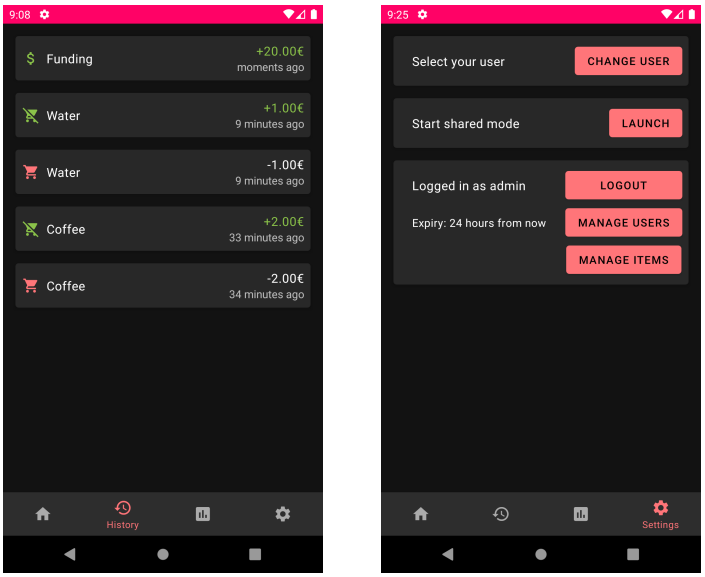
Abbildung 1.1: Begrüßungsdialog und Nutzerauswahl



(a) Hauptseite (b) Artikelseite

Abbildung 1.2: Haupt- und Artikelseite

onszeitpunkten sortiert und geben Art, Wert und weitere Details der Transaktion an, wie in Abbildung 1.3a abgebildet ist. Durch Klicken auf Käufe oder Stornierungen wird zur Seite des zugehörigen Artikels navigiert. Wie bei allen Bildschirmen dieser App ist es zudem möglich die Seite durch eine *Swipe-to-Refresh*-Geste zu aktualisieren. Auf der Statistikseite finden sich Angaben zu Anzahl und letzten Kaufzeitpunkten pro Artikel vom aktiven Nutzer. Auch hier ist es möglich, durch Klicken auf einen Eintrag zur Seite des zugehörigen Artikels zu navigieren.

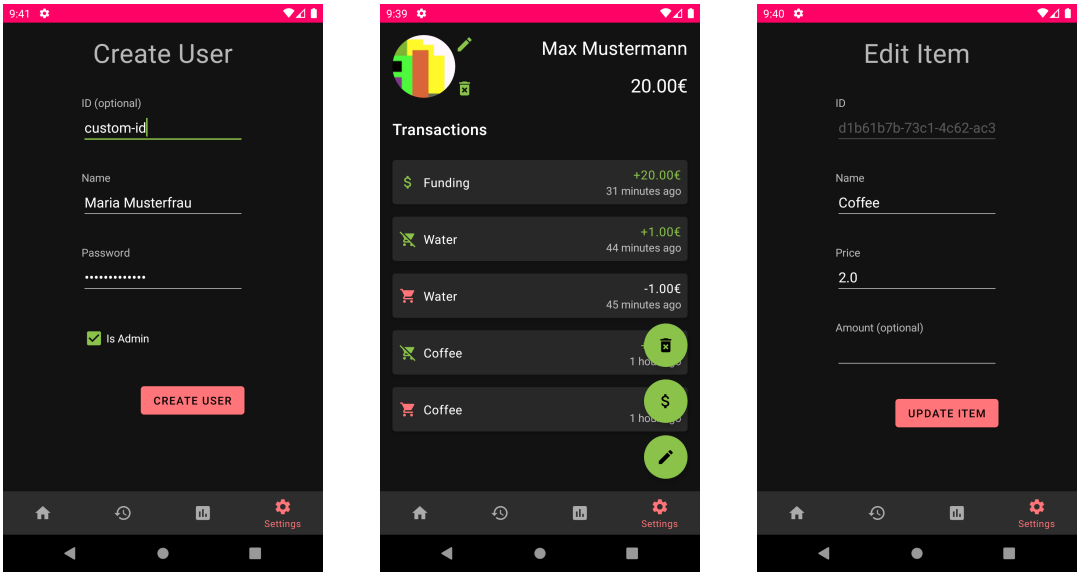


(a) Historie (b) Einstellungen

Abbildung 1.3: Historie und Einstellungen

Die Einstellungen umfassen neben Optionen zur erneuten Nutzerauswahl und Starten des geteilten Modus auch die Administrator-Funktionen (siehe Abbildung 1.3b). Diese sind jedoch erst nach einem Login zugänglich, welcher über eine eigene Seite stattfindet. Die Buttons „MANAGE USERS“ und „MANAGE ITEMS“ öffnen Seiten, welche jeweils denen der Nutzer- und Artikelauswahl entsprechen. Sie verfügen jedoch über zusätzliche Buttons, die Seiten zum Erstellen neuer Nutzer beziehungsweise Artikel öffnen. Abbildung 1.4a zeigt die Nutzervariante dieser Seite. Felder, welche als optional markiert sind, müssen nicht ausgefüllt werden. IDs können beispielsweise von der Serveranwendung vergeben werden. Um einen Nutzer oder Artikel zu bearbeiten muss auf diesen geklickt werden. Während sich bei Artikeln eine erweiterte Ansicht der Artikelseite öffnet, haben Administratoren eine besondere Ansicht für Nutzer (siehe Abbildung 1.4b). Dort sehen sie alle Transaktionen, können Profilbildern bearbeiten sowie löschen und haben die Möglichkeit Guthaben aufzuladen. Letzteres wird über eine eigene Seite durchgeführt. Sowohl bei Nutzern als auch bei Artikeln gibt es Optionen zum Löschen und Bearbeiten der gesamten Entität. Aufbau der Bearbeitungsseiten entspricht denen der Seiten

zum Erstellen, wie in Abbildung 1.4c zu sehen ist. Beim Löschen müssen zwei Dialoge bestätigt werden um sicherzustellen, dass die Aktion bewusst stattfindet.



(a) Nutzer erstellen (b) Administrator-Ansicht (c) Artikel bearbeiten

Abbildung 1.4: Administrator-Funktionen

Der geteilte Modus dient dazu einen schnellen Nutzerwechsel zu ermöglichen. Er umfasst ebenfalls Nutzerauswahl, Haupt- und Artikelseiten. Es ist jedoch nur möglich in dieser Reihenfolge zwischen ihnen zu navigieren, da ausgewählte Nutzer nicht gespeichert werden. Nachdem dieser Modus gestartet wurde, kann er nur durch einen Neustart der App wieder verlassen werden.

Sollten während einer Verwendung der App Fehler auftreten, werden Nutzer über einen Text am unteren Bildschirmrand darüber informiert. Parallel werden Fehlerbehebungen, wie das Entfernen veralteter Daten, automatisch durchgeführt.

1.2 Architektur

Die App verwendet die von Google empfohlene *Model-View-ViewModel* Architektur (*MVVM*) [4]. Bei dieser wird die Anwendung in Datenmodell, Benutzeroberfläche und die sogenannten *ViewModels* aufgeteilt (siehe Abbildung 1.5). MVVM wurde 2005 von Microsoft vorgestellt. Ziel war es, Quelltext zum Verwalten von Benutzeroberflächen zu reduzieren und falls möglich gänzlich zu eliminieren. Essentiell dafür ist das *Binding*-Konzept (siehe Unterabschnitt 1.3.3).

Die Ebene des Datenmodells liegt in Form von *Repositories* vor. Diese kommunizieren über eine Netzwerkschnittstelle mit der Serveranwendung um Daten zu laden oder Aktionen wie Käufe und Logins durchzuführen. So erhaltene Daten werden unmittelbar

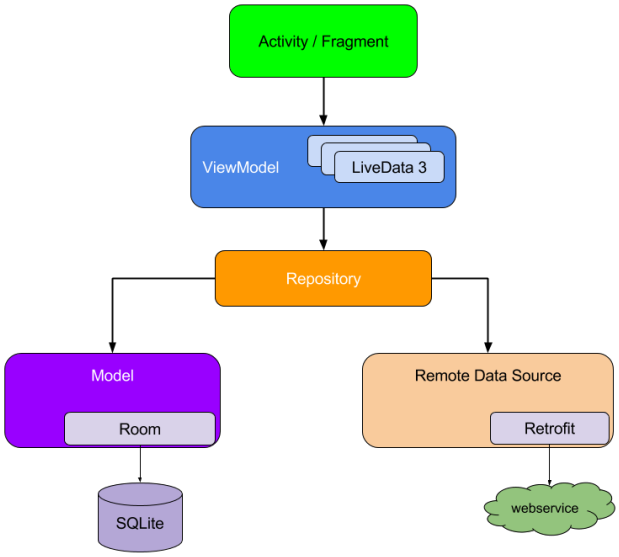


Abbildung 1.5: MVVM-Entwurfsmuster in Android [4]

in die lokale Datenbank geschrieben und auch nur von dort aus geladen. Über diesen Mechanismus wird eine lokale Persistenz ermöglicht, wodurch Daten auch ohne Verbindung zum Server angezeigt werden können. Weil der Zustand eines Datenmodells ausschließlich vom zugehörigen Repository bestimmt wird, werden sie auch als *Single source of truth* bezeichnet. Durch diese Eigenschaft ist es möglich, inkonsistente Datenbestände zu verhindern.

ViewModels passen von Repositories bereitgestellte Daten für Benutzeroberflächen an und stellen sie in Datencontainern bereit. Sie selbst haben keine Informationen über Benutzeroberflächen. Stattdessen reagieren diese auf Änderungen an ViewModels und entscheiden selbst über die Datenanzeige. Bei Android liegen Benutzeroberflächen in Form von Fragmenten und Activities vor.

Die strikte Trennung der Schichten ermöglicht ein Testen der Geschäftslogik, ohne Benutzeroberflächen zu verwenden. Gleichzeitig ist es so möglich Logik unabhängig von der konkreten Benutzeroberfläche wiederzuverwenden.

1.3 Android Jetpack

Android Jetpack umfasst eine Reihe von Bibliotheken, welche dazu dienen, die Entwicklung moderner Android Apps zu vereinfachen. Dazu reduzieren sie beispielsweise den notwendigen *Boilerplate Code* und vereinheitlichen Interaktionen mit dem Android Framework über verschiedene API Level hinweg. Des Weiteren existieren Komponenten, welche bereits umfassende Grundlagen der MVVM-Architektur implementieren oder komplexe Elemente für Benutzeroberflächen bereitstellen.

Vor der Einführung von Jetpack mit Android 9.0 konnten Entwickler auf Support Libraries zurückgreifen. Diese stellten bereits einen kleinen Teil der Jetpack Funktionalität zur Verfügung. Sie unterscheiden sich jedoch dadurch, dass ihre Veröffentlichungen unterschiedliche API Level unterstützten [5]. Dementsprechend konnten keine einheitliche Entwicklung über API Level hinweg ermöglichen. Jetpack behebt dieses Problem, indem auf unterschiedliche Versionen für verschiedene API Level verzichtet wird. Stattdessen wird Kompatibilität von Jetpack selbst sichergestellt, indem Interaktionen mit dem Android Framework verkapselt werden. Die Entwicklungsumgebung Android Studio erstellt neue Projekte standardmäßig mit Jetpack. Projekte, welche Support Libraries verwenden, können automatisch migriert werden, so dass Entwickler schnell und einfach umsteigen können [6].

Zum Zeitpunkt dieser Ausarbeitung existieren 82 Jetpack Bibliotheken, von denen ein Großteil produktionsbereite Veröffentlichungen besitzt [7]. Diese werden kontinuierlich aktualisiert, erweitert und durch neue Bibliotheken ergänzt. Im Folgenden werden die in dieser App verwendeten Android Jetpack Bibliotheken kurz vorgestellt. Detaillierte Informationen können der offiziellen Dokumentation entnommen werden.

1.3.1 Core und AppCompat

Die *Core* und *AppCompat* Bibliotheken ermöglichen eine Verwendung von Funktionen neuer Android API Level auf älteren Android Versionen. Dazu stellen sie eine Vielzahl an Hilfsklassen zur Verfügung, welche die unterschiedliche Behandlung der API Level verkapseln und die Entwicklung so vereinheitlichen und vereinfachen. Die Klasse *AppCompatActivity* erlaubt beispielsweise eine Verwendung von *ActionBars*, welche ohne sie auf älteren Android Versionen nicht möglich wäre. Core enthält hingegen Kompatibilitätskomponenten, welche Benachrichtigungen, Berechtigungen und weitere Interaktionen mit dem Android Framework über API Level hinweg vereinheitlichen.

1.3.2 Fragment

Fragmente sind modulare Teile von Benutzeroberflächen. Sie sind wiederverwendbar, leichtgewichtig und lassen sich in *Activities* sowie anderen Fragmenten kombinieren um komplexe Benutzeroberflächen zu erstellen. Ihr Lebenszyklus wird vom Android Framework verwaltet, weshalb Entwickler für diesen keinen Mehraufwand betreiben müssen. Darüber hinaus ist es möglich, Fragmente dynamisch hinzuzufügen und zu entfernen. Komplexe *Layouts* lassen sich so in mehrere Fragmente unterteilen, welche unabhängig voneinander wiederverwendbar sind.

1.3.3 Databinding

Mithilfe der *Databinding* Bibliothek ist es möglich, Datenquellen für Eigenschaften von Benutzeroberflächen und deren Elemente, welche auch *Views* genannt werden, in Layout-Dateien festzulegen. Dafür werden sogenannte *BindingImplementations* generiert, welche die dafür benötigte Logik bereitstellen und zudem direkte Zugriffe auf Views ermöglichen.

In Kombination mit *LiveData* passen sich Benutzeroberflächen bei Änderungen automatisch an, wodurch der Anteil an Quelltext für Benutzeroberflächen reduziert wird. Zusätzlich lassen sich *BindingAdapters* definieren. Dabei handelt es sich um Funktionen, welche eine View sowie weitere beliebige Parameter erhalten. Sie können in Layout-Dateien verwendet werden um benutzerdefiniertes Verhalten sowie Eigenschaften zu definieren. Zum Beispiel kann ein BindingAdapter definiert werden, welcher in Abhängigkeit eines Parameters Views aus- und einblendet.

1.3.4 Lifecycle

Die *Lifecycle* Bibliothek umfasst primär ViewModels und LiveData, aber auch allgemeine Komponenten zum Verwalten und Implementieren von Lebenszyklen.

Jetpack ViewModels werden vom Android Framework verwaltet. Sie besitzen einen Lebenszyklus, welcher sich Fragmenten anpasst, die ihre Instanzen verwenden. So kann sichergestellt werden, dass sie nach einer Zerstörung zugehöriger Fragmente korrekt beendet werden.

LiveData baut auf diesem Konzept auf. Dabei handelt es sich um *Wrapper* für beliebige Daten, deren Werte beobachtet werden können. Anders als traditionelle *Observables* besitzt LiveData jedoch einen Lebenszyklus. Sobald keine *Oberserver* mehr registriert sind, da deren Lebenszyklus beendet wurde, propagiert LiveData keine Änderungen mehr. Entwickler müssen deshalb keine Referenzen entfernen oder auf andere Art und Weise sicherstellen, dass ihre Observables korrekt terminiert werden. LiveData übernimmt diesen Aspekt, welcher bei inkorrekter Behandlung leicht zu *Memory Leaks* führen kann.

1.3.5 Navigation

Fragmente können mit Argumenten erzeugt werden. Dabei ist jedoch keine Typsicherheit garantiert und es gibt keinen Mechanismus, der die Vollständigkeit von Argumenten sicherstellt. Die *Navigation* Bibliothek löst beide Probleme durch Navigationsgraphen. In diesen werden verfügbare Fragmente und ihre Argumente festgelegt. Durch Verbindungen zwischen Fragmenten werden *Directions* definiert, welche Navigationsrichtungen symbolisieren. Deren Verwendung bei der Navigation stellt sicher, dass alle Argumente mit korrektem Typ vorhanden sind und zum korrekten Fragment navigiert wird. Zudem kann über Directions das Verhalten einer Navigation festgelegt werden. Darunter fallen beispielsweise Übergangsanimationen und ein Aufräumen des *Android Backstacks*[1]. Darüber hinaus unterstützt die Bibliothek Navigationscontroller, die das Initialisieren von Benutzerelementen wie *Bottom Navigations*, *Hamburger Menus* und *Toolbars* übernehmen.

[1] Siehe `https://developer.android.com/guide/components/activities/tasks-and-back-stack`.

1.3.6 ConstraintLayout und SwiperefreshLayout

Android Jetpack umfasst auch mehrere Layout-Bibliotheken. Im Rahmen dieses Projektes wurden *ConstraintLayout* und *SwiperefreshLayout* verwendet.

Komplexe Hierarchien in Benutzeroberflächen verringern deren Performanz bei Layoutänderungen. ConstraintLayout ermöglicht ein freies Platzieren von Views, durch relative Angaben zur Einschränkung ihrer Positionen. Dies funktioniert, da Einschränkungen relativ zum Container als auch anderen Views angegeben werden können. Auf Grund der flachen Hierarchie von ConstraintLayouts ist ihre Performanz deutlich besser als die alternativer Layouts [8]. Gleichzeitig können sich so erstelle Layouts an unterschiedliche Bildschirmgrößen und -orientierungen anpassen, wodurch Benutzeroberflächen flexibler entwickelt werden können.

Bei mobilen Anwendungen gibt es die gängige *Swipe-to-Refresh*-Geste zum Aktualisieren eines Bildschirms. Mithilfe von SwiperefreshLayout ist es möglich, diese Funktionalität mit wenigen Zeilen Quelltext zu bestehenden Benutzeroberflächen hinzuzufügen. SwiperefreshLayout verfügt über die von vielen Google Apps verwendete Spinner Animation und ist mit scrollbaren Views und Layouts kompatibel.

1.3.7 RecyclerView

RecyclerViews ermöglichen ein Anzeigen von langen Listen, ohne dass Speicherverbrauch oder Performanz leiden. Dies ist möglich, indem nicht mehr Container für Listenelemente erzeugt werden, als gleichzeitig darstellbar sind. Zudem werden sie für andere Listenelemente wiederverwendet, sobald sie nicht mehr sichtbar sind. Durch dieses Konzept sind RecyclerViews performanter als beispielsweise *ListViews* oder *LinearLayouts*. Zur Interaktion mit RecyclerViews werden Adapter verwendet, welche Erzeugen von Container und Zuweise von Elementen definieren. Es stehen Hilfsklassen bereit, die bereits über einen Großteil der benötigten Logik verfügen. *ListAdapter* ermöglicht beispielsweise eine direkte Übergabe von Listen. Um Änderungen an Listeneinträgen zu erkennen werden in der Regel *DiffCallbacks* verwendet. Diese sind in der Bibliothek enthalten und verwenden komplexe Algorithmen um Änderungen an Listen darzustellen, so dass nur betroffene Container aktualisiert werden müssen.

1.3.8 Room

Bei *Room* handelt es sich um eine Abstraktionsebene für *SQLite* Datenbanken. Datenbankinteraktionen werden dabei in *Data Access Objects*, auch *DAOs* genannt, über Annotationen an Funktionen definiert. Bei DAOs handelt es sich um *Interfaces*, deren Implementierungen von Room generiert werden. Komplexe SQL-Anfragen können ebenfalls in Textform angegeben werden. Die Hauptfunktion von Room liegt jedoch in der Möglichkeit Rückgabewerte in reaktiven Wrapper-Klassen, wie beispielsweise LiveData, zu erhalten. Room übernimmt in diesem Fall die Aktualisierung der Daten, wodurch es reicht Datenbankanfragen einmalig auszuführen. Darüber hinaus können Datenbanktransaktionen programmatisch definiert und Konfliktstrategien an Voraussetzungen angepasst

werden. Room unterstützt zudem benutzerdefinierte Migrationsstrategien, wodurch Aktualisierungen von Datenbankschemas erleichtert werden.

1.3.9 Paging

Das Laden großer Datenmengen aus Datenbanken oder Netzwerkquellen ist ein nicht zu unterschätzender Aufwand. Die *Paging* Bibliothek ermöglicht deshalb ein stückweises Laden von Datensätzen. Dazu muss eine *DataSource Factory* vorliegen, welche dieses Konzept unterstützt. Unter anderem ermöglicht Room solche *Factories* als Rückgabewerte zu definieren. Sie lassen sich zu LiveData Objekten transformieren, welche *PagedLists* enthalten. Diese besonderen Listen lassen sich unter anderem mit RecyclerViews und *PagedListAdaptern* verwenden. Im Endeffekt können so Performanz und Speicherverbrauch verbessert werden.

1.3.10 Work

Hintergrundaufgaben, wie beispielsweise das Bereinigen von Datenbanken, können über die *Work* Bibliothek verwaltet werden. Die zu erledigende Arbeit wird in *Worker*-Klassen definiert, welche optional Coroutines verwenden können um vollständig asynchron ausgeführt zu werden. Beim Registrieren von Worker-Instanzen können Einschränkungen definiert werden, welche Zustände, Zeitpunkte und Intervalle von Ausführungen festlegen. So kann sichergestellt werden, dass beispielsweise eine Internetverbindung besteht oder Akkus von Endgeräten nicht übermäßig stark belastet werden. Im Endeffekt erlaubt die Bibliothek so eine benutzerfreundliche Ausführung von Aufgaben, welche im Hintergrund stattfinden sollen.

1.3.11 KTX

Kotlin ist die von Google empfohlene Sprache für Android-Entwicklung [9]. Sie zeichnet sich unter anderem dadurch aus, dass sie im Vergleich zu Java deutlich weniger verbos und in der Regel entsprechend kürzer ist. Eine Vielzahl an Android Jetpack Bibliotheken liegen in *KTX*-Varianten vor. Diese enthalten zusätzliche Erweiterungsmethoden und Sprachfunktionen, welche die Programmierung von Android Apps mit Kotlin vereinfachen. Dazu zählen beispielsweise Erweiterungsmethoden, welche den Lesefluss von Aufrufen verbessern oder mehrere Methodenaufrufe zusammenfassen. Des Weiteren existieren zahlreiche Methoden zum Erzeugen von ViewModels und LiveData, welche einen Großteil der Konfiguration verkapseln. Auch Kotlin-spezifische Sprachkonstrukte wie *Delegated Properties* und *Coroutines* werden von Jetpack unterstützt, um die Entwicklung mit Kotlin weiter zu verbessern.

1.4 Weitere Bibliotheken

Im Rahmen der App-Entwicklung wurden auch Bibliotheken verwendet, welche nicht Teil von Android Jetpack sind. Im Folgenden werden ihre Funktionen und Verwendungsgrün-

de vorgestellt.

1.4.1 Retrofit und Moshi

Über *Retrofit* können abstrakte Methoden zum Anbinden von APIs definiert werden. Entwickler müssen dazu Informationen über die Art von Anfragen angeben. Implementierungen dieser Methoden werden von der Bibliothek generiert. In Kombination mit der Serialisierungsbibliothek *Moshi* können so Netzwerkschnittstellen mit geringem Quelltextaufwand angebunden werden.

1.4.2 Glide

Glide übernimmt das asynchrone Laden und Darstellen von Bildern. Dabei lassen sich Einstellungen für Verhalten und Darstellung anpassen. Unter anderem kann Glide Bilder für schnelleres Laden zwischenspeichern, zuschneiden und während einem Ladevorgang durch Platzhalter ersetzen.

1.4.3 Image Picker

Die Bibliothek *Image Picker* ermöglicht die Auswahl von Bildern zur Laufzeit, die Nutzer entweder aus ihrer Galerie auswählen oder mit ihrer Kamera aufnehmen können. Eine äquivalente Funktionalität zu implementieren ist mit nicht-trivialem Aufwand verbunden, weshalb eine Verwendung existierender Bibliotheken die Entwicklung verkürzte.

1.4.4 PrettyTime

PrettyTime wird verwendet um Zeitstempel in ein für Menschen lesbares Format zu transformieren. Dabei werden keine Daten, sondern Angaben wie „9 minutes ago“ und „moments ago“, verwendet. Im Kontext des Verwendungszwecks der App sind eher kürzlich getätigte Transaktionen relevant. Durch das Format der Zeitstempel ist es einfacher diese schnell zu erkennen. Die Bibliothek unterstützt zudem mehrere Sprachen, unter anderem Deutsch sowie Englisch, und erspart Entwicklern so das Übersetzen.

1.4.5 Timber

Bei *Timber* handelt es sich um eine Bibliothek die das Protokollieren vereinfacht, Nachrichten mit zusätzlichen Informationen wie beispielsweise ihrem Ursprung ausgibt und jegliche Protokollierung außerhalb der Entwicklungsphase deaktiviert. So ist es während der Entwicklung möglich schnell detaillierte Protokollnachrichten zu verfassen, welche bei der Suche nach Fehlern helfen und keine Auswirkung auf ausgelieferte Endprodukte haben.

1.5 Konfiguration

Zur Konfiguration muss die Adresse der Serveranwendung angepasst werden. Dies findet über den Wert des Feldes `KOFFEE_BACKEND_URL`, in der Datei `build.gradle`, statt. Eine weitere Konfiguration ist nicht notwendig.

Kapitel 2

Serveranwendung

Das Backend wurde, wie auch die App, mit der Programmiersprache Kotlin entwickelt. Es handelt sich dabei um eine *REST*-Schnittstelle[1], die dazu dient den Zustand der digitalen Kaffeekasse zu verwalten. Die folgenden Abschnitte stellen Funktionalität, Architektur, Authentifizierung, verwendete Bibliotheken sowie Konfiguration und Inbetriebnahme des Backends vor.

2.1 Funktionalität

Alle Funktionen des Backends sind über eine Vielzahl von Endpunkten verwendbar (siehe Anhang D). Nach erfolgreicher Authentifizierung können Administratoren Nutzer sowie Artikel erstellen, aktualisieren und löschen. Zudem können Administratoren Guthabenaufladungen durchführen. Ohne Authentifizierung ist es möglich, Artikel zu kaufen und Käufe innerhalb einer Minute zu stornieren. Zudem können Nutzer Profilbilder hochladen und löschen.

2.2 Architektur

Das Backend verwendet eine serviceorientierte Architektur. Dabei werden Anfragen an Dienste weitergeleitet. Antworten von Diensten enthalten Informationen über die Ergebnisse von Aktionen, wie Statuscodes, Daten oder Fehlermeldungen, und werden an Kommunikationspartner weitergeleitet.

Es existieren Dienste für Nutzer-, Profilbild-, Artikel- und Transaktionsverwaltung. Validierung sowie Fehlerbehandlung finden ausschließlich in Diensten statt. Dafür verwenden sie einen funktionalen Ansatz und sind selbst zustandslos.

Analog zur Android App liegt die Persistenzebene in Form von Repositories vor. Die gegebenen Implementierungen der Nutzer-, Profilbild- und Artikel-Repositories fungieren als Schnittstellen zur verwendeten *MongoDB*.

[1] Siehe `https://restfulapi.net/`.

Sowohl Dienste als auch Repositories liegen als Interfaces und Implementierungen vor, um separates Testen zu ermöglichen.

2.3 Authentifizierung

Um Administrator-Funktionen zu verwenden müssen sich Nutzer authentifizieren. Dies findet mithilfe von *JSON Web Tokens*[2], auch *JWT* genannt, statt. Diese bestehen aus Header, Payload und Signatur. Die Teile enthalten Informationen über den zur Signierung verwendeten Algorithmus, (Meta-)Daten sowie die errechnete Signatur. Letztere kann zum Verifizieren von JWTs verwendet werden. So ist es möglich sicherzustellen, dass Nachrichten vom erwarteten Absender stammen und nicht manipuliert wurden.

Das Backend generiert JWTs nach erfolgreichen Logins. Sie sind für 24 Stunden gültig und müssen bei jeder Anfrage für Administrator-Funktionen enthalten sein. Einloggen können sich nur Nutzer, die über Administratorrechte verfügen, welche wiederum nur von Administratoren vergeben werden können. Bei jedem Start generiert das Backend einen Standardadministrator, welcher zuvor festgelegte Daten besitzt (siehe Abschnitt 2.6).

2.4 Datenbank

Bei der MongoDB des Backends handelt es sich um eine dokumentenbasierte Datenbank. Die von ihr verwendete Schemas entsprechen deshalb denen der im Quelltext definierten Klassen (siehe Anhang C). Falls die über *Docker Compose* vorkonfigurierte MongoDB verwendet wird, lassen sich Daten mithilfe zweier Kommandozeilenbefehle sichern und wiederherstellen (siehe Anhang B).

2.5 Bibliotheken

Um die Entwicklung des Backends zu beschleunigen wurden mehrere Open Source Bibliotheken verwendet. Diese werden im Folgenden kurz vorgestellt. Zusätzliche Informationen können den jeweiligen Projektseiten entnommen werden.

2.5.1 Ktor

Ktor, ein *Web Framework* vom Entwickler der Sprache Kotlin, bildet die Grundlage des Backends. Ktor verwendet eine *Domain Specific Language*, auch *DSL* genannt, zum Definieren von Modulen. Solche Module können in der Serveranwendung installiert werden um ein erwünschtes Verhalten zu erreichen. Ktor selbst stellt dabei Module für verschiedene Bereiche, wie unter anderem Protokollierung, Serialisierung und Authentifizierung, zur Verfügung. Auch vordefinierte Module können im Quelltext angepasst werden. Ktor zeichnet sich zudem dadurch aus, dass es vollständig asynchron arbeitet und Kotlin Coroutines an allen Stellen der Konfiguration verwendbar sind. Dadurch können Entwickler

[2] Siehe `https://jwt.io/`.

asynchronen und performanten Quelltext schreiben, ohne zusätzliche Sprachkonstrukte zu verwenden.

2.5.2 Koin

Koin ist ein Framework für *Dependency Injection*, mit dem einzelne Komponenten des Servers unabhängig von ihrem Einsatzort initialisiert werden können. So ist es möglich Instanzen von Diensten sowie Repositories wiederzuverwenden und den für Instanziierung benötigten Quelltext zu minimieren.

2.5.3 KMongo

Die *KMongo* Bibliothek ermöglicht eine simple Verwendung von *Mongo*-Datenbanken in Kotlin. Dies erreicht sie durch automatische Serialisierung, Unterstützung für Kotlin Coroutines und programmatische, typsichere Anfragen.

2.5.4 jBCrypt

Das Backend muss Passwörter sicher in der Datenbank ablegen. Dafür müssen diese mit einer geeigneten *Hash*-Funktion transformiert werden. Die Bibliothek *jBCrypt* implementiert die bewährte *bcrypt*-Hash-Funktionen und stellt zudem einen *Salt*-Generator bereit. bcrypt setzt auf einen skalierenden Berechnungsaufwand, um das Entschlüsseln von Passwörtern möglichst aufwändig zu gestalten. Es gilt deshalb als sicher und für die Zukunft gewappnet [10]. Darüber hinaus ermöglicht jBCrypt auch das Überprüfen von Passwörtern während einer Authentifizierung.

2.5.5 Arkenv

Zur Konfiguration des Backends an eine Einsatzumgebung müssen einige Parameter, wie die Adresse der Datenbank, gesetzt werden. *Arkenv* erlaubt es diese Umgebungsvariablen programmatisch und typsicher zu definieren, so dass die Fehleranfälligkeit beim Einlesen reduziert wird.

2.5.6 Docker Secrets

Neben Umgebungsvariablen müssen auch sensible Daten, wie die Zugangsinformationen des Standardadministrators, festgelegt werden. Um diese zu schützen werden beim *Deployment* des Backends mit Docker sogenannte *Secrets* verwendet, welche nur das Backend selbst einsehen kann. Da es sich dabei um Dateien handelt ist das Einlesen ihrer Informationen mit einer Syntaxanalyse verbunden. Die leichtgewichtige Bibliothek *Docker Secrets* übernimmt diesen Aufwand und lädt Parameter von Dateien in *Maps*.

2.6 Konfiguration und Inbetriebnahme

Um das Backend an Einsatzumgebungen anzupassen sollten einige Parameter konfiguriert werden. Dies findet über `.env`-Dateien statt, welche sich in dem `environments`-Ordner des Hauptverzeichnisses befinden. Dort werden die Adressen von Server und Datenbank sowie Eigenschaften der Zertifizierung festgelegt. Darüber hinaus wird der Namen des Docker Secrets angegeben, in welchem sensible Daten hinterlegt werden. Bei der Standardkonfiguration muss sich diese Datei im `secrets`-Verzeichnis befinden. `koffee.secret` enthält den zur Signierung von JWTs verwendeten Schlüssel sowie Informationen des Standardadministratoren. Das folgende Beispiel zeigt eine mögliche Konfiguration dieser Datei.

```
1 ID=admin
2 NAME=Admin
3 PASSWORD=adminPassword
4 HMAC_SECRET=geheimeZeichenkette
```

Beispielinhalt von /secrets/koffee.secret

Um das Backend in Betrieb zu nehmen müssen zuerst der `URL`-Parameter in der Datei `domain.env` angepasst und die `.secret`-Datei angelegt werden. Nachdem das Projekt mit dem Befehl `./gradlew build` erzeugt wurde, kann die Serveranwendung über Docker Compose gestartet werden. Schrittweise Anleitungen zur Inbetriebnahme befinden sich im Anhang (siehe Anhang A).

Kapitel 3

Projektverlauf

In diesem Kapitel werden zunächst die Entstehungsprozesse von Backend und App detailliert. Dabei wird das Vorgehen begründet und die Programmiermethodik kurz beschrieben. Anschließend werden die Ergebnisse der Projektarbeit zusammengefasst.

3.1 Entstehungsprozess

Die Projektarbeit begann mit der Entwicklung des Backends. Zunächst wurden Aufbau sowie Struktur festgelegt und implementiert um eine Arbeitsgrundlage zu schaffen. Dank der modularen Zusammensetzung des Backends war es möglich die benötigten Endpunkte für Nutzer, Artikel sowie Transaktionen schnell und unabhängig voneinander zu implementieren. Anschließend wurde die Authentifizierung hinzugefügt um Endpunkte vor unautorisierten Zugriffen zu schützen. Da eine langfristige Nutzung des Backends in Frage kam, wurde der Quelltext nahezu vollständig getestet. Tests wurden parallel zum eigentlich Programm geschrieben um Fehler schnell zu erkennen. Im Laufe der Entwicklung wurde das Backend mehrfach überarbeitet um einen möglichst funktionalen Programmierstil zu erreichen. So konnten Lesbarkeit des Quelltextes verbessert und Fehleranfälligkeit reduziert werden. Während der gesamten Projektarbeit wurde zudem die Formatierung des Quelltextes mit dem Tool *klint*[1] kontrolliert und korrigiert. Darüber hinaus wurden Open Source Bibliotheken verwendet, falls Problemlösungen einen größeren Aufwand erforderten. Dadurch konnte die Entwicklungszeit verkürzt und auf die eigentliche Aufgabenstellung fokussiert werden.

Die Entwicklung der App begann Ende April. Analog zu den Endpunkten des Backends wurden die Bildschirme der App nacheinander implementiert, wobei jeweils mehrere Revisionen durchgeführt wurden. Eine Verwendung der Android Jetpack Bibliotheken erlaubten dabei eine schnelle Entwicklung mit einem geringen Anteil an Boilerplate Code. Um diesen weiter zu reduzieren wurden im Laufe der Entwicklung häufig auftretende Muster, wie beispielsweise von ViewModels initiierte Aktionen, in Klassen ausgelagert. Auch bei den ViewModels selbst sowie Fragmenten wurde Vererbung ver-

[1] Siehe `https://github.com/pinterest/ktlint`.

wendet um oft benötigtes Verhalten nicht mehrfach zu implementieren. Zudem wurde die Fehlerbehandlung zentralisiert um die Fehlerausgabe zu vereinfachen und Abstürze zu verhindern. Durch solche Änderungen war es möglich den Quelltext deutlich zu kürzen.

Zunächst wurde die App für einen Nutzer pro Endgerät entwickelt. Anfang Juni wurde, nach Anfrage des Projektbetreuers, ein zusätzlicher Modus für mehrere Nutzer implementiert. Weil die einzelnen Komponenten der App bereits modular und wiederverwendbar waren, konnte diese größere Änderung schnell umgesetzt werden. Im Anschluss wurden ein Großteil der Benutzeroberflächen sowie deren Verwendung überarbeitet und verbessert.

Nachdem die Funktionalität beider Teilprojekte vollständig implementiert war, wurde mit der Dokumentation des Quelltextes begonnen, welche im Internet abrufbar ist[2].

3.2 Ergebnisse

Alle Grundfunktionen des Kaffeekassensystem wurden sowohl in der App als auch im Backend umgesetzt. Dazu gehören die administrative Verwaltung von Artikeln und Nutzern sowie Kaufen und Stornieren.

Nach Wünschen der Projektbetreuer wurden Benutzeroberflächen umgebaut und die Verwendung der App angepasst. Zusätzlich wurde ein alternativer Modus für eine geteilte Verwendung durch mehrere Nutzer in der App implementiert.

Für das Backend wurde zusätzlich eine Konfiguration erstellt, die einen sicheren Betrieb über HTTPS ermöglicht und diesen selbstständig einrichtet. Des Weiteren resultiert die Modulare Struktur des Backends in einer einfachen Erweiterbarkeit, falls ein Hinzufügen neuer Funktionalitäten erwünscht ist. Die vollständige Dokumentation des Quelltexts und das Einhalten von Formatierungskonventionen unterstützen diese Eigenschaft weiter.

[2] Siehe `https://koffee.yeger.eu`.

Kapitel 4

Fazit

Die Vorzüge der Android Jetpack Bibliotheken zeigten sich bereits während der Entwicklung. Mit Fragmenten und ViewModels war es möglich neue Bildschirme schnell zu implementieren. Zudem ermöglichten Android Jetpacks visuelle Navigationsgraphen ein einfaches und übersichtliches Verknüpfen von Fragmenten. Der dadurch gewonnene Gesamtüberblick resultierte in einer sicheren und fehlerresistenten Struktur. Der automatisierte Lebenszyklus vieler Jetpack Komponenten erlaubte zudem eine Minimierung an Interaktionen mit dem Android Framework. Stattdessen war es möglich den Fokus auf das Verhalten von Benutzeroberflächen und Geschäftslogik zu legen. Des Weiteren hat sich Room als eine der wichtigsten Komponenten von Android Jetpack erwiesen. Bei der Entwicklung trivialisierte sie Datenbankzugriffe und reduzierte Datenverwaltung auf ein Minimum. Sichtbar wurden diese Vorzüge besonders bei der Entwicklungszeit. Meilensteine wurden meist Wochen zuvor erreicht, wodurch zusätzliche Funktionen implementiert werden konnten.

Auch bei Verwendung der App sind Vorzüge der Jetpack Bibliotheken erkennbar. Die Asynchronität der Kombination von Room, LiveData und ViewModels stellt sicher, dass Datenbankzugriffe nicht zu einer Blockierung der Benutzeroberfläche führen. RecyclerViews sorgen zudem für eine performante Darstellung von Listen und ermöglichen flüssiges Scrollen durch diese. Die Paging Bibliothek begünstigt dies weiter. Sie sorgt dafür, dass das Laden von Listen aus der Datenbank auch bei vielen Einträgen nicht langsamer wird. Lifecycle Komponenten sorgen zudem dafür, dass sie nicht länger als benötigt aktiv bleiben, wodurch Speicher- sowie Prozessorverbrauch reduziert werden.

Mit Android Jetpack Bibliotheken entwickelte Apps sind zwar nicht zwingend schneller oder besser als Herkömmliche, jedoch ist es für Entwickler einfacher diese Ziele zu erreichen, da Grundbausteine und Architektur bereits gegeben sind. Weil sie zugleich eine beschleunigte Entwicklung ermöglichen und die Zukunft der von Google entwickelten Android Bibliotheken bilden, ist es empfehlenswert bei der App Entwicklung auf Jetpack Komponenten zurückzugreifen.

Referenzen

[1] Bidouille.org. *Android version distribution history*. URL: https://www.bidouille.org/misc/androidcharts (besucht am 16. 07. 2020).

[2] Google Developers. *Support Library*. URL: https://developer.android.com/topic/libraries/support-library (besucht am 16. 07. 2020).

[3] Google Developers. *AndroidX Overview*. URL: https://developer.android.com/jetpack/androidx (besucht am 16. 07. 2020).

[4] Google Developers. *Guide to app architecture*. URL: https://developer.android.com/jetpack/guide (besucht am 16. 07. 2020).

[5] Google Developers. *Version Support and Package Names*. URL: https://developer.android.com/topic/libraries/support-library#api-versions (besucht am 23. 07. 2020).

[6] Google Developers. *Using androidx libraries in your project*. URL: https://developer.android.com/jetpack/androidx#using_androidx_libraries_in_your_project (besucht am 23. 07. 2020).

[7] Google Developers. *Jetpack libraries*. URL: https://developer.android.com/jetpack/androidx/versions#version-table (besucht am 23. 07. 2020).

[8] Google Developers. *Manage complexity: layouts matter*. URL: https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies#managing (besucht am 25. 07. 2020).

[9] Google Developers. *Develop Android apps with Kotlin*. URL: https://developer.android.com/kotlin (besucht am 16. 07. 2020).

[10] Dan Arias. *Hashing in Action: Understanding bcrypt*. URL: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/ (besucht am 25. 08. 2020).

Dokumentationen

aPureBase. *Arkenv*. URL: https://apurebase.gitlab.io/arkenv/ (besucht am 16. 07. 2020).

Bump Technologies. *About Glide*. URL: https://bumptech.github.io/glide/ (besucht am 16. 07. 2020).

Cars.com. *Docker Secrets*. URL: https://github.com/carsdotcom/docker-secrets-java (besucht am 16. 07. 2020).

Google Developers. *Explore the Jetpack libraries*. URL: https://developer.android.com/jetpack/androidx/explorer (besucht am 16. 07. 2020).

InsertKoin.io. *Koin*. URL: https://insert-koin.io/ (besucht am 16. 07. 2020).

JetBrains. *Ktor*. URL: https://ktor.io/ (besucht am 16. 07. 2020).

Litote. *KMongo - a Kotlin toolkit for Mongo*. URL: https://litote.org/kmongo/ (besucht am 16. 07. 2020).

Damien Miller. *jBCrypt*. URL: https://www.mindrot.org/projects/jBCrypt/ (besucht am 16. 07. 2020).

ocpsoft. *prettytime*. URL: https://www.ocpsoft.org/prettytime/ (besucht am 16. 07. 2020).

Dhaval Patel. *Image Picker Library for Android*. URL: https://github.com/Dhaval2404/ImagePicker (besucht am 16. 07. 2020).

Square. *Moshi*. URL: https://github.com/square/moshi (besucht am 16. 07. 2020).

Square. *Retrofit*. URL: https://square.github.io/retrofit/ (besucht am 16. 07. 2020).

Jake Wharton. *Timber*. URL: https://github.com/JakeWharton/timber (besucht am 16. 07. 2020).

Anhang

Anhang A

Inbetriebnahme der Serveranwendung

A.1 Einsatz während der Entwicklung

1. `./gradlew build`
2. Das benötigte Docker Secret anlegen (siehe Abschnitt 2.6).
3. `docker-compose up --build -d`
4. Der Server ist nun unter `http://localhost:8080` erreichbar.

A.2 Einsatz in der Produktion

1. `./gradlew build`
2. Das benötigte Docker Secret anlegen (siehe Abschnitt 2.6).
3. Den Wert von `URL` in `./environments/domain.env` auf eine valide Domäne ändern, über welche die Host-Maschine erreichbar ist.
4. Sicherstellen, dass die Host-Maschine über die Ports 80 und 443 erreichbar ist.
5. `docker-compose -f docker-compose-prod.yml up --build -d`
6. Der Server ist nun unter `https://your.domain/koffee` erreichbar.

Anhang B

Datenbankverwaltung

B.1 Datensicherung

```
$ docker-compose exec -T mongo mongodump --archive --gzip --db
    koffee-database > dump.gz
```

B.2 Datenwiederherstellung

```
$ docker-compose exec -T mongo mongorestore --archive --gzip <
    dump.gz
```

Anhang C

Datenbankschemas

C.1 User

Feld	Typ	Anmerkung
id	String	Muss einzigartig sein
name	String	
isAdmin	Boolean	Erlaubt Authentifizierung
password	String	Optional
transactions	List<Transaction>	Verwendet Polymorphie
profileImage	ProfileImage	Optional

C.2 ProfileImage

Feld	Typ	Anmerkung
encodedImage	String	Base64-Kodierung
timestamp	Long	

C.3 Item

Feld	Typ	Anmerkung
id	String	Muss einzigartig sein
name	String	
amount	Int	Optional
price	Double	

C.4 Transaction - Funding

Feld	Typ	Anmerkung
type	String	Muss „funding“ sein
value	Double	Wert der Guthabenaufladung
timestamp	Long	

C.5 Transaction - Purchase

Feld	Typ	Anmerkung
type	String	Muss „purchase“ sein
value	Double	Gesamtwert der Transaktion
timestamp	Long	
itemId	String	
itemName	String	
amount	Int	

C.6 Transaction - Refund

Feld	Typ	Anmerkung
type	String	Muss „refund“ sein
value	Double	Gesamtwert der Transaktion
timestamp	Long	
itemId	String	
itemName	String	
amount	Int	

Anhang D

Endpunkte

Alle Endpunkte verwenden JSON um Daten zu senden und erhalten. Endpunkte, welche eine zuvorige Authentifizierung voraussetzen, sind mit einem Sternchen (*) gekennzeichnet. Es werden keine *Query*-Parameter verwendet. Felder eventuell benötigter *Body*-Parameter sind unter den jeweiligen Endpunkten aufgelistet.

POST /login Authentifiziert einen Nutzer und gibt im Erfolgsfall einen JWT zurück.

- id: String
- password: String

GET /users Gibt IDs und Namen aller Nutzer zurück.

***POST /users** Erstellt einen neuen Nutzer und gibt dessen ID zurück.

- id: String (Optional)
- name: String
- isAdmin: Boolean
- password: String (Optional)

***PUT /users** Aktualisiert einen existierenden Nutzer.

- id: String
- name: String
- isAdmin: Boolean
- password: String (Optional)

GET /users/:id Gibt ID, Name und Guthaben des Nutzers mit der gegebenen ID zurück.

***DELETE /users/:id** Löscht den Nutzer mit der gegebenen ID.

***POST /users/:id/funding** Lädt das Guthaben des Nutzers mit der gegebenen ID um den gewünschten Betrag auf.

- amount: Double

POST /users/:id/purchases Kauft einen Artikel für den Nutzer mit der gegebenen ID.

- itemId: String
- amount: Int

POST /users/:id/purchases/refund Falls möglich wird der letzte Kauf des Nutzers mit der gegebenen ID storniert.

GET /users/:id/image Gibt Bytes des Profilbilds eines Nutzers mit der gegebenen ID zurück.

GET /users/:id/image/timestamp Antwortet mit dem Zeitstempel des Profilbilds eines Nutzers mit der gegebenen ID.

POST /users/:id/image Speichert die hochgeladene Datei als Profilbild des Nutzers mit der gegebenen ID.

- Multipart Datei (kein JSON)

DELETE /users/:id/image Löscht das Profilbilds des Nutzers mit der gegebenen ID.

GET /items Gibt IDs, Namen, Preise und Mengenangaben aller Artikel zurück.

***POST /items** Erstellt einen neuen Artikel und gibt dessen ID zurück.

- id: String (Optional)
- name: String
- amount: Int (Optional)
- price: Double

***PUT /items** Aktualisiert einen existierenden Artikel.

- id: String
- name: String
- amount: Int (Optional)
- price: Double

GET /items/:id Gibt ID, Name, Preis und Mengenangabe des Artikels mit der gegebenen ID zurück.

***DELETE /users/:id** Löscht den Artikel mit der gegebenen ID.