FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen
Campus Jülich

Prototyp einer mobilen Applikation zum Kontrollieren von Account-Based Tickets in der IVU.Suite

Seminararbeit
im Studiengang Angewandte Mathematik und Informatik
von Jonathan Liersch
Matrikelnummer: 3619788

Betreut durch:

Prof. Dr.rer.nat. Alexander Voß
M. Sc. Christoph Becker

Fachbereich 09
Medizintechnik und Technomathematik

Aachen, 15. Dezember 2025

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Prototyp einer mobilen Application zum Kontrollieren von Account-Based Tickets in der IVU.Suite

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Liersch, Jonathan

Aachen, den 15.12.2025

Unterschrift der Studentin / des Studenten

J. Liersch

Zusammenfassung

Diese Arbeit behandelt die Konzeption und prototypische Umsetzung einer mobilen Applikation zur Kontrolle von Account-Based Tickets (ABT) im Kontext der IVU.Suite. ABT ist ein modernes Ticketing-Verfahren, bei dem Fahrberechtigungen nicht mehr lokal auf einem Medium gespeichert, sondern in einem Hintergrundsystem verwaltet werden. Ziel der Arbeit war die Entwicklung einer Kontroll-Applikation, die mithilfe der externen TOKA-App ein Identitätstoken generiert und dieses zur Validierung der Fahrberechtigung nutzt. Nach einer Analyse der Anforderungen und Anwendungsfälle wurde die Applikation mit Kotlin Multiplatform und Compose Multiplatform umgesetzt, um eine plattformübergreifende Architektur zu ermöglichen. Die Softwarearchitektur folgt dem MVVM-Muster und integriert Mechanismen für automatisierte Tests sowie eine CI/CD-Pipeline zur kontinuierlichen Auslieferung. Der entwickelte Prototyp erfüllt die definierten Anforderungen und bietet eine Grundlage für eine produktionsreife Lösung. Abschließend werden die Ergebnisse bewertet und ein Ausblick auf mögliche Erweiterungen gegeben.

Inhaltsverzeichnis

1 Einführung in den ÖPV

In Deutschland ist öffentlicher Personenverkehr (ÖPV) im ständigen Wandel: Die bisher kraftstoffbetriebenen Busse werden immer häufiger ersetzt durch elektrische Busse. Ein wachsender Busfahrermangel motiviert das Forschen an autonomen Fahrzeugen im ÖPV und nach einer Preiserhöhung im Jahr 2026 wird der Preis des Deutschlandtickets erstmals nicht mehr erhöht [9].

Auch der Erwerb einer gültigen Fahrberechtigung wird für Ortsfremde zunehmend komplizierter. Durch Tarifsysteme, die in jeder Stadt anders sind, und unterschiedlichste Ticketarten wie zum Beispiel Ermäßigungstickets, Einzeltickets oder Mehrfahrtentickets ist der Erwerb einer Fahrberechtigung nicht mehr trivial. Unter anderem führen die genannten Faktoren dazu, dass das Kaufen einer Fahrberechtigung als „ein undurchsichtiger Dschungel aus Hürden“ [18] beschrieben wird.

So wird auch die Kontrolle von Fahrgästen ein immer aufwändigerer Prozess. Durch die seit Jahren etablierte und fortlaufende Erweiterung der Menge an Ticket-Medien ist der Kontrollprozess immer vielseitiger geworden. Anstatt alle notwendigen Informationen auf ein Papierticket zu drucken, sind viele Verkehrsverbünde daran interessiert, möglichst viele digitale Fahrberechtigungen zu verkaufen. Diese sparen unter anderem Zeit und Geld ein, da beispielsweise nicht mehr regelmäßigen Monats- und Jahreskarten ausgestellt werden müssen. Seit der Einführung von Chipkarten und des internationalen UIC-Barcodes in Deutschland sind Software-unterstütze Kontrollen zum Standard für Kontrolleure geworden, da alle notwendigen Kontrollinformationen auf dem Ticket-Medium gespeichert werden [6].

Seit einigen Jahren ist auch das Thema Account-Based Ticketing (ABT) in Deutschland präsent und wird im Verband Deutscher Verkehrsunternehmen (VDV) sowie in den Softwareunternehmen der ÖPV-Branche aktiv diskutiert. ABT bietet eine Grundlage für den universellen Erwerb und Verkauf von Fahrberechtigungen für Fahrgäste [25]. Die gängigen Ticket-Medien werden dabei durch neue Ticket-Medien ersetzt, wie z. B. eine Bankkarte, die zum Bezahlen und Referenzieren der Fahrberechtigung in einem Hintergrundsystem genutzt wird. Dabei werden keine Ticketinformationen auf der Bankkarte gespeichert, sondern sie sind ausschließlich im Hintergrundsystem hinterlegt [20]. Die Bedeutung von ABT wird in Kapitel 2.1 noch näher erläutert.

In der IVU Traffic Technologies AG (IVU) wird ABT ebenso viel diskutiert. Aktuell arbeiten verschiedenste Teams in den verschiedenen Bereichen der IVU an einer einheitlichen Produktlösung. In Zusammenarbeit mit einem externen Zahlungsdienstleister wird

ein ABT-fähiges System entwickelt, welches vom Erwerb über die Kontrolle bis hin zur Abrechnung im ABT-Kontext zum Einsatz kommen soll.

Dieser Zahlungsdienstleister stellt der IVU auch die sogenannte TOKA-App zur Verfügung, mit der es möglich ist, ein eindeutiges Identitätstoken für eine Bankkarte zu generieren. In Kapitel 2.2 wird ersichtlich, dass die TOKA-App Grundlagen liefert, um eine darauf aufbauende Kontroll-Applikation zu erstellen. Da ABT in der IVU einen großen Stellenwert besitzt, ist die IVU daran interessiert, eine eigene Kontroll-Applikation zum Kontrollieren von Fahrberechtigungen im Kontext von ABT zu erstellen. Aufgrund dessen wird im Laufe dieser Arbeit ein Prototyp einer Kontroll-Applikation mithilfe der TOKA-App erstellt.

In den nächsten Kapiteln dieser Arbeit wird beschrieben, wie und womit die Kontroll-Applikation entwickelt wird. In Kapitel 2 wird zunächst auf die Ticketkontrolle in Deutschland näher eingegangen, um den notwendigen Kontext zur Kontroll-Applikation darzustellen. Anschließend werden in Kapitel 3 die daraus resultierenden Anwendungsfälle und Anforderungen abgeleitet. Danach wird die Softwarearchitektur in den Kapiteln 4 und 5 sowie in Kapitel 6 die konkrete Implementierung der Kontroll-Applikation erläutert. Abschließend wird in den Kapiteln 7 und 8 noch eine Übersicht zu Continuous Integration (CI)/Continuous Delivery (CD) und automatisierten Tests der Kontroll-Applikation gegeben. Zuletzt werden die Erkenntnisse des Prototyps zusammengefasst.

2 Ticketkontrolle in Deutschland

Seit der Jahrtausendwende werden in Deutschland elektronische Fahrberechtigungen im ÖPV verkauft und kontrolliert [20]. Dabei wurden Fahrberechtigungen auf Chipkarten, in Barcodes oder auf Handys ausgegeben. Die Form des Ticket-Mediums nennt sich auch Medium-Based Ticketing (MBT).

Seit 2007 werden (((eTickets nach dem VDV-Kernapplikation (VDV-KA) Standard mit dem (((eTicket Deutschland vertrieben. 2010 kam der VDV-Barcode sowie der internationale UIC-Barcode in den Vertrieb [20]. Vor allem der VDV-Barcode wird in Deutschland viel genutzt. Internationale Ticket-Medien und Fahrberechtigungen sind nur schwer oder nicht kontrollierbar gewesen. Touristen mussten deshalb für die Nutzung des ÖPV in Deutschland immer ein deutsches Ticket-Medium kaufen, um eine gültige Fahrberechtigung zu besitzen.

Fahrberechtigungen im MBT sind die derzeit am häufigsten verkauften Ticketarten in Deutschland [20]. Sie benötigen zur Kontrolle keine dauerhafte Internetverbindung. Es

werden Sperrlisten für Fahrbescheinigungen zyklisch von einem Hintergrundsystem heruntergeladen. Zusätzlich sind alle wichtigen Kontrollinformationen auf dem Ticket-Medium gespeichert, wodurch eine internetlose Kontrolle ermöglicht wird. Aufgrund dessen ist das Kontrollieren der Fahrberechtigungen auch in abgelegenen Regionen Deutschlands möglich.

2.1 ABT - der moderne Weg

ABT ist im Gegensatz zu MBT-basierten Medien ein internationales und länderübergreifendes Verfahren zum Erwerb einer gültigen Fahrberechtigung. Die meiste Datenverarbeitung, inklusive Preisberechnung und Ablage der Fahrberechtigung, findet auf zentralen Hintergrundsystemen statt, wodurch ABT auch als „server-based ticketing“ bezeichnet wird [15]. Die Fahrberechtigung für einen Fahrgast wird über ein Identitätstoken verknüpft, welches für den Fahrgast eindeutig sein soll. Beispielsweise kann eine Bankkarte als Identitätstoken genutzt werden und dient zugleich als Zahlungsmittel für die gekaufte Fahrberechtigung. Dieses Token wird beim Kontrollieren benutzt, um mit einem Hintergrundsystem abzugleichen, ob ein gültiger Fahrausweis vorliegt.

Es gibt bei ABT ein weites Spektrum an Definitionen für die Umsetzungsmöglichkeiten innerhalb der Software- und Verkehrsbetriebe. Es gibt Gesamtlösungen, welche jegliche Berechnungen und Speicherung von Informationen dem zentralen System überlassen und eine durchgängige Internetverbindung benötigen. Andererseits gibt es Teillösungen, die auf eine Zusammenarbeit zwischen lokaler Kontrolllogik und zentralen Systemen beruhen. Vorteil hiervon ist, dass die Teillösungen keine durchgehende Internetverbindung benötigen [16]. In der Praxis ist dies im ÖPV interessant, wenn zum Beispiel Busse durch Dörfer durchfahren, die keine ausreichende Internetverbindung besitzen.

Generell ist ABT ein weltweites Thema und wird von verschiedenen Projekten erforscht und umgesetzt [16]. In London wurde ABT schon 2014 durch eine Kooperation von Mastercard und Transport for London (TFL) eingeführt und ist seitdem im Einsatz [7]. Auch in Deutschland ist ABT ein viel diskutiertes Instrument und wird von der Bundesregierung sowie dem VDV weiter vorangetrieben. In einer Stellungnahme erklärt Oliver Wolff vom Ausschuss für Verkehr und digitale Infrastruktur des Deutschen Bundestages, dass ABT bisher als eine „[...] vielversprechende Lösung für eine grenzüberschreitende Reisekette über unterschiedliche Modalitäten“ [25] angesehen wird. Insbesondere der VDV habe in Deutschland in den letzten Jahren die Vernetzungsinitiative der Branche vorangebracht.

Mit ABT kommt zugleich eine neue Hürde für die Kontrolle von Reisenden. Die bisheri-

gen Kontrollmethoden beruhen auf Medien, die die Fahrberechtigung lokal auf dem Ticket-Medium abspeichern. So werden alle notwendigen Informationen für eine Fahrberechtigung in einem Barcode oder auf einer Chipkarte abgespeichert. Durch den Verkauf von Fahrberechtigungen mit ABT müssen nun auch Ticket-Medien kontrolliert werden können, die keine lokale Speicherung von Fahrberechtigungen vornehmen. Die Kontroll-Applikation soll eine Kontrolle basierend auf einem Identitätstoken und der im Hintergrundsystem gespeicherten Fahrberechtigung durchführen. Nur wenn hinter dem Identitätstoken auf dem Hintergrundsystem eine gültige Fahrberechtigung vorliegt, wird dem Reisenden dieses attestiert.

Die Konzeption und Erstellung des Kontroll-Applikation-Prototyps für das Kontrollieren einer Fahrberechtigung mit ABT ist die Aufgabe dieser Arbeit. Es soll eine mobile Applikation für Smartphones entwickelt werden, die beim Vorhalten eines ABT-fähigen Ticket-Mediums ein Identitätstoken generiert und den Abgleich mit dem Hintergrundsystem vornimmt. Die Generierung des Identitätstoken wird der TOKA-App überlassen, die im ABT-Kontext schon zum Einsatz kommt.

2.2 TOKA

Bei der Generierung des Identitätstoken wird die externe Applikation des Zahlungsdienstleisters namens Tokenization Application (TOKA) zur Hilfe genommen. Sie übernimmt die Aufgabe, aus einem vorgehaltenen Ticket-Medium ein eindeutiges Identitätstoken zu generieren. Um ein Ticket-Medium auszulesen, sind viele Randbedingungen und Informationen notwendig, da zum Beispiel Bankkarten entschlüsselt und geprüft werden müssen, bevor sie ausgelesen werden können. Diese Anforderungen an eine auslesende Applikation müssen zertifiziert werden, bevor sie offiziell genutzt werden dürfen. Zusätzlich soll die Kontroll-Applikation keine Informationen über die Bankkarte und dessen Inhaber besitzen, damit keine Bezahlungen durchgeführt oder personenbezogene Daten über den Inhaber erhoben werden können. Letzteres würde bedeuten, dass die Kontroll-Applikation datenschutzrechtliche Maßnahmen ergreifen müsste. Wegen der genannten Gründe fällt das Auslesen des Ticket-Mediums und das Generieren des Identitätstoken nicht in das Aufgabengebiet der Kontroll-Applikation. In den nachfolgenden Kapiteln dieser Arbeit wird TOKA als TOKA-App bezeichnet, wobei damit die gleiche Applikation gemeint ist.

Die TOKA-App ist eine Android-exklusive Applikation und kann durch Android-spezifische Kommunikationswege angesteuert werden, wobei die Kommunikationsdaten im JSON-Format übertragen werden. Die TOKA-App erkennt per Near Field Communication (NFC)

ein am Smartphone vorgehaltenes Ticket-Medium wie zum Beispiel eine Bankkarte. Zu diesem Ticket-Medium wird ein eindeutiges Identitätstoken generiert und der aufrufenden Applikation wieder mitgeteilt. Somit kann die Kontroll-Applikation mithilfe der TOKA-App ein Identitätstoken generieren.

Anschließend wird das Identitätstoken benutzt, um vom Hintergrundsystem Informationen über die Fahrberechtigung des Reisenden zu erhalten. Es wird unter anderem geprüft, ob das Ticket-Medium auf einer Sperrliste steht, wodurch dem Reisenden keine gültige Fahrberechtigung attestiert werden würde. Dies geschieht genau dann, wenn die Bankkarte nicht belastet werden kann. Falls keine Fahrberechtigung festgestellt wird, wird dies dem Kontrolleur mitgeteilt.

Die TOKA-App legt demnach eine Basis für das Erstellen der Kontroll-Applikation.

3 Anwendungsfälle und Anforderungen

Wie im Kapitel 2.2 schon erläutert wurde, gibt es die zugrundeliegende TOKA-App nur für Android-basierte Geräte. Aufgrund dessen wird auch der Prototyp der Kontroll-Applikation eine mobile Applikation für Android-Geräte sein. Dabei sind noch verschiedene Implementierungsfragen zu klären, die in den nachfolgenden Kapiteln beantwortet werden. So muss unter anderem die Graphical User Interface (GUI)-Bibliothek für die Kontroll-Applikation festgelegt werden oder wie neue Versionen des Prototypen automatisch an die Kontrolleure weitergegeben werden können.

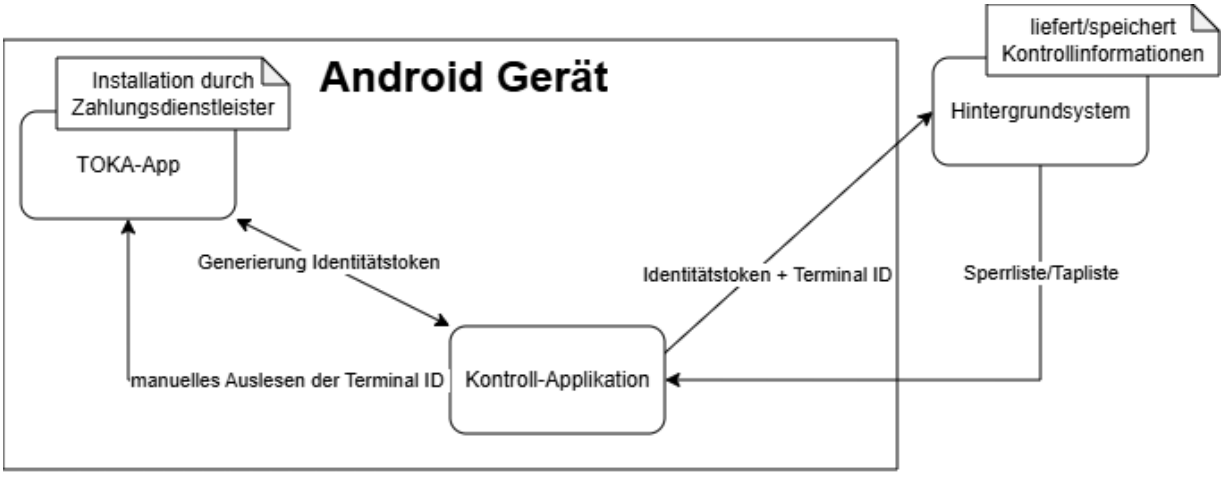

Abbildung 1: Übersicht der Komponenten für die Kontroll-Applikation

3.1 Inbetriebnahme

Für die Inbetriebnahme der Kontroll-Applikation auf einem Android-Gerät sind folgende Schritte notwendig:

1. Die Kontroll-Applikation muss beispielsweise aus einem App-Store installiert werden. Wie die aktuelle Version der Kontroll-Applikation in einen App-Store geladen wird, wird in Kapitel 8 erklärt.

2. Wie in Abbildung 1 angemerkt, muss die TOKA-App vom Zahlungsdienstleister installiert werden. Ohne diese ist das Generieren eines Identitätstoken und demnach auch das Kontrollieren nicht möglich. Die Kontroll-Applikation soll eine Fehlermeldung anzeigen, wenn die TOKA-App nicht installiert oder die Kommunikation mit TOKA nicht möglich ist.

3. Die Terminal-ID, welche Teil der TOKA-App ist und das Kontroll-Gerät eindeutig identifiziert, soll beim erstmaligen Öffnen der Kontroll-Applikation eingegeben werden. Die Kontroll-Applikation kann die Terminal-ID nicht automatisch auslesen. Aus Abbildung 1 kann abgelesen werden, dass die Terminal-ID genutzt wird, um eine Anfrage an das Hintergrundsystem zu stellen. Demnach kann ohne Terminal-ID keine Kontrolle durchgeführt werden.

3.2 Anwendungsfälle

Nach der Inbetriebnahme, soll das Android-Gerät bereit sein, eine Kontrolle mithilfe der Kontroll-Applikation durchführen zu können. In Abbildung 2 wird eine erfolgreiche Kontrolle in Form eines Sequenzdiagramms dargestellt. Es bietet einen Überblick, um die nachfolgenden Anwendungsfälle zu verdeutlichen. Auf die Details des Sequenzdiagramm wird noch genauer in Kapitel 6 eingegangen.

- Wenn der Fahrgast vor der Kontrolle eine gültige Fahrberechtigung erworben hat und das Ticket-Medium nicht auf der Sperrliste steht, dann soll die Kontroll-Applikation ein positives Kontrollergebnis anzeigen.

- Wenn das Ticket-Medium des Fahrgastes auf der Sperrliste steht, dann soll die Kontroll-Applikation eine ungültige Fahrberechtigung anzeigen. Eine Sperrung des Ticket-Mediums kommt genau dann vor, wenn der Fahrgast mit dem Ticket-Medium aufgrund fehlender Zahlungsmittel nicht bezahlen konnte.

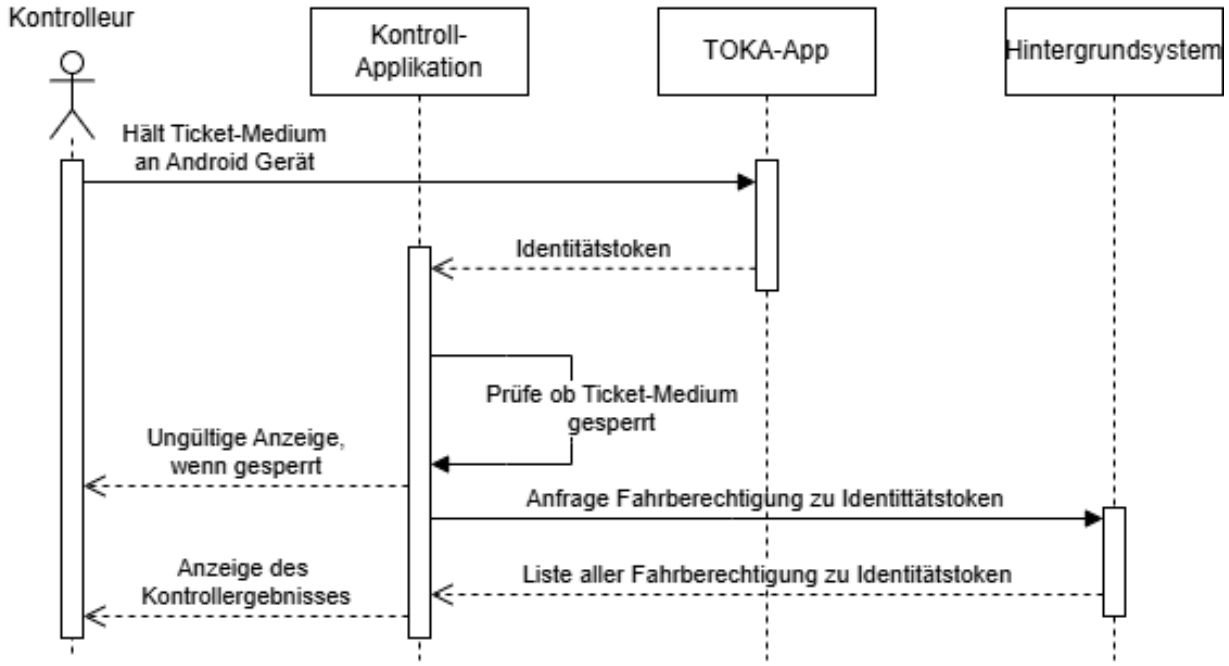

Abbildung 2: Kontrollablauf für eine erfolgreiche Kontrolle

- Wenn der Fahrgast keine Fahrberechtigung vor Antritt der Fahrt gekauft hat, zeigt die Kontroll-Applikation eine ungültige Fahrberechtigung an.
- Wenn beim Vorhalten des Ticket-Mediums von der TOKA-App das Medium nicht ausgelesen werden konnte, um das Identitätstoken zu generieren, dann soll die Kontroll-Applikation einen Fehlerzustand anzeigen. Zugleich wird der Kontrolleur aufgefordert, das Ticket-Medium wiederholt vorzuhalten, um die Kontrolle der Fahrberechtigung erneut zu versuchen.
- Wenn die Kommunikation mit dem Hintergrundsystem nicht möglich ist, beispielsweise durch eine fehlende Internetverbindung oder einen Hintergrundsystemfehler, kann das zu kontrollierende Ticket-Medium nicht auf eine gültige oder ungültige Fahrberechtigung geprüft werden. Die Kontroll-Applikation soll eine entsprechende Fehlermeldung anzeigen.
- Wenn die Kommunikation mit dem Hintergrundsystem nicht möglich ist, das Ticket-Medium jedoch auf der Sperrliste gefunden wurde, dann soll die Kontroll-Applikation eine ungültige Fahrberechtigung anzeigen.

Die Kontrolle einer Fahrberechtigung eines Fahrgastes wird, wie in Abbildung 2 zu sehen ist, mit einer Anfrage an ein Hintergrundsystem erfolgen. Dabei wird geprüft, ob

eine gültige Fahrberechtigung im Hintergrundsystem hinterlegt ist. Eine weitere Prüfung auf räumliche Gültigkeit oder Liniengültigkeit wird nicht Teil dieser Arbeit sein. Diese Prüfungen würden beispielsweise durch das Einbinden von IVU-internen C++ Bibliotheken ermöglicht werden, welche unter Android mithilfe des Java Native Interface innerhalb der Java Virtual Machine (JVM) laufen würden. Dieser Schritt würde den Rahmen des Prototyps übersteigen und ist aufgrund dessen nicht Teil der Kontroll-Applikation.

Basierend auf den genannten Anwendungsfällen und den beschriebenen Schritten für die Inbetriebnahme ergeben sich unter anderem Testfälle, auf welche in Kapitel 7 näher eingegangen wird. Die Implementierung und Umsetzung der Anwendungsfälle innerhalb des Prototypen werden in Kapitel 6 erläutert.

3.3 Fachliche Anforderungen

Zusätzlich zu den genannten Anwendungsfällen gibt es weitere fachliche Anforderungen an die Kontroll-Applikation. Aufgrund dessen werden zusätzlich zu den Anwendungsfällen nicht-funktionale Anforderungen definiert, die sich mit der Speicherung von Daten befassen:

- Die Terminal-ID soll nach der erstmaligen Eingabe in der Kontroll-Applikation auf dem Android-Gerät gespeichert werden, damit die Terminal-ID nicht bei jedem Start der Kontroll-Applikation erneut eingegeben werden muss.
- Die Sperrliste soll periodisch vom Hintergrundsystem geladen und auf dem Android-Gerät gespeichert werden. Dies ermöglicht eine Teil-Kontrolle, falls die Verbindung zum Hintergrundsystem am Kontrollzeitpunkt nicht vorhanden ist. In dem Fall kann noch verglichen werden, ob das Ticket-Medium auf der Sperrliste steht, wie es in den Anwendungsfällen schon beschrieben wurde.

Zusammen mit den fachlichen Anforderungen ergeben die Anwendungsfälle eine Fragestellung, die die Entwicklung des Prototyps beantworten soll: „Wie könnte eine mobile Android-Applikation zum Kontrollieren von Fahrberechtigungen mit ABT technisch und inhaltlich aussehen?“

4 Kotlin Multiplattform mit Compose Multiplattform

Um die Wahl einer GUI-Bibliothek zu treffen, wurden zunächst zwei verschiedene Ansätze verglichen. Konkret ging es dabei um die GUI-Bibliothek Jetpack Compose im Vergleich zur XML basierten GUI-Entwicklung. Der Vergleich der beiden Varianten ergab, dass Compose die GUI-Bibliothek für die Kontroll-Applikation aufgrund der im folgenden beschriebenen Vorteile sein wird.

4.1 Jetpack Compose - Deklarative GUI

Jetpack Compose ist, im Gegensatz zur imperativen GUI-Entwicklung, eine deklarative GUI-Bibliothek. Jetpack Compose unterscheidet sich wesentlich vom imperativen Ansatz in folgenden Aspekten wie beschrieben in [24]:

- Die GUI-Komponenten werden in Kotlin geschrieben, derselben Programmiersprache, die auch für die Entwicklung von Android-Applikationen genutzt wird. GUI-Komponenten werden als Funktionen beschrieben und bieten dadurch eine hohe Wiederverwendbarkeit in mehreren Teilen einer Applikation.
- Die Performance der GUI kann aufgrund der 'smart recomposition' spürbar schneller sein als der imperative Ansatz. Dabei werden nur die GUI-Komponenten neu gezeichnet, wenn diese eine Änderung in ihrem Zustand haben.
- Die Menge an Code, welcher die GUI-Komponenten mit der Logik verbindet, wird reduziert, da nicht mehr mit `findViewById` gearbeitet werden muss, um die GUI-Komponente mit der Logik zu verbinden.

Zusätzlich zu den genannten Aspekten ist Jetpack Compose auch die modernere Bibliothek, die durchgehend weiterentwickelt wird und eine große Gemeinschaft hat, um Fragen und Probleme zu klären. Aufgrund der verschiedenen Aspekte wurde die GUI-Bibliothek Compose für das Projekt ausgewählt.

4.2 Kotlin Multiplattform

Zugleich wurde auch in Betracht gezogen, dass die Kontroll-Applikation auf mehreren Plattformen laufen können soll. Dafür wurde ein Vergleich zwischen Android-Nativer und

Kotlin Multiplatform (KMP)-App-Entwicklung durchgeführt, um herauszufinden, wie viel Mehraufwand eine Entwicklung mit KMP mit sich zieht.

KMP ist ein Framework, um mehrere Plattformen in einer Codebasis gleichzeitig zu unterstützen. Gemeinsame Codeanteile können plattformübergreifend verwendet werden. Genauso können plattformspezifische Inhalte zugeschnitten auf die Plattform entwickelt werden. Unter anderem zählen Android, iOS, Desktop (JVM) und Server-Side (JVM) zu den unterstützten Plattformen. [14].

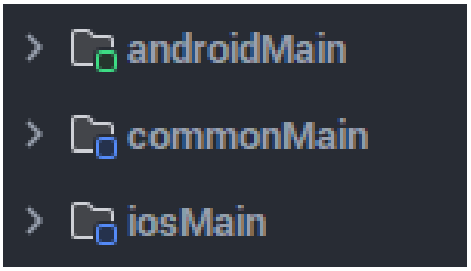

Abbildung 3: Ordnerstruktur einer KMP Applikation

Die unterstützten Plattformen lassen sich durch die Ordnerstruktur erkennen und unterscheiden. Die in Abbildung 3 gezeigte Ordnerstruktur entspricht einer KMP Applikation. In `commonMain` sind alle Codeanteile enthalten, die plattformunabhängig sind, wie zum Beispiel die Business-Logik. In `androidMain` liegen alle Android- und in `iosMain` alle iOS-spezifischen Codeanteile wie zum Beispiel die Interaktion mit Hardwarekomponenten oder Abspeichern von Daten auf einem Gerät.

Durch den Vergleich wurde klar, dass mehrere Plattformen mithilfe von KMP mit geringem Mehraufwand unterstützt werden können. Die Implementierung einer Multiplattform-Applikation unterscheidet sich nicht fundamental von der einer Android-nativen Applikation, da Android-nativ im Wesentlichen ein Subset von KMP ist. Eine gute App-Architektur entsteht, wenn beispielsweise Klassen durch Interfaces entkoppelt und durch Dependency Injection (DI) übergeben werden [12]. Diese Vorgehensweise wird in KMP vertieft und durch die Keywords `expect/actual` erweitert. `expect` ist ein Interface in `commonMain`, welches vorgibt, wie die plattformspezifische Implementierung aussieht. Die tatsächliche plattformspezifische Umsetzung nutzt `actual` [13].

4.3 Compose Multiplatform

Nachdem in Kapitel 4.2 festgelegt wurde, dass für mehrere Plattformen entwickelt werden kann, müsste noch die plattformunabhängige GUI-Bibliothek festgelegt werden. Durch KMP wurde eine Multiplattform-Variante von Jetpack Compose namens Compose Multiplatform (CMP) in Betracht gezogen. Die Eigenschaften von Jetpack Compose wurden schon in Kapitel 4.1 diskutiert und viele der dort genannten Aspekte sind auch für CMP zutreffend:

- CMP baut auf den gleichen Prinzipien wie Jetpack Compose auf: Es ist deklarativ, hat die gleiche Syntax und ist ebenso in Kotlin geschrieben [3].
- CMP besitzt fast die gleichen Funktionen wie Jetpack Compose. Es fehlen nur ein paar wenige Klassen wie zum Beispiel die native Integration einer Kartenansicht in Jetpack Compose [11].
- CMP bindet Drittanbieter-Bibliotheken ein [11].

Wenn eine KMP mit CMP-Applikation für Android gebaut wird, dann werden intern die Pakete von Jetpack Compose verwendet, um die Android-Applikation zu bauen [4]. Seit Mai 2025 ist CMP als stabile Laufzeitkomponente gekennzeichnet und gilt als produktionsreif für iOS-Geräte. Dort wird nach SwiftUI kompiliert, das native GUI-Framework für iOS [2]. Aufgrund dessen wurde CMP als GUI-Bibliothek gewählt, um weiterhin plattformübergreifend entwickeln zu können.

Jedoch stellt die TOKA-App ein Problem für die iOS-Applikation dar, denn die TOKA-App gibt es nur für Android-Geräte - wie schon in Kapitel 2.2 erwähnt. Demnach wäre es sinnvoll, die Kontroll-Applikation nur auf Android-Geräten zu entwickeln. Allerdings wurden in den Kapiteln 4.2 und 4.3 schon gezeigt, dass mit wenig Mehraufwand eine Multiplattform-Variante der Applikation gebaut werden kann. Dabei entstehen kaum Einschränkungen, wenn für beide Betriebssystemvarianten gleichzeitig entwickelt wird. Zusätzlich ist es zukünftig möglich, dass die TOKA-App unter iOS funktioniert. Daraus folgt, dass die Kontroll-Applikation zunächst nicht auf iOS-Geräten laufen soll. Jedoch soll die Integration der iOS-Geräte mit in Betracht gezogen werden, um in der Zukunft mehrere Plattformen unterstützen zu können.

5 Applikationsarchitektur

Aus den in Kapitel 4 diskutierten Möglichkeiten ergibt sich, dass die Kontroll-Applikation mit KMP und CMP entwickelt wird. Aufbauend auf der grundlegenden Architektur werden weitere Softwarearchitekturmuster und Bauprozesse genutzt.

5.1 Softwarearchitekturmuster

Der Prototyp wird nach dem Model-View-ViewModel (MVVM) Architekturmuster aufgebaut, um eine klare Trennung zwischen Benutzeroberfläche, Logik und Daten zu gewährleisten.

Die 3 verschiedenen Rollen des Architekturmusters können - auf den Prototypen bezogen - wie folgt beschrieben werden:

- **Model**: Die verschiedenen Modelle beschreiben die Business-Logik der Kontroll-Applikation. Unter anderem zählen die Anbindung an das Hintergrundsystem, die Kommunikation mit der TOKA-App sowie die Logik für das Kontrollieren einer gültigen Fahrberechtigung dazu.
- **ViewModel**: Diese sind eng gekoppelt mit der View und stellen Daten zur Darstellung bereit. Das ViewModel sammelt Daten aus verschiedenen Modellen und bereitet diese zur Darstellung auf.
- **View**: Views bestehen aus simplen deklarativen GUI-Komponenten, die die aufbereiteten Daten eines ViewModels anzeigen. Dabei besitzen Views keine Business-Logik, sondern nur Logik zur Konfiguration der angezeigten Komponenten.

Die meisten Code-Anteile innerhalb der 3 Rollen sind im `commonMain`-Ordner wiederzufinden. Vor allem die Views und ViewModels haben keine plattformspezifischen Unterschiede und bauen auf der gleichen Code-Basis auf. Bei den Modellen gibt es manchmal Abweichungen bei Hardware-Schnittstellen, was dazu führt, dass dort mit plattformspezifischem Code gearbeitet wird. Dieser liegt in `androidMain` und `nativeMain`. So ist beispielsweise das Kommunizieren und Interagieren mit einer anderen Applikation in jedem Betriebssystem unterschiedlich.

5.2 Bauprozess

Der Bauprozess ist mithilfe von Gradle plattformübergreifend einheitlich geregelt. Gradle bietet Möglichkeiten zur einheitlichen Definition von Abhängigkeiten für jede einzelne Plattform. Gleichzeitig können auch plattformspezifische Einstellungen vorgenommen werden.

Grundsätzlich werden die meisten Abhängigkeiten für die `commonMain` Code-Anteile definiert. Dies ist nicht immer möglich, da Abhängigkeiten plattformspezifisch umgesetzt sein können. So muss zum Beispiel für die Kommunikation über einen HTTP-Client eine HTTP-Client-Engine für jede Plattform festgelegt werden. In Android wird OkHttp und für native Plattformen Darwin als HTTP-Client-Engine verwendet [10]. Die jeweiligen plattformspezifischen Abhängigkeiten werden in der entsprechenden Plattform initialisiert

und anschließend über den in Kapitel 4.2 beschriebenen Prozess einheitlich in `commonMain` verwendet.

Auch plattformspezifische Einstellungen können in Gradle vorgenommen werden. Für Android kann zum Beispiel die Signierung der Applikation beim Bauen einer Release-Version konfiguriert werden. In iOS unterscheidet sich der Signierungsprozess von dem für Android-Geräte. Trotzdem wird dieser genauso in Gradle definiert und ausgeführt.

6 Prototyp der Kontroll-Applikation

Die in Kapitel 5.1 beschriebene Architektur wird während der Umsetzung der Kontroll-Applikation durchgehend angewandt und bietet eine gute Grundlage für sofwareseitige Designentscheidungen. In diesem Kapitel wird auf die konkrete Umsetzung der Anwendungsfälle, der fachlichen Anforderungen sowie der Inbetriebnahme aus Kapitel 3 eingegangen.

6.1 Anbindung der TOKA-App

In Kapitel 2.2 wurde die TOKA-App eingeführt, wobei erklärt wurde, dass die TOKA-App eine Android-exklusive Applikation ist. Wie jedoch in Kapitel 4 beschrieben wurde, soll die Kontroll-Applikation plattformunabhängig funktionieren können. Dafür muss die Anbindung der TOKA-App in `commonMain` definiert werden. Sie nutzt Android-spezifische Broadcasts, um ihre generierten Daten an andere Applikationen weiterzugeben. Bevor auf die plattformunabhängige Umsetzung eingegangen werden kann, muss noch im Detail erklärt werden, wie und welche Informationen die TOKA-APP versendet.

Die TOKA-App erkennt automatisch, dass ein Ticket-Medium an den NFC-Leser des Android-Gerätes vorgehalten wird. Daraufhin versucht die TOKA-App das Ticket-Medium auszulesen, wobei es zu drei verschiedenen Ergebnissen kommen kann: Das Ticket-Medium konnte erfolgreich ausgelesen werden und ein Identitätstoken wurde generiert; das Ticket-Medium konnte nicht ausgelesen werden; es gab einen anderen unbekannten Fehler während der Generierung. Bei jedem der drei Möglichkeiten sendet die TOKA-App einen Broadcast mit Daten, die entweder einen Fehler oder ein generiertes Identitätstoken beinhalten.

In der Kontroll-Applikation wurde ein BroadcastReceiver implementiert, der auf den Broadcast der TOKA-App reagiert. Damit der BroadcastReceiver die Daten empfangen kann, müssen der Kontroll-Applikation noch besondere Berechtigungen erteilt werden, die in der

`AndroidManifest.xml` definiert werden. Sobald ein Ticket-Medium an den NFC-Leser des Android-Gerätes vorgehalten wird und die TOKA-App ein Ergebnis liefert, bekommt dies die Kontroll-Applikation mit. Anschließend kann das Identitätstoken weiter verarbeitet werden.

Um die Anbindung plattformunabhängig zu gestalten, braucht es einen Kommunikationsweg, um die empfangenen Daten der TOKA-App auf `commonMain`-Ebene zu erhalten und weiterzuleiten. Dafür wird ein Observer-Pattern genutzt, das aus der Standardbibliothek in Kotlin stammt. Die SharedFlows werden auf ModelView-Ebene genutzt, um auf den Broadcast zu reagieren, der durch die TOKA-App ausgesendet wird. Daraufhin werden die Daten, wie schon in Kapitel 5.1 erklärt, für die Darstellung in der View aufbereitet, wie es im Screenshot der Kontroll-Applikation in Abbildung 6 zu sehen ist. Unter iOS wäre das Einbinden der TOKA-App demnach über das gleiche Observer-Pattern möglich.

6.2 Umsetzung der Inbetriebnahme

Die in Kapitel 3.1 beschriebenen Anforderungen an die Inbetriebnahme, also das Installieren der TOKA-App und das Eingeben der Terminal-ID, wurden unabhängig voneinander umgesetzt. Das Installieren der Kontroll-Applikation ist Teil der Entwicklung und in Kapitel 8 wird erläutert, wie sie heruntergeladen werden kann.

- Um die TOKA-App zu installieren, wurden die Android-Geräte an den Zahlungsdienstleister versendet, damit dieser die TOKA-App installieren und konfigurieren kann. Die TOKA-App muss vom Zahlungsdienstleister auf das Android-Gerät geladen werden, da zugleich bei der Installation bestimmte Zertifikate notwendig sind, ohne die das Auslesen eines Ticket-Mediums per NFC nicht möglich wäre. Anschließend wurden die Android-Geräte zurück an die IVU gesendet, um den Kontroll-Applikations-Prototyp entwickeln zu können.

- Das manuelle Eingeben der Terminal-ID wurde über eine eigene Ansicht gelöst, welche beim erstmaligen Öffnen der Kontroll-Applikation dargestellt wird. Wie in Abbildung 4 zu sehen ist, wird der Kontrolleur darum gebeten, die Terminal-ID manuell einzugeben. Dazu gibt es einen kurzen Hinweistext, wodurch der Kontrolleur weiß, woher er die Terminal-ID beziehen soll. Nach der Eingabe einer Terminal-ID wird ein Bestätigungsdialog eingeblendet. Dieser soll gewährleisten, dass der Kontrolleur die korrekte Terminal-ID eingegeben hat. Bei einer Bestätigung wird die nächste Ansicht dargestellt, wie sie in Abbildung 5 zu sehen ist.

Um die fachliche Anforderung an die Terminal-ID aus Kapitel 3.3 einzuhalten, wird die Terminal-ID auf dem Gerät abgespeichert. Mithilfe eines StorageRepository auf Modell-Ebene wird das Speichern und Auslesen plattform-unabhängig umgesetzt. Sobald die Kontroll-Applikation geöffnet wird, prüft die ViewModel-Ebene mit dem StorageRepository, ob eine Terminal-ID vorhanden ist. Wenn die Terminal-ID schon im Gerätespeicher vorliegt, wird die nächste Ansicht angezeigt, ohne dass der Kontrolleur erneut die Terminal-ID eingeben muss. Dadurch wird die fachliche Anforderung umgesetzt.

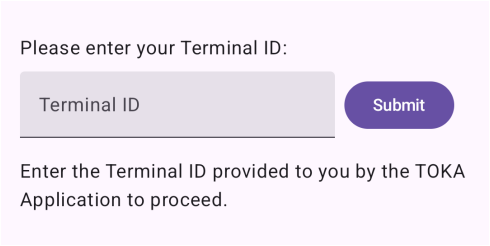

Abbildung 4: Kontroll-Applikation Terminal-ID Input

6.3 Umsetzung der Anwendungsfälle

Bevor auf die Umsetzung der Anwendungsfälle aus Kapitel 3.2 eingegangen werden kann, muss die fachliche Anforderung aus Kapitel 3.3 zur Sperrliste umgesetzt werden, da die Anwendungsfälle teilweise auf dieser fachlichen Anforderung basieren.

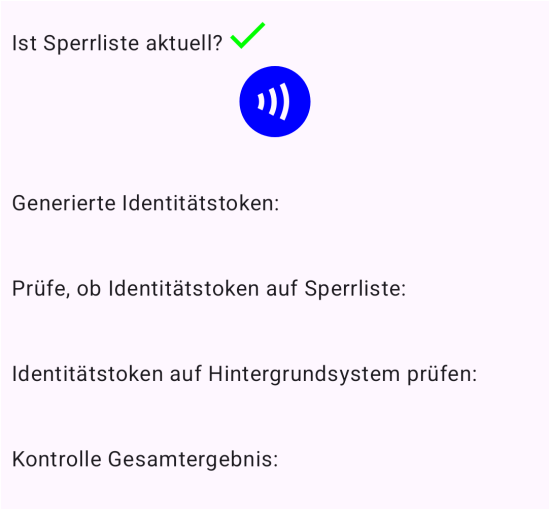

Abbildung 5: Kontroll-Applikation Hauptansicht

Die Sperrliste wird periodisch vom Hintergrundsystem geladen, indem eine parallele Aufgabe beim Starten der Kontroll-Applikation ausgeführt wird, die bis zum Beenden der Kontroll-Applikation für das Herunterladen der Sperrliste zuständig ist. Die Kontroll-Applikation holt die Sperrliste vom Hintergrundsystem und speichert diese im Gerätespeicher mit der Information zusammen ab, zu welchem Zeitpunkt die Sperrliste das letzte mal gespeichert wurde. Wie alt die abgespeicherte Sperrliste für eine Kontrolle sein darf, ist konfigurierbar. Die Konfiguration steuert zugleich das zeitliche Intervall, nach welchem die Sperrliste erneut heruntergeladen wird. Die Kontroll-Applikation kann bei Bedarf daraufhin die aktuelle Sperrliste aus dem Gerätespeicher laden, um eine Kontrolle durchführen zu können. Ob die Sperrliste für eine Kontrolle geeignet ist, wird in der

Kontroll-Applikation durch einen entsprechenden Hinweis angezeigt, wie es in Abbildung 5 zu sehen ist. Dadurch wird die fachliche Anforderung umgesetzt, da die Teil-Kontrolle bei fehlender Verbindung zum Hintergrundsystem über die lokale Sperrliste durchgeführt werden kann.

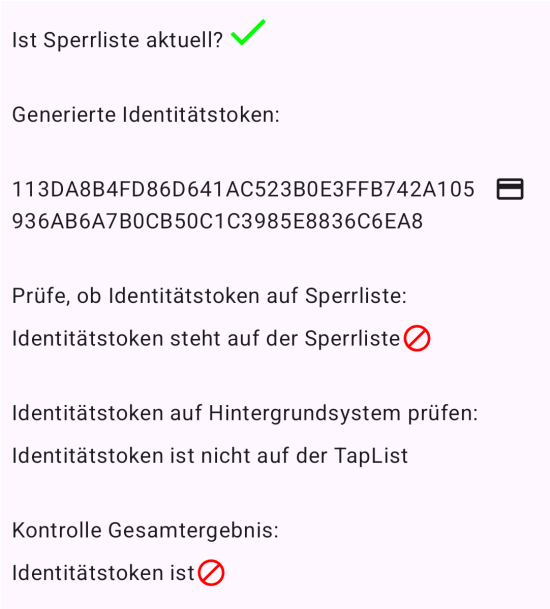

Abbildung 6: Kontroll-Applikation Identitätstoken auf Sperrliste

Um die Anwendungsfälle besser nachvollziehen zu können, wurde die Ansicht während der Kontrolle im Prototyp der Kontroll-Applikation so aufgebaut, dass jeder Schritt in der richtigen Reihenfolge der Kontrolle abgebildet ist. Dabei verhält sich die Kontrolle so, wie es im Sequenzdiagramm aus Abbildung 2 zu sehen ist. Die Kontrolle beginnt damit, dass ein Ticket-Medium an den NFC-Leser des Android-Gerätes vorgehalten wird und die TOKA-App darauffolgend das generierte Identitätstoken broadcasted. Anschließend wird das Identitätstoken gegen die lokal gespeicherte Sperrliste geprüft und gegebenenfalls als ungültig gekennzeichnet. Ansonsten wird das Identitätstoken genutzt, um mit dem Hintergrundsystem abzugleichen, ob eine gültige Fahrberechtigung vorliegt. Wie die einzelnen Anwendungsfälle umgesetzt wurden, wird im Folgenden in gleicher Reihenfolge wie in Kapitel 3.2 erläutert:

- Im Gegensatz zu dem in Abbildung 6 abgebildeten Gesamtergebnis wird bei einem positiven Kontrollergebnis ein grüner Haken angezeigt, der eine gültige Fahrberechtigung symbolisiert.
- Sobald das generierte Identitätstoken auf der Sperrliste gefunden wurde, wird die ungültige Fahrberechtigung mit einem entsprechenden Icon symbolisiert. Dieses wird, wie in Abbildung 6 zu sehen, einmal bei der Sperrlistenprüfung und ein zweites Mal bei dem Gesamtergebnis der Kontrolle angezeigt.
- Um anzuzeigen, dass zu dem generierten Identitätstoken keine Fahrberechtigung existiert, wird analog zur Sperrliste ein entsprechendes Icon angezeigt. Diesmal besteht dieses aus einem roten Hintergrund und einem Ausrufezeichen, womit dem Kon-

trolleur verdeutlicht wird, dass der Reisende keine Fahrberechtigung vor Antritt der Fahrt erworben hat.

- Wenn beim Vorhalten eines Ticket-Mediums von der TOKA-App das Medium nicht ausgelesen werden kann, wird dieser Fehler in der Ansicht angezeigt. Dafür wird ein roter Text mit entsprechender Fehlermeldung unterhalb der in Abbildung 5 angezeigten Schritte dargestellt. Dieser Text beinhaltet einen Aufforderung an den Kontrolleur, das Ticket-Medium erneut vorzuhalten, damit ein Identitätstoken erfolgreich generiert werden kann.

- Um die fehlende Kommunikation zum Hintergrundsystem darzustellen, wurde eine Statusanzeige erstellt, welche dem Kontrolleur anzeigt, dass er keine Internetverbindung hat. Die in Abbildung 7 abgebildete Statusbar kann entweder den Zustand „Offline“ oder „Backend unreachable“ besitzen. Je nachdem, welches von beiden aktuell zutrifft, wird die Anzeige der Statusbar geändert. Ähnlich zu der beschriebenen nicht-funktionalen Anforderung der Sperrliste, wird auch hier eine parallele Aufgabe gestartet, welche während der gesamten Laufzeit der Kontroll-Applikation aktiv ist.

- In Sequenzdiagramm aus Abbildung 2 ist beschrieben, dass die Sperrlistenprüfung ausgeführt wird, bevor die Fahrberechtigung mit dem Hintergrundsystem abgeglichen wird. Im Falle eines Eintrags in der Sperrliste bedeutet dies, dass die Kontrolle beendet wird, bevor die Internetverbindung notwendig ist. Zusammen mit der beschrieben nicht-funtkionalen Anforderung der Sperrliste ergibt dies, dass eine Teilkontrolle ohne Verbindung zum Hintergrundsystem durchgeführt werden kann.

Die Abbildungen in diesem Kapitel sind prototypisch implementiert und entsprechen nicht dem finalen Produkt. Das Ziel des Prototyps ist es, dass die Kontroll-Applikation einen produktiven Implementierungsstand erreicht, weshalb die in Kapitel 7 und 8 beschriebenen Maßnahmen erfolgen. Aufgrund dessen werden die abgebildeten Ansichten im Nachgang noch überarbeitet, sodass sie nur die relevanten Informationen für die Kontrolle beinhalten. Der Kontrolleur wird dementsprechend das Gesamtergebnis der Kontrolle angezeigt bekommen.

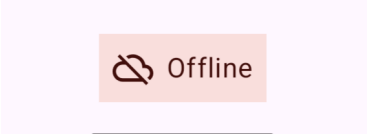

Abbildung 7: Kontroll-Applikation Verbindungsstatusbar

7 Automatisierte Tests zur Fehlererkennung

Bei der Implementierung der Kontroll-Applikation wird der geschriebene Code durchgängig an neue Funktionalitäten oder an behobene Fehler angepasst. Durch diesen Prozess kann es zu Fehlverhalten in schon existierenden Funktionalitäten kommen. Diese Fehler werden in der Literatur Regressionsfehler genannt [17]. Die Code-Anteile können Abhängigkeiten an bestimmte Funktionalitäten oder Abläufe besitzen, die sich durch das Implementieren einer neuen Funktionalität oder das Lösen eines Fehlers ändern können. Demnach ist das Testen des geschriebenen Codes ein sinnvoller Schritt im Entwicklungsprozess, um ungewolltes Fehlverhalten zu vermeiden.

Die Testpyramide in Abbildung 8 sieht vor, dass Tests in 3 Ebenen unterteilt werden: GUI-Tests, Service- oder Integration-Tests und Unit-Tests. Innerhalb der Kontroll-Applikation werden allerdings nur zwei Testarten verwendet, da bisher noch keine Service-Tests aufgesetzt wurden. Diese sind aktuell im Aufbau und werden erst nach dem Ende des Projekts lauffähig sein.

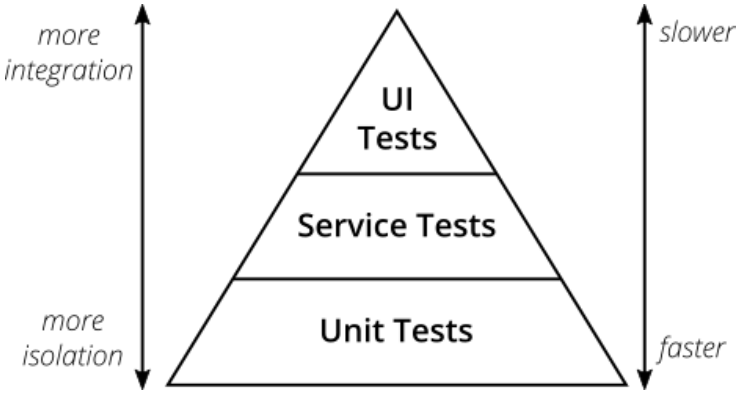

Abbildung 8: Testpyramide nach Mike Cohn [22]

7.1 Unit-Tests

Unit-Tests bilden beim Testen die Grundlage der Testpyramide und sichern den Code auf einer grundlegenden Ebene ab. Sie testen Module, häufig definiert als eine Klasse oder als kleine Sammlung von Funktionen im Code. Die Module sind größenmäßig übersichtlich, gut zu testen und im besten Fall abhängig von wenigen anderen Modulen. Aufgrund dessen haben die geschriebenen Testfälle eine Laufzeit von wenigen Millisekunden, wie in Abbildung 9 zu sehen ist. Dies erlaubt hunderten Unit-Tests in kurzer Zeit zu prüfen, ob der neue geschriebene Code eine bestehende Funktionalität aus Versehen beeinflusst hat [22].

Auch wenn es häufig nur wenige Abhängigkeiten gibt, können diese zu Problemen beim Testen führen. Eine Klasse kann beispielsweise von einem HTTP-Client abhängig sein,

der normalerweise Anfragen ins Internet verschickt. Diese dauern im Normalfall ein paar hundert Millisekunden und würden einen Unit-Test um ein Hundertfaches verlangsamen. Zusätzlich wäre in dem konkreten Beispiel schwer zu steuern, wie sich der Server im Internet verhält. Eine Server-Testumgebung aufzusetzten, die alle Fehler- und Erfolgsbeispiele wiedergeben kann, ist zeit- und gegebenenfalls kostenintensiv. Auch das Testen mit einer Hardware-Schnittstelle wie von der TOKA-App verwendet wäre schwer zu ermöglichen, da jedes Fehler-szenario der TOKA-App über eine eigene TOKA-Applikation geprüft werden müsste. Aus diesen Gründen wird eine Fake-Abhängigkeit dem zu testenden Modul übergeben. Diese Vorgehensweise wird im Allgemeinen als Mocking bezeichnet.

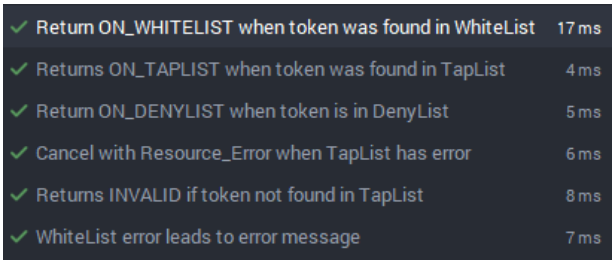

Abbildung 9: Beispiel Unit-Test Ausführung

Beim Mocking werden eine oder mehrere Pseudo-Klassen implementiert. Diese implementieren die Schnittstelle der Abhängigkeit, sodass im Unit-Test bestimmte Verhaltensweisen im Modul getestet werden können [23]. So kann beispielsweise dem Modul mit der HTTP-Client-Abhängigkeit vorgetäuscht werden, dass die Anfrage einen festgelegten Datensatz oder der Server einen Fehler zurückgibt, ohne dass eine echte Anfrage ins Internet gemacht wird. Somit ist die Laufzeit eines Unit-Tests wieder auf ein paar Millisekunden beschränkt.

In der Kontroll-Applikation sind die meisten Tests im `commonTest` Ordner geschrieben, da der Großteil des Codes in `commonMain` liegt. Doch auch die plattformspezifischen Komponenten, die mit `expect/actual` geschrieben wurden, müssen mit Unit-Tests abgedeckt werden. Wichtig dabei ist, dass die Unit-Tests nun plattformspezifisch arbeiten und das gesonderte Testabhängigkeiten definiert werden müssen. So ist auch das Mocking, wie oben beschrieben, möglich. Jedoch wird hier zusätzlich eine Bibliothek genutzt, die das Mocken von Android-spezifischem Kontext übernimmt. Dies dient der schnelleren Erstellung und Umsetzung der Unit-Tests durch Nutzung etablierter Bibliotheken.

7.2 GUI-Tests

Auch wenn Unit-Tests am häufigsten benutzt werden, um Code zu testen, sind zusätzliche GUI-Tests eine wichtige Testform, um die GUI-Komponenten sinnvoll zu prüfen. Wie in Kapitel 5.1 beschrieben, beinhalten Views auch eine Anzeigelogik, die ungetestet ebenso zu

Fehlern führen kann. Zusätzlich spielen in modernen Anwendungen die Benutzerführung und die intuitive Bedienung der Applikationen eine immer größere Rolle [19]. Aufgrund dessen sollten auch die einzelnen GUI-Komponenten getestet werden, um ungewollte Fehlerquellen beim Anzeigen in der View zu verhindern.

Dabei weichen die hier beschriebenen GUI-Tests von der gängigen Interpretation ab. Im Allgemeinen werden sie als Tests für die gesamte Prüfung der Software verstanden und von Entwicklern häufig als weniger notwendig angesehen. Demnach gibt es eine deutlich höhere Menge an Unit-Tests als GUI-Tests, weil Unit-Tests noch andere wichtige Faktoren beinhalten. Zum Beispiel sind Unit-Tests stabil, sie haben eine schnelle Durchlaufzeit und können komplexe Szenarien einfach testen [21]. In der Kontroll-Applikation wird ein GUI-Test als ein Test der Anzeigelogik verstanden und wird dementsprechend als gleichgültig zu Unit-Tests angesehen.

In Kapitel 4.3 wurde für CMP ein einheitlicher Weg erläutert, um GUI-Tests zu schreiben. Mithilfe der bereitgestellten `runComposeUiTest`-Methode kann der aktuell zu testende Inhalt festgelegt und die zu prüfende Anzeigelogik getestet werden. Die bereitgestellten Methoden zum Testen beinhalten folgende Funktionalitäten:

- Finden einer bestimmten GUI-Komponente durch die Suche nach einem festgelegten Test-Tag oder durch die Suche nach dem aktuellen Text der Komponente.
- Sicherstellen der Anzeige von bestimmten Subkomponenten oder im Gegenfall der ausbleibenden Anzeige dieser Komponenten. Dies wird durch die Anzeigelogik innerhalb der GUI-Komponente gesteuert und kann durch das Mocken eines ViewModels getestet werden.
- Interagieren mit einzelnen Subkomponenten wie zum Beispiel das Klicken auf einen Knopf oder eine TextBox.
- Überprüfen von Eigenschaften der einzelnen GUI-Komponenten wie zum Beispiel, dass ein genauer Text vorhanden ist, dass die Komponente anklickbar ist oder dass die Höhe/Breite einem vorgegebenen Wert entspricht.

Die Tests werden normalerweise auf jeder konfigurierten Plattform durch das Anschließen eines physischen Gerätes ausgeführt. Jedoch gibt es beim Testen eine wesentliche Anforderung, nämlich dass die GUI-Tests automatisch ausgeführt werden können, ohne dass ein physisches Gerät angeschlossen sein muss. Aus diesem Grund wird für die GUI-Tests ein Emulator verwendet. Dieser simuliert die Ausführung auf einem physischen Gerät und

bietet dadurch eine unabhängige Testausführung und führt zusätzlich zu einer verkürzten Laufzeit der Tests.

Der für Android genutzte Emulator ist `Robolectric` und funktioniert technisch gesehen nicht wie ein klassischer Emulator eines physischen Gerätes. `Robolectric` läuft innerhalb der JVM in Sekundenschnelle und bietet eine Testumgebung, welche konfigurierbar ist und Android präzise emuliert [5]. Die Testumgebung muss vor jedem Test eingerichtet werden, jedoch wird dies dem Test-Schreibenden durch die Vererbung der selbstgeschriebenen `UsingContext`-Klasse abgenommen, die für die Konfiguration zuständig ist. Da nicht jede Plattform mit `Robolectric` läuft, wird auch hier wieder der `expect/actual` Mechanismus genutzt wie in Kapitel 4.2 beschrieben, um plattformspezifische Inhalte zu abstrahieren.

8 Continuos Integration (CI) und Continuos Delivery (CD)

In der Softwarewelt ist eine CI/CD-Pipeline ein fester Bestandteil der Softwareentwicklung. CI/CD ist ein wichtiger Grundbaustein für die skalierbare Integration und Auslieferung von neuer Software. Unter anderem spielen das automatische Bauen und Testen, die Unterstützung mehrerer Plattformen und die Skalierbarkeit der Infrastruktur eine große Rolle [1].

8.1 Wozu CI/CD?

Durch die fortlaufende Weiterentwicklung oder Fehlerbehebung in der Codebasis kann es zu den in Kapitel 7 beschriebenen Regressionsfehlern kommen. Eine Regression wird dabei beschrieben als „[...] a specific type of bug or issue that occurs when new code changes, like software enhancements, patches, or configuration changes, introduce unintended side effects or break existing functionality that was working correctly before.“ [8].

Demnach sollten nach jeder Änderung in der Implementierung alle geschriebenen GUI- und Unit-Tests ausgeführt werden. Bei einem fehlschlagenden Test wird festgestellt, dass entweder der neu geschriebene Code noch fehlerhaft ist oder es an einer anderen Stelle zu einer Regression gekommen ist.

Um nicht mehr kontinuierlich manuell alle Tests ausführen zu müssen, was die Produktivität des Entwickelnden beeinträchtigt, wird eine CI/CD Pipeline aufgebaut, die das

Bauen und Ausführen der Tests automatisch übernimmt.

8.2 Einbindung in den Prototypen

Die für die Kontroll-Applikation gebaute CI/CD-Pipeline baut unter anderem die Android-Applikation sowie die geschriebenen Tests, damit diese automatisch ausgeführt werden können.

1. Im zentralen git-Repository sind beim Hochladen von neuem Code oder beim Anlegen einer Pull Request Aktionen hinterlegt, welche automatisch ausgeführt werden. Es wird bei der automatischen Bauumgebung ein Webhook aufgerufen, wodurch der automatische Bau- und Test-Prozess beginnt. Erst nach erfolgreichem Durchlaufen der Tests wird die Pull Request als genehmigt markiert und kann zusammengeführt werden.

2. In der automatischen Bauumgebung wird durch einen Webhook der Bauprozess gestartet. Dieser nutzt einen vorgebauten Docker-Container als Grundlage, um darin das Bauen zu beginnen. Der Container beinhaltet alle notwendigen Bau-Tools zum Erstellen und Testen der Kontroll-Applikation, da nicht sichergestellt werden kann, dass in der Bauumgebung alle notwendigen Tools installiert sind.

3. Zum Schluss wird noch mithilfe von SonarQube eine statische Code-Analyse durchgeführt. Zusätzlich überprüft SonarQube die Testabdeckung mit der Erwartung, dass mindestens 80% des geschriebenen Codes durch Tests abgedeckt sind. Die Kennzahl ist eine IVU-interne Vorgabe und muss demnach eingehalten werden.

4. Nachdem alle Tests erfolgreich ausgeführt wurden, wird die gebaute Kontroll-Applikation noch auf einem Datei-Server zusammen mit der Software Bill of Material (SBOM) hochgeladen. Die SBOM ist nicht notwendig, jedoch ermöglicht sie Dependency-Tracking, um automatisch Risiken in Abhängigkeiten zu erkennen. Die hochgeladene Kontroll-Applikation kann anschließend dazu genutzt werden, um die Software auf verschiedene Geräte aufzuspielen.

5. Bei einem Fehlschlag in einer der vorherigen Schritte wird eine Mail an den Ersteller der letzten Änderung des Code-Anteils gesendet, die die Fehlerursache kurz wiedergibt.

Die gebaute Kontroll-Applikation kann bei Bedarf heruntergeladen werden. Auch das Ergebnis der Sonar-Analyse ist in einer SonarQube-Installation einzusehen.

In Zukunft wird es zusätzlich möglich sein, die Kontroll-Applikation direkt in den App-Store der entsprechenden Plattform hochzuladen. Die beschriebene Pipeline würde dann erweitert werden, um die Kontroll-Applikation in den App-Store hochzuladen. Dadurch wird gewährleistet, dass die Kontrolleure immer die aktuelle Version der Kontroll-Applikation zum Kontrollieren der Reisenden nutzen, ohne dass manuelle Eingriffe von weiteren beteiligten Personen nötig sind.

9 Fazit

In dieser Arbeit wurde ein Prototyp einer mobilen Kontroll-Applikation für ABT entwickelt. Die Analyse zeigt, dass ABT neue Herausforderungen für die Kontrolle von Fahrberechtigungen mit sich bringt. Durch die Integration der TOKA-App konnte die Generierung eines Identitätstokens realisiert werden, das als Basis für die Prüfung im Hintergrundsystem dient. Die Umsetzung mit KMP und CMP erwies sich als effizient und zukunftssicher, da sie eine plattformübergreifende Entwicklung ermöglicht. Ergänzend wurden automatisierte Tests und eine CI/CD-Pipeline implementiert, um Qualität und Skalierbarkeit sicherzustellen. Der Prototyp erfüllt die wesentlichen Anforderungen und ist nahezu produktionsreif. Für die Zukunft bietet sich die Erweiterung auf iOS-Geräte sowie die Integration zusätzlicher Prüfmechanismen (z. B. räumliche Gültigkeit) an. Insgesamt zeigt die Arbeit, dass eine moderne Kontroll-Applikation für ABT technisch umsetzbar ist und eine wichtige Grundlage für die Digitalisierung im ÖPV bildet.

Abkürzungsverzeichnis

Abbildungsverzeichnis

Literatur

[1] Best CI/CD platforms for Enterprise: Top 8 solutions in 2025 — octopus.com. https://octopus.com/devops/ci-cd/ci-cd-tools-for-enterprise/. [Accessed 20-10-2025].

[2] Compose Multiplatform 1.8.0 Released: Compose Multiplatform for iOS Is Stable and Production-Ready. https://blog.jetbrains.com/kotlin/2025/05/compose-multiplatform-1-8-0-released-compose-multiplatform-for-ios-is-stable-and-#production-ready-and-easy-to-adopt. [Accessed 03-11-2025].

[3] Compose Multiplatform and Jetpack Compose — Kotlin Multiplatform — jetbrains.com. https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-and-jetpack-compose.html#jetpack-compose-and-composables. [Accessed 31-10-2025].

[4] Compose Multiplatform and Jetpack Compose — Kotlin Multiplatform — jetbrains.com. https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-and-jetpack-compose.html#technical-details. [Accessed 03-11-2025].

[5] Robolectric — robolectric.org. https://robolectric.org/. [Accessed 17-11-2025].

[6] Was ist ein eticket - und wie funktioniert es? https://www.eticket-deutschland.de/ticketing. [Accessed 28-11-2025].

[7] Account based ticketing: Chancen und herausforderungen der ticketing-technologie. https://www.eticket-deutschland.de/magazin/account-based-ticketing-chancen-und-herausforderungen-der-ticketing-technologie/, March 2023. [Accessed 21-11-2025].

[8] Regression Testing: Definition, Types, and Tools — github.com. https://github.com/resources/articles/regression-testing-definition-types-and-tools, 2024. [Accessed 20-10-2025].

[9] Autonomes Fahren im ÖPNV: „Robo-Bus-Radar 2025“ zeigt Chancen und Herausforderungen — zukunftsnetzwerk-oepnv.de. https://www.zukunftsnetzwerk-oepnv.de/aktuelles/news/autonomes-fahren-im-oepnv-robo-bus-radar-2025-zeigt-chancen-und-herausforderungen 2025. [Accessed 15-10-2025].

[10] Client engines — Ktor — ktor.io. https://ktor.io/docs/client-engines.html#platforms, 2025. [Accessed 15-11-2025].

[11] Compose Multiplatform and Jetpack Compose — Kotlin Multiplatform — jetbrains.com. https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-and-jetpack-compose.html#compose-multiplatform-and-jetpack-compose-features, 2025. [Accessed 03-11-2025].

[12] Dependency injection in Android — App architecture — Android Developers — developer.android.com. https://developer.android.com/training/dependency-injection?hl=en#what-is-di, 2025. [Accessed 31-10-2025].

[13] Expected and actual declarations — Kotlin Multiplatform — jetbrains.com. https://www.jetbrains.com/help/kotlin-multiplatform-dev/multiplatform-expect-actual.html#rules-for-expected-and-actual-declarations, 2025. [Accessed 31-10-2025].

[14] Stability of supported platforms — Kotlin Multiplatform — jetbrains.com. https://www.jetbrains.com/help/kotlin-multiplatform-dev/supported-platforms.html#

current-platform-stability-levels-for-the-core-kotlin-multiplatform-technology, 2025. [Accessed 30-10-2025].

[15] Alliance, S. T. Account-based ticketing: some basic common insights. https://www.smart-ticketing.org/_files/archives/fcd3b9_f40c7bddc26843279feee1ad02f0c134.zip?dn=ABT%20Onepager%20Management%20Summary.zip, September 2024.

[16] Alliance, S. T. Account-based ticketing: some opportunities and challenges. https://www.smart-ticketing.org/_files/archives/fcd3b9_f40c7bddc26843279feee1ad02f0c134.zip?dn=ABT%20Onepager%20Management%20Summary.zip, November 2024.

[17] Herschel, M. Eine einführung in die regressionstests. Ausarbeitung für projekt- und qualitätsmanagement im masterstudiengang „angewandte informatik“, Hochschule Hannover, Fakultät IV, Abteilung Informatik, December 2020.

[18] Heyer, S. Die Komplexität des Ticketkaufs im heutigen ÖPNV: Eine Schritt-für-Schritt-Übersicht — linkedin.com. https://www.linkedin.com/pulse/die-komplexit%25C3%25A4t-des-ticketkaufs-im-heutigen-%25C3%25B6pnv-eine-sascha-heyer-7gzhe/, 2024. [Accessed 15-10-2025].

[19] Kousar, A., Khan, S. U. R., Mashkoor, A., and Iqbal, J. A systematic literature review on graphical user interface testing through software patterns. *IET Software 2025*, 1 (2025), 9140693.

[20] Köller, D., Mehlich, M., and Omers, M. Weißbuch „account based ticketing“ – vorschläge für ein gemeinsames vokabular in der debatte. *Der Nahverkehr 12* (2024), 16–19. Homepageveröffentlichung genehmigt für www.rms-consult.de.

[21] Oberlerchner, M. Die Test(automations)pyramide: ein einfaches Gebilde voller Missverständnisse — austriantestingboard.at. https://www.austriantestingboard.at/die-testautomationspyramide-ein-einfaches-gebilde-voller-missverstaendnisse/, 2023. [Accessed 21-11-2025].

[22] Poenisch, M. Die Testpyramide — openknowledge.de. https://www.openknowledge.de/blog/die-testpyramide, August 2022.

[23] Schwichtenberg, D. H. Mock-Objekt - Begriffserklärung im Entwickler-Lexikon/Glossar auf www.IT-Visions.de — it-visions.de. https://www.it-visions.de/glossar/alle/3923/MockObjekt.aspx. [Accessed 02-12-2025].

[24] Team, A. Modern Android Dev: Jetpack Compose vs. XML (2025) — androshelf.com. https://androshelf.com/blogs/jetpack-compose-vs-xml.html, 2025. [Accessed 30-10-2025].

[25] Wolff, O. Stellungnahme zur anhörung des ausschusses für verkehr und digitale infrastruktur des deutschen bundestages – bundesweites/digitales ticketing. Webkonferenz, May 2020. Ausschussdrucksache 19(15)352-A, 71. Sitzung.