Technische Hochschule Rosenheim

Fakultät für Informatik

Informatik Master

Cross Plattform Apps mit Flutter und Dart

Projektarbeit

von

Simon Treutlein, Tobias Lautenschlager, Serdar Polat

Datum der Abgabe:	17.01.2020
Erstprüfer:	Herr Frai
Zweitprüfer:	Herr Prof. Dr. Beneken

Inhaltsverzeichnis

Abbildungsverzeichnis i
Tabellenverzeichnis ii
1 Einleitung 1
2 Plattformübergreifende Entwicklung 1
2.1 Flutter 1
2.2 Vergleich mit Nativer Entwicklung 3
2.3 Cross-Platform Entwicklung 3
2.4 Fazit 5
3 Entwicklungsprozess 6
3.1 Entwicklungstools 6
3.1.1 Android Studio und IntelliJ 6
3.1.2 VS-Code 7
3.1.3 DevTools 7
3.1.4 Hot Reload 7
3.2 Debuggingtools 8
3.3 Testtools 9
4 Laufzeit- und Deploymentmodell 9
4.1 Allgemeine Informationen zu Dart 9
4.2 Laufzeitmodell 10
4.3 Deployment 11
5 Architektur 12
5.1 Aufbau einer Flutter App 13
5.2 Aufbau des Prototypen 13
5.3 Handling von Netzwerk-Requests 14
6 Schluss 16
Literaturverzeichnis 17

Abbildungsverzeichnis

Abbildung 1: Systemarchitektur 2
Abbildung 2: Xamarin Architektur 4
Abbildung 3: Multi OS Engine 5
Abbildung 4: Android Studio Plugins 6
Abbildung 5: Flutter Timeline Screen 7
Abbildung 6: Context Switch 11
Abbildung 7: Ordnerstruktur der Anwendung 13
Abbildung 8: Ablauf der Anwendung 14
Abbildung 9: Netzwerkaufruf 15

Tabellenverzeichnis

Tabelle 1: Vergleich Nativ und Cross-Plattformen 3
Tabelle 2: Vergleich von Cross-Plattform Technologien 6

1 Einleitung

Die Welt der mobilen Kommunikationsgeräte hat sich seit dem Markteintritt des Iphones von Apple stark verändert. Smartphones wurden mit der Zeit immer leistungsfähiger und die Anzahl an Wettbewerbern stieg in diesem Markt stark an. Aktuell haben sich zwei Smartphone Betriebssysteme auf dem Markt stark etabliert. Dies ist zum einen das Android-Betriebssystem mit einem Marktanteil von ca. 75 % (Zeitraum 2018-2019) und zum anderen das Apple iOS-Betriebssystem mit einem Marktanteil von ca. 23 % (Zeitraum 2018-2019).[1] Die restlichen 2 % werden im Rahmen dieser Arbeit vernachlässigt. Das Entwickeln derselben App für beide Betriebssysteme separat ist sehr aufwändig. Deshalb entstanden mit der Zeit Cross Plattform-Technologien, die diesen Entwicklungsaufwand auf die Hälfte reduzieren sollen. Das bedeutet, dass durch die Entwicklung mit den Cross Plattform-Technologien eine App nur einmal entwickelt werden muss, damit diese dann beispielsweise auf Android und iOS-Geräten installierbar und benutzbar ist.

Zunächst wird im Rahmen dieser Arbeit allgemein erläutert was plattformübergreifende Entwicklung ist. Dazu wird für dieses Projekt das Cross-Plattform Framework Flutter verwendet. Um Transparenz für die Entwicklung zu schaffen wird die plattformübergreifende Entwicklung mit der nativen Entwicklung verglichen. Darüber hinaus werden andere Cross-Plattform-Technologien vorgestellt.

Daraufhin werden der Entwicklungsprozess, das Deployment- und Laufzeitmodell, sowie die Architektur von Flutter untersucht. Hierbei wird vereinzelt auf die prototypische Anwendung eingegangen, welche im Rahmen dieser Veranstaltung mit Flutter entwickelt wurde.

Ziel ist es, alle Komponenten zu erläutern, welche nötig sind, um eine Flutter App ausführen zu können. Am Ende der Arbeit soll der Leser ein Gefühl für die Nutzbarkeit von Flutter in realen Projekten erlangt haben.

2 Plattformübergreifende Entwicklung

Bei der Plattformübergreifenden (in Folge: Cross-Plattform) Programmierung kann eine Codebasis so kompiliert werden, dass sie auf verschiedenen Betriebssystemen lauffähig ist.
Aus in Kapitel 1 aufgeführten Gründen wird im Bereich der Smartphones von Cross-Plattform-Entwicklung gesprochen, wenn die Betriebssysteme Android und iOS bedient werden. Im Folgenden wird die Cross-Plattform Entwicklung mit der nativen Entwicklung verglichen. Anschließend wird ein Vergleich verschiedener Cross-Plattform Technologien erstellt und am Ende ein Fazit gezogen.

2.1 Flutter

Flutter ist ein von Google entwickeltes Cross-Plattform Framework, welches aus einem Dart Framework, einer Flutter Engine sowie einer Laufzeitumgebung besteht. Bei Dart handelt es sich um eine ebenfalls von Google entwickelte Programmiersprache, welche für die Flutter-Entwicklung verwendet wird. Sie wird später in Kapitel 4.1 genauer behandelt.

[1] vgl. [stat]

Die Laufzeitumgebung kombiniert dabei die Kern-Bibliotheken von Flutter, wie beispielsweise Animationen und Grafik, Netzwerk I/O und vielen weiteren.[2] Große Vorzüge der Flutter-Entwicklung sind:

- Schnelle Entwicklungszyklen
- UI durch integrierte Material Design Elemente
- Adaption von Plattformspezifischem Scrollen, Navigation, Icons und Schriftarten

Falls Betriebssystem-spezifische APIs noch nicht von Flutter adaptiert worden sind, bietet Flutter die Möglichkeit, plattformspezifischen Code einzubinden. Des Weiteren können bestehende Android oder iOS Apps mit Flutter erweitert werden.

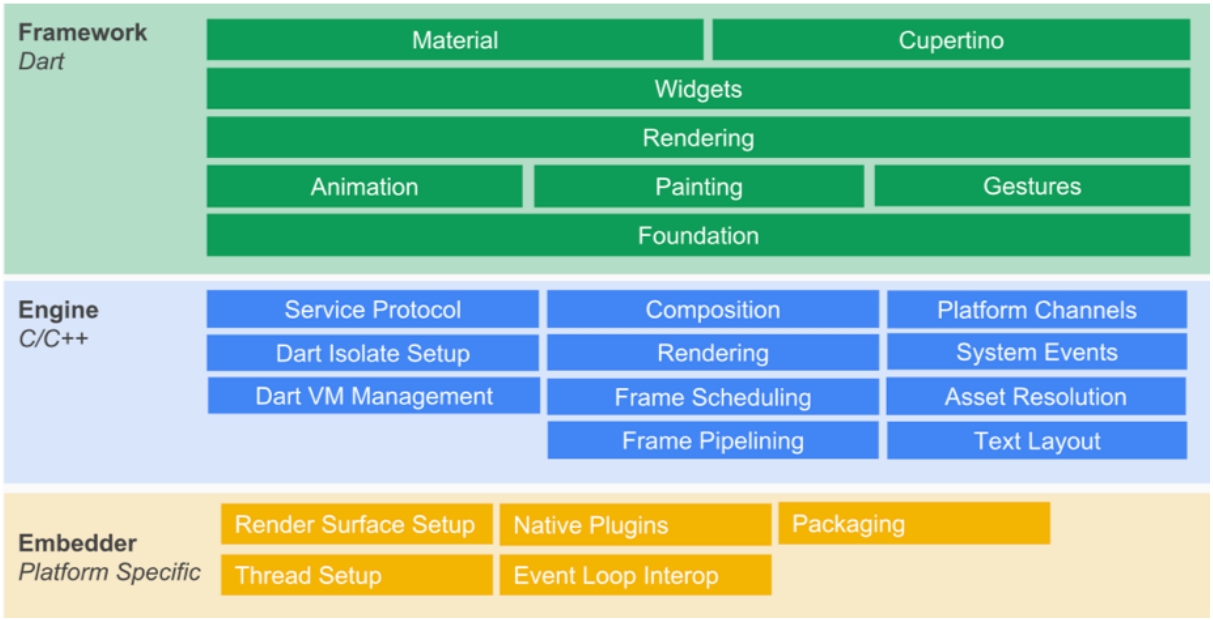

Abbildung 1: Systemarchitektur

Flutter ist mit C, C++, Dart und der Skia Grafik Engine gebaut. In Abbildung 1 ist die Systemarchitektur von Flutter dargestellt. Das Dart Framework mit den zugehörigen Widgets, dem Rendering und der Foundation ist über der Engine, die in C/C++ entwickelt ist. Der Embedder ist beispielsweise für das Thread Setup und die nativen Plugins verantwortlich.

Widgets

Die Flutter UI funktioniert mit Widgets, welche visuell, wie beispielsweise ein Button, oder strukturell, wie beispielsweise eine Reihe oder Spalte, sein können. Jedes Widget besitzt die "build()" Methode, welche letztendlich Änderungen in der UI vornimmt. Widgets werden zwischen "Stateless" und "Stateful" unterschieden. Je nach Art erben sie von der Klasse *StatefulWidget* oder StatelessWidget.

Stateless Widgets haben keinen Zustand, das bedeutet, das Verhalten des Widgets bleibt konstant. Beispiele hierfür sind Icon, Text und Image.

Ein Stateful Widget besitzt zusätzlich einen State (zu Deutsch: Zustand), dessen Felder mit der "setState()" Methode geändert werden können.

[2]

2.2 Vergleich mit Nativer Entwicklung

Native Entwicklung ist das Entwickeln von Applikationen, die auf einer spezifischen Plattform laufen. Für iOS wird mit Objective C oder Swift und der Entwicklungsumgebung XCode, für Android mit Java oder Kotlin und der Entwicklungsumgebung Android Studio nativ programmiert.

Die native Entwicklung bietet verschiedene Vorteile.[3] Aufgrund der Möglichkeit, direkt auf die Betriebssystem-spezifischen APIs zugreifen zu können, ist die Performance im Vergleich zu den meisten Cross-Plattform-Frameworks besser. Außerdem ist die UI bei der nativen Entwicklung einheitlich, wodurch eine konstante Benutzererfahrung gewährleistet ist. Bei der nativen Entwicklung ist der sofortige Zugriff die neuesten Features des Betriebssystems möglich.

Parameter	**Nativ**	**Cross-Plattform**
Kosten	hoch	niedrig
Code-Verwendbarkeit	eine Plattform	mehrere Plattformen
Gerätezugriff	Plattform-SDK versichert Zugriff auf Geräte API	kein versicherter Zugriff auf Geräte API
UI Konsistenz	Volle Konsistenz der UI Komponenten des Gerätes	Limitierte Konsistenz der UI Komponenten des Gerätes
Performance	optimale Performance ohne Probleme	Verzögerungen- und Hardware Kompatibilitätsprobleme

Tabelle 1: Vergleich Nativ und Cross-Plattform[4]

Tabelle 1 bietet einen Überblick der üblichen Vorteile und Nachteile von Cross-Plattform Entwicklung im Vergleich zu nativer Entwicklung. Es ist allerdings anzumerken, dass sich manche Aspekte je nach Technologie abweichen können. Eine grundlegende Aussage zu treffen, bezüglich ob native oder Cross-Plattform Entwicklung besser ist, ist nicht möglich. Für jedes Projekt müssen die Vor- und Nachteile abgewogen werden.

2.3 Cross-Platform Entwicklung

Xamarin

Xamarin ist eine Cross-Platform Technologie, die von Microsoft unterstützt und mit C# und dem .NET Framework entwickelt wird. Xamarin.Forms ermöglicht es, neben Xamarin, eine UI zu bauen. Mit Xamarin kann durchschnittlich 90 Prozent der Codebasis plattformübergreifend genutzt werden.[5]

[3] vgl. [rayg]
[4] vgl. [nets]
[5] vgl. [mic]

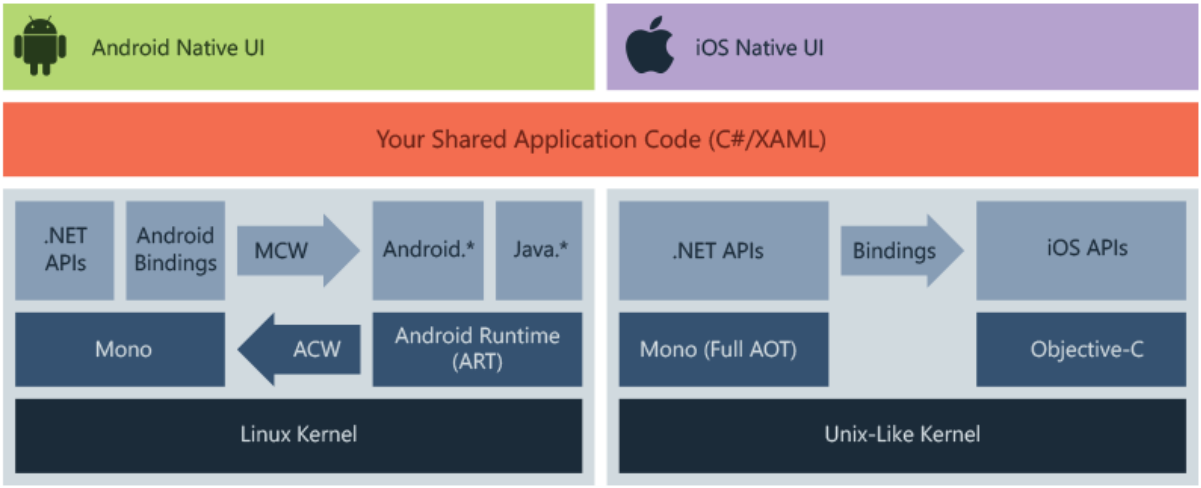

Abbildung 2: Xamarin Architektur[6]

In Abbildung 2 ist die Funktionsweise von Xamarin dargestellt. Der Anwendungscode wird von Android und iOS geteilt und basiert auf XAML oder C#. Xamarin baut auf der Open Source Version des .NET Frameworks 'Mono' auf, basierend auf den .NET ECMA-Standards.[7] Speicherbelegung, Garbage Collection und weitere Aufgaben werden von Mono übernommen. Außerdem bietet Xamarin die folgenden Eigenschaften:

- Vollständige Anbindung des vorliegenden SDKs
- Objective-C, Java, C und C++ Interop
- Moderne Sprachkonstrukte
- Robuste Basisklassen Bibliotheken
- Moderne IDEs
- Mobile Cross-Platform Unterstützung

Xamarin.Forms stellt eine API zur Verfügung, um Benutzeroberflächenelemente auf verschiedenen Plattformen zu erstellen. XAML oder C# kann für die Implementierung verwendet werden und außerdem werden Data Binding Patterns wie beispielsweise Model-View-ViewModel unterstützt.[8]

React Native
React Native ist ein JavaScript Framework, das von Facebook unterstützt wird und die JavaScript Syntax verwendet. React Native ermöglicht die Entwicklung von Android, iOS, Windows und Web Applikationen. React Native kann mit React verglichen werden, benutzt aber native Komponenten anstatt Web Komponenten.

React Native besitzt das sogenannte 'Fast Refresh'. Dieses kann mit dem 'Hot Reload' von Flutter verglichen werden.

Viele internationale Firmen wie beispielsweise Facebook, Instagram, Skype, Tesla und Discord benutzen React Native für die Entwicklung ihrer mobilen Applikationen.[9]

6
7
8
9 [fb]

Multi-OS Engine
Die Multi-OS Engine von Intel ermöglicht die Entwicklung nativer mobiler Applikationen für iOS und Android Geräte durch die Nutzung von Java Funktionen.

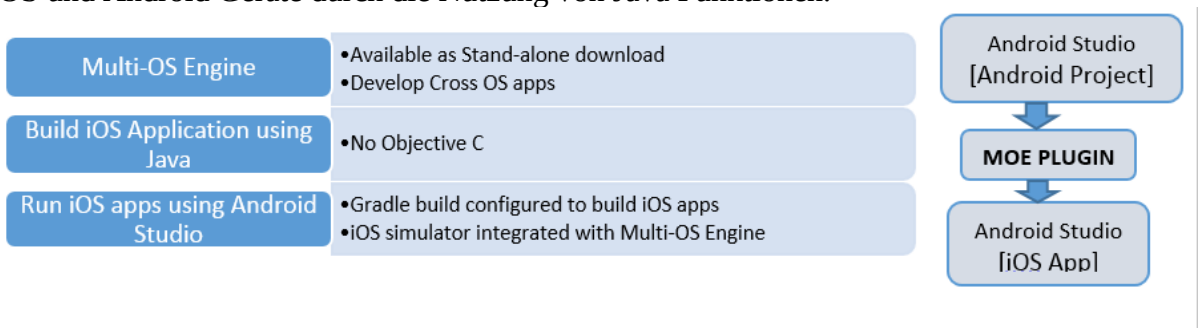

Abbildung 3: Multi OS Engine[10]

In Abbildung 3 ist der Entwicklungsablauf einer Multi-OS Engine Applikation zu sehen. Eine Anwendung startet als Android Studio Projekt. Die Multi-OS Engine konfiguriert anschließend das Projekt, dass eine iOS Applikation in einem Emulator oder auf einem echten Gerät laufen kann.

Die Multi-OS Engine Laufzeitumgebung basiert auf der Android Runtime (ART). ART hat Eigenschaften für optimale Performance der Applikation auf iOS Geräten:

- Ahead-Of-Time (AOT) Kompilierung
- Gleiche Java Runtime Bibliotheken wie Android
- Erweitertes Speichermanagement und Garbage Collection

Flutter
Die Grundlagen von Flutter werden in Kapitel 2.1 beschrieben. Hot-Reload ist ein Vorteil von Flutter um den Entwicklungsprozess zu beschleunigen. Hot Reload injiziert den aktualisierten Quellcode in die laufende Virtuelle Dart Maschine. Nachdem die Virtuelle Maschine die Klassen aktualisiert hat baut das Flutter Framework den Widget Baum automatisch neu, sodass die Änderungen zu sehen sind.[11]

Es gibt noch weitere Cross-Platform Technologien, die in dieser Arbeit nicht behandelt werden. Beispiele hierfür sind PhoneGap, Titanium, Unity, Monocross und Ionic.

2.4 Fazit

Kriterium	Flutter	Xamarin	React Native	Multi-OS Engine
Entwickler	Google	Microsoft	Facebook	Intel
Programmiersprache	Dart	C#	JavaScript	Java
Plattformen	iOS, Android, Web	iOS, Android, Web	iOS, Android, Web, Windows	iOS, Android
Hot Reload	Ja	Ja	Ja	Nein

10
11

Bestehende Apps portieren	Ja	Ja	Ja	Ja

Tabelle 2: Vergleich von Cross-Platform Technologien

In Tabelle 2 ist ein Vergleich der betrachteten Cross-Platform Technologien zu sehen. Keine der vorgestellten Technologien ist perfekt ohne Nachteile. Abschließend lässt sich sagen, dass für das vorliegende Problem die Vor- und Nachteile der verschiedenen Cross-Platform Technologien anhand der Anforderungen verglichen werden müssen.

3 Entwicklungsprozess

Um die Entwicklung einer Flutter-App einfacher zu gestalten, existieren mehrere Tools, die man während dem Entwicklungsprozess verwenden kann. In diesem Kapitel werden Tools für die Entwicklung, zum Testen und zum Debugging erläutert. Darüber hinaus wird noch ein Tool für das Hot-Reloading beleuchtet.

3.1 Entwicklungstools

3.1.1 Android Studio und IntelliJ

Um in einen der beiden IDE's mit Flutter arbeiten zu können bzw. die Programmiersprache Dart verwenden zu können, müssen zuerst in dem Menüpunkt „Settings für das Projekt" die jeweiligen Plugins installiert werden. *(vgl. Abbildung 4)*

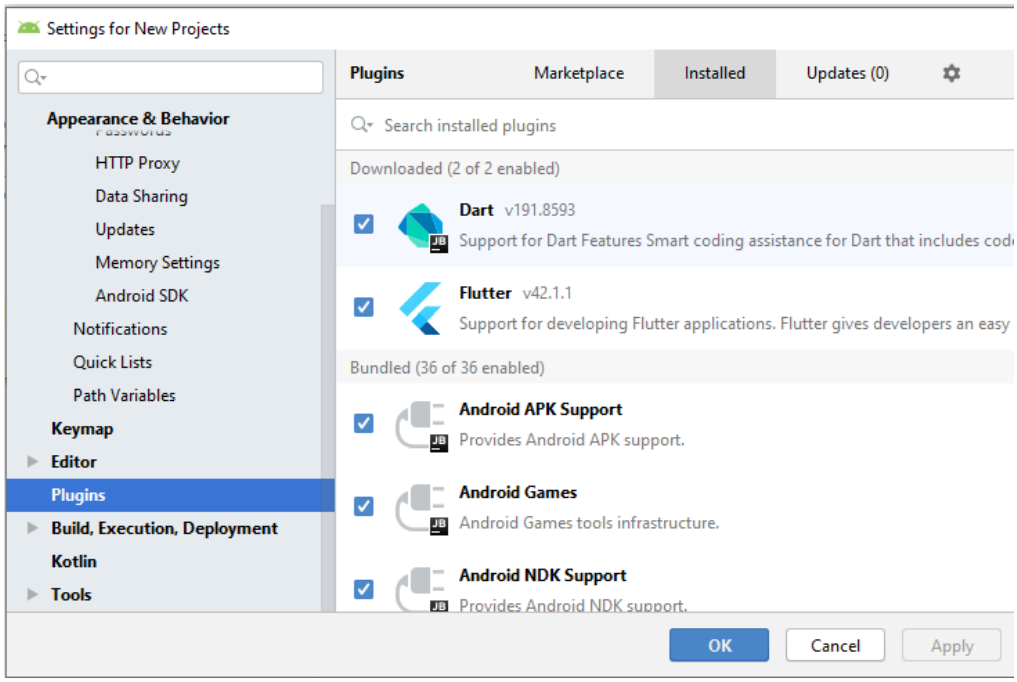

Abbildung 4: Android Studio Plugins

Durch das Dart-Plugin wird eine automatisierte Code-Analyse durchgeführt. Diese ermöglicht es z. B. die Syntax hervorzuheben, den Code durch Rich Type Analyse vorzuschlagen, in die Ursprungsdeklaration zu navigieren oder Verwendungsnachweise von Typen zu finden und zeigt eine Übersicht aller Quellcode Probleme. Darüber hinaus bieten diese Entwicklungsumgebungen einen virtuellen Gerätemanager, mit dem man ein Smartphone

emuliert, um die implementierte Flutter-Anwendung zu testen. Beide Umgebungen bieten außerdem einen Performance Assistenten, mit dem die Bildrate der Applikation gemessen werden kann. Ebenso unterstützen beide IDE's einen Debugger. Auf das Debugging wird in Kapitel 3.2 genauer eingegangen.[12]

3.1.2 VS-Code

VS-Code ist ein Editor, mit indem man das Dart- und Flutter-Plugin installieren kann. Dieser unterstützt fast dieselben Funktionen wie Android Studio und IntelliJ außer z. B. einem integrierten virtuellen Gerätemanager.[13] Generell ist VS-Code deutlich minimalistischer, es können jedoch sämtliche Features bei Bedarf hinzuinstalliert werden.

3.1.3 DevTools

„DevTools ist eine Suite von Performance- und Debugging-Tools für Dart und Flutter."[14] Dieses Tool befindet sich jedoch aktuell noch in der Entwicklung. Das Tool unterstützt die Entwickler folgenden Funktionen:

- Überprüfung des UI-Layouts und den Status einer Flutter-App
- Diagnostiziert Probleme mit der Jank-Leistung der UI in einer Flutter-App
- Debuggen auf Quellcodeebene durch Kommandozeilen Eingabe
- Debuggen von Speicherproblemen durch die Kommandozeile
- Zeigt allgemeine Protokoll- und Diagnoseinformationen durch eine Kommandozeilen Eingabe in der Flutter-Anwendung.[15]

Abbildung 5 zeigt einen kleinen Ausschnitt der Timeline-Ansicht:

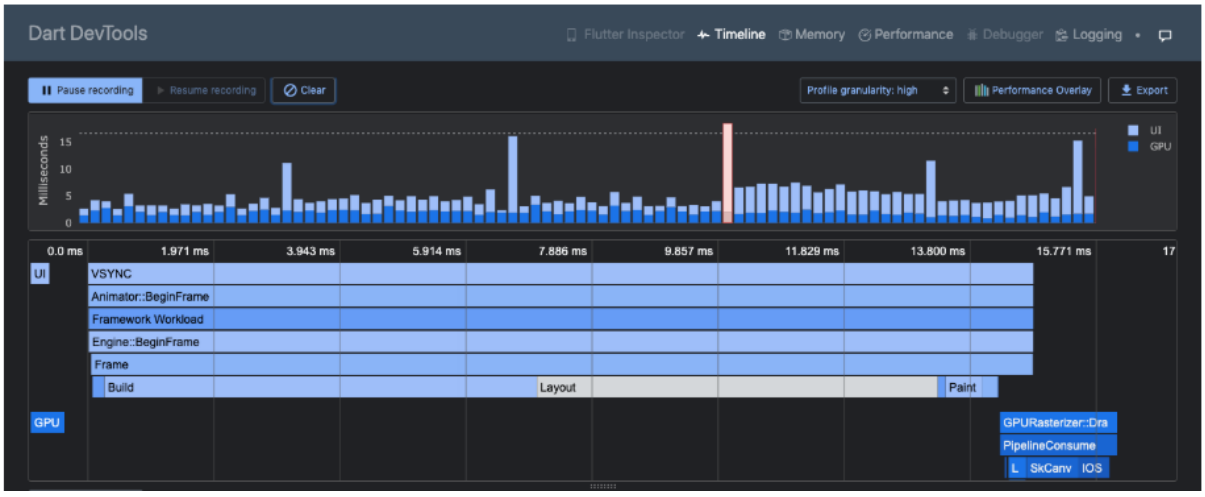

Abbildung 5: Flutter Timeline Screen[16]

3.1.4 Hot Reload

Mit dem „Hot Reload" kann ein Entwickler einfach und schnell Veränderungen wie zum Beispiel Anpassungen in der Benutzeroberfläche, nachvollziehen. Dies geschieht während des

[12] vgl. [Flu] /docs/development/tools/android-studio
[13] vgl. [Flu] /docs/development/tools/vs-code
[14] vgl. [Flu] /docs/development/tools/devtools/overview
[15] vgl. [Flu] /docs/development/tools/devtools/overview
[16] vgl. [Flu] /docs/development/tools/devtools/overview

Programms der sogenannten DartVM läuft und „Just in Time“ kompiliert wird. Details hierzu folgen in Kapitel 4. Durch das Hot-Reloading gelingt es, dass die App beim Entwickeln nur in äußerst seltenen Fällen neu gestartet werden muss, zum Beispiel wenn neue Abhängigkeiten hinzugefügt worden sind. Nur Flutter-Anwendungen im Debug-Modus können „hot reloaded“ werden. Beim starten des Hot-Reloads wird die Anwendung, an der Stelle fortgeführt, an der der Entwickler zuletzt gewesen ist bevor er den Vorgang gestartet hat. Natürlich wird der aktualisierte Code in das aktuelle Debugging geladen. Bei Fehlerhafte Codeänderungen, werden Fehlermeldungen ausgegeben. Um den Hot-Reload wieder ausführen zu können müssen die Fehler behoben werden.

Codeänderungen werden zwar beim Hot-Reload übernommen, jedoch werden statische Felder oder globale Variablen nicht erneut initialisiert. Falls hier Änderungen vorgenommen werden, muss die Anwendung neu gestartet werden um diese zu sehen. Konstante Felder werden wiederum neu initialisiert. Dart will allgemein mit diesem Konzept eine kostspielige Initialisierung bei Programmstart vermeiden.

Der Hot-Reload hat in bestimmten Bereichen Einschränkungen. Dazu gehören Veränderungen innerhalb der initState()-Methode. Wenn diese vorgenommen werden, muss die Anwendung nochmal neu ausgeführt werden. Bei Änderungen von Enumerationen zu Klassen oder Klassen zu Enumerationen oder bei Änderungen eines generischen Typs, erstellt der Hot-Reload Fehlermeldung und die Veränderungen werden nicht ausgeführt.

Der Reload funktioniert, indem beim Ausführen die Flutter Engine den Code, der während der Kompilierung bearbeitet wurde, abändert. Hierzu werden alle Bibliotheken in dem sich der Code geändert hat, die Hauptbibliothek der Anwendung und alle Bibliotheken, die zur Hauptbibliothek führen neu kompiliert. In Dart 2 werden diese Änderungen in einer Kern-Datei umgewandelt und an die virtuelle Maschine geschickt. Die VM lädt alle Bibliotheken neu aus der Kern-Datei. Zum Schluss werden alle vorhandenen Widgets und Renderobjekte neu erstellt.[17]

3.2 Debuggingtools

Zum Debuggen gibt es eine Reihe von Tools, die man für Flutter-Anwendungen verwenden kann. Die eben behandelten DevTools bieten z. B. Performance- und Profiling-Tools. Android Studio/IntelliJ und VS-Code beinhalten einen integrierten Quellcode-Debugger mit dem man Breakpoints setzen kann. Dadurch ist man in der Lage den Code an festgelegter Zeile schrittweise zu durchlaufen, um die Belegung sämtlicher Felder zu untersuchen. Mit dem Flutter-Inspector kann man Widget-Bäume untersuchen, indem man direkt einzelne Elemente aus dem Baum anklicken kann, welche dann wiederum in der App hervorgehoben werden. Diese Funktionalität wird ebenfalls von DevTools und den anderen Entwicklungsumgebungen bereitgestellt. Mit dem Dart-Analysator kann man den Code auf mögliche Fehler überprüfen. Die Protokollierung ist ein weiteres Debugging-Tool, welches Entwickler bei der Softwareentwicklung unterstützt. Hierbei kann man programmgesteuert Protokollnachrichten implementieren, die dann beispielsweise in der Konsole ausgegeben werden. Selbst Animationen können in Flutter-Anwendungen mit dem Debugger untersucht werden. Der

[17] vgl. [Flu] /docs/development/tools/hot-reload

Flutter-Inspector bietet die Möglichkeit, Animation zu verlangsamen. Darüber hinaus kann man die Startzeit durch eine Kommandozeileneingabe messen. Die Messung wird dabei in einer JSON-Datei gespeichert. Gemessen werden zum Beispiel die Dauer der Initialisierung des Flutter-Frameworks, die Dauer, die beim Rendern des ersten Frames beansprucht wird oder auch die Zeit, die benötigt wird um auf den Engine-Code zuzugreifen. Alle gemessenen Werte werden in Mikrosekunden erfasst.[18]

3.3 Testtools

Testtools werden benötigt, weil Anwendungen mit der Zeit an Komplexität zunehmen. Oft kann es umständlich sein, große Anwendungen durch manuelle Tests abzusichern. Aus diesem Grund haben sich im Laufe der Zeit automatisierte Tests etabliert. Diese werden unterteilt in Unit-Tests, Widget-Tests und Integrations-Test.

Durch Unit-Tests testet man Klassen und Methoden abgekapselt voneinander. Hierbei prüft man, ob die implementierte Logik bei unterschiedlichen Bedingungen funktioniert. Dadurch kann man im Falle von Logikfehlern den Code schneller an getesteten Bereichen korrigieren. Unit-Tests laufen automatisch beim Ausführen der Anwendung durch, ohne dass ein Benutzer eine Eingabe durchführen muss.

Widget-Tests testen einzelne Widgets. Dadurch soll geprüft werden ob eine Benutzeroberfläche richtig aussieht und wie gewünscht funktioniert. Das Testen von Widgets ist umfangreich und betrifft mehrere Klassen. Das Bedeutet, dass eine Testumgebung verfügbar sein muss, welche den passenden Lebenszykluskontext des Widgets bereitstellt.

Integrationstests können einen Teil oder die vollständige Anwendung abdecken. Hier wird überprüft, ob alle Widgets und Dienste wie definiert zusammenarbeiten. Zusätzlich kann man mit diesem Test auch die Leistung der Anwendung überprüfen. Integrationstest werden für gewöhnlich auf echten Geräten oder gekapselt auf Emulatoren durchgeführt um Ergebnisse nicht zu verzerren.[19]

4 Laufzeit- und Deploymentmodell

In Hinblick auf das Deployment- und Laufzeitmodell muss initial untersucht werden, wie die zugrunde liegende Programmiersprache kompiliert. Im Zuge dessen wird zunächst die Sprache *Dart*, mit welcher Flutter Applikationen entwickelt werden, vorgestellt.

4.1 Allgemeine Informationen zu Dart

Dart wird von Google entwickelt, ebenso wie das Flutter Framework selbst. Die Programmiersprache ist bereits im Flutter SDK enthalten und muss nicht separat installiert werden. Dies hat als positiven Seiteneffekt, dass bei einer neuen Flutter Version die aktuellste Dart Version enthalten ist. Dart ist eine objektorientierte Programmiersprache mit Garbage Collection. Ursprünglich wurde Dart entwickelt um einige Mängel der Sprache JavaScript zu beheben. Somit sollte das Entwickeln von großen Anwendungen erleichtert werden.[20]

[18] vgl. [Flu] /docs/testing/debugging
[19] vgl. [Flu] /docs/testing
[20] vgl. [Cla]

Heute hat sich Dart zu einer sehr flexiblen Sprache entwickelt. Die Syntax ist vergleichbar mit Sprachen wie beispielsweise Scala.[21] Außerdem ist die Kompilierung zu nativem JavaScript möglich.

4.2 Laufzeitmodell

Ein großer Grund, wieso sich Google bei der Wahl nach einer Sprache für ihre neue Cross-Platform-Technologie für Dart entschieden hat, ist, weil Dart optional typisiert ist. Das bedeutet, dass die Sprache sowohl typsicher geschrieben werden kann, als auch – wie z.B. bei JavaScript – dynamisch typisiert werden kann. Um nachvollziehen zu können, inwiefern sich das auf die Runtime auswirkt, werden kurz die Charakteristika von typsicheren Sprachen (auch gängig als statische Sprachen) und dynamisch typisierten Sprachen (dynamische Sprachen) erläutert:

- Typsichere (=statische) Sprachen werden AOT („Ahead of time“) kompiliert. Das heißt, dass noch vor Programmstart ein Compiler den Code in nativen Maschinencode (Assembly) übersetzt, welcher direkt vom Prozessor ausgeführt werden kann. Dies hat zur Folge, dass bei einem typischen Entwicklungsprozess, bei dem ein Entwickler regelmäßig das Programm neu startet durch den Compiler Wartezeiten auftreten. Sobald das Programm dann allerdings startet, ist die Performance gut und stabil, da der gesamte Code bereits kompiliert ist. Nativer Code, der direkt auf der Maschine ausgeführt (also Java/Kotlin für Android Applikationen und Objective-C/Swift für iOS), muss in der Regel AOT kompiliert werden.[22]

- Dynamisch typisierte (= dynamische) Sprachen werden JIT („Just in time“) kompiliert. Im Gegensatz zu AOT kompilierten Sprachen starten JIT kompilierte Sprachen sofort, haben allerdings eine merkbare Verzögerung bis Code ausgeführt wird, da der JIT-Compiler den Code während der Laufzeit kompilieren muss.[23]

Zusammenfassend kann also gesagt werden, dass Dart die Vorteile der beiden Arten an Kompilation vereint: Während dem Programmieren kann ein Softwareentwickler durch *Just in Time Kompilation* (welche durch eine Dart VM ermöglicht wird) den sogenannten *Hot Reload* nutzen, durch welchen sich die App innerhalb von wenigen hundert Millisekunden aktualisieren lässt und die Entwicklungszyklen kürzt. Gleichzeitig kann bei Release Versionen auf AOT-Kompilation gesetzt werden. Der generierte ARM Code sorgt für schnelle Startup-Zeiten und stabile Performanz der Apps.

Native Performanz im Detail

Die Fähigkeit AOT zu kompilieren unterscheidet Flutter ganz maßgeblich von anderen Cross-Platform Technologien. React Native beispielsweise nutzt JavaScript als Programmiersprache, die App soll aber letztendlich auf nativen Threads des Geräts ausgeführt werden. So entstehen zwei “Bereiche”, wie auf Abbildung 6 zu sehen ist.

[21] vgl. [Cla]
[22] vgl. [Ste]
[23] vgl. [Ste]

Abbildung 6: Context Switch

Wenn nun beide Bereiche durch eine "Brücke" ("Context Bridge") kommunizieren, tritt aufgrund der verschiedenen Umgebungen ein *Kontext Wechsel* auf, worunter die Performanz leidet und das Risiko besteht, dass der App Zustand verloren geht.[24]

Da Dart direkt zu Maschinencode kompilieren kann, tritt dieses Phänomen bei einer AOT kompilierten App nicht auf, was letztendlich bedeutet, dass Flutter Applikationen den nativ in Android und iOS Entwickelten performanz-technisch keinerlei unterlegen sind.

Alternative Laufzeitumgebungen

Flutter ist in seinem Einsatzgebiet nicht ausschließlich auf mobile Applikationen beschränkt. Obwohl der Fokus auf Android und iOS Apps liegt, wird Flutter bereits für weitere Plattformen evaluiert:

- **Flutter im Web:** Wie zu Beginn des Kapitels bereits erwähnt, besitzt Dart die Möglichkeit, zu nativen JavaScript zu kompilieren, was Dart für Webbrowser lauffähig macht und dem Grundgedanken einer Cross-Platform-Technologie weiter bestärkt. Unter dem Projektnamen *Hummingbird* können bereits Webapps mit Flutter gebaut werden. Dies geschieht durch eine abgewandelte Systemarchitektur. Anders wie auf Abbildung 1 in Kapitel 2.1, kann direkt unter der Framework Schicht (welche zu nativen JS kompiliert) direkt der Browser angesetzt werden.[25] Es ist anzumerken, dass sich Flutter für das Web noch im Beta Stadium befindet.

- **Flutter auf Embedded Geräten:** Die eben erwähnte Systemarchitektur kann nicht nur für Browser angepasst werden, sondern theoretisch für jedes Gerät, wenn ein eigener Embedder (die unterste Schicht auf Abbildung 1) für die jeweilige Plattform geschrieben wird.[26] Auf diese Weise wird Flutter bereits für das Display des Google Home Hub verwendet und wurde experimentell auf einem Raspberry Pi laufen gelassen.[27]

4.3 Deployment

Wenn eine Flutter App aus der IDE heraus gestartet wird, baut Flutter standardmäßig eine *Debug* Version der App. Bei dem Bauen einer *Release* Version, wenn die App beispielsweise in die Stores geladen werden soll, unterscheidet sich das Vorgehen nur geringfügig zu einem herkömmlichen Release. Die Flutter Dokumentation bietet hierfür zwei detaillierte

[24] vgl. [Lelb]
[25] vgl. [Flu]
[26] vgl. [Emb]
[27] vgl. [Emb2]

Anleitungen, um auf Android[28] und iOS[29] Geräten zu releasen. Folgend werden einige wichtige Punkte daraus zusammengetragen:

Release Version bauen

Mithilfe der Kommandozeile können innerhalb der Projektstruktur mit den Befehlen "`flutter build apk --split-per-abi`" (für Android) und "`flutter build ios`" (iOS) Release Versionen gebaut werden. Da die Ordnerstruktur einer Flutter Anwendung standardmäßig einen *Android* und *iOS* Ordner vorsieht, werden die build-Dateien im jeweiligen Ordner generiert. Wenn beim Bauen der Android APK der Zusatz *--split-per-abi* weggelassen wird, wird eine *Fat APK* gebaut, die den Code für alle verfügbaren ABI generiert - das generierte Paket ist folglich größer. Es ist zu erwähnen, dass Flutter ebenfalls *Flavoring*[30] und *Continuous Deployment*[31] unterstützt.

Hochladen in die Stores

Das Hochladen in die Stores geschieht genauso wie das Hochladen einer unabhängigen Android und iOS App hochladen. Bevor die App in die jeweiligen Stores geladen wird, ist zu überprüfen, ob die Versionsnummer innerhalb der "pubspec.yaml" Datei aktuell ist, da die App ansonsten abgelehnt wird.

Soll vor dem richtigen Release noch eine Testphase angestellt werden, können im Google Playstore offene, geschlossene und interne Tests eingerichtet werden.[32] Für iOS kann *Testflight* verwendet werden.

5 Architektur

In diesem Kapitel wird die Architektur der prototypischen Anwendung untersucht, welche im Rahmen dieser Veranstaltung entwickelt wurde. Vorerst wird allerdings auf die standardmäßige Ordnerstruktur eingegangen, welche Flutter Applikationen besitzen.

[28] vgl. [Flu] /docs/deployment/android
[29] vgl. [Flu] /docs/deployment/ios
[30] vgl. [Flu] /docs/deployment/flavors
[31] vgl. [Flu] /docs/deployment/cd
[32]

5.1 Aufbau einer Flutter App

Die Projektstruktur einer Flutter Applikation sieht folgende Komponenten vor:

- Der **lib Ordner** enthält den eigentlichen Quellcode der Applikation. Er wird mit einer *main.dart* Klasse vorinitialisiert.

- Die **pubspec.yaml** Datei enthält sämtliche Konfigurationsdaten der App. Hier werden der Appname, die Versionsnummer, sämtliche Abhängigkeiten sowie Pfade für Assets hinterlegt.

- Der **test Ordner** kann Widget Tests enthalten um Funktionalitäten zu sichern.

- Die **android und ios Ordner** werden benötigt, da beim Ausführen einer Flutter App (abhängig von der Plattform) entweder ein Gradle oder Xcode build innerhalb dieser Ordner ausgeführt wird um die App zu starten. Hier können außerdem plattformspezifische Änderungen wie z.B. Berechtigungen der App geändert werden. Ebenso können die Ordner native Komponenten enthalten, die über sogenannte *Method Channel* angesteuert werden können.

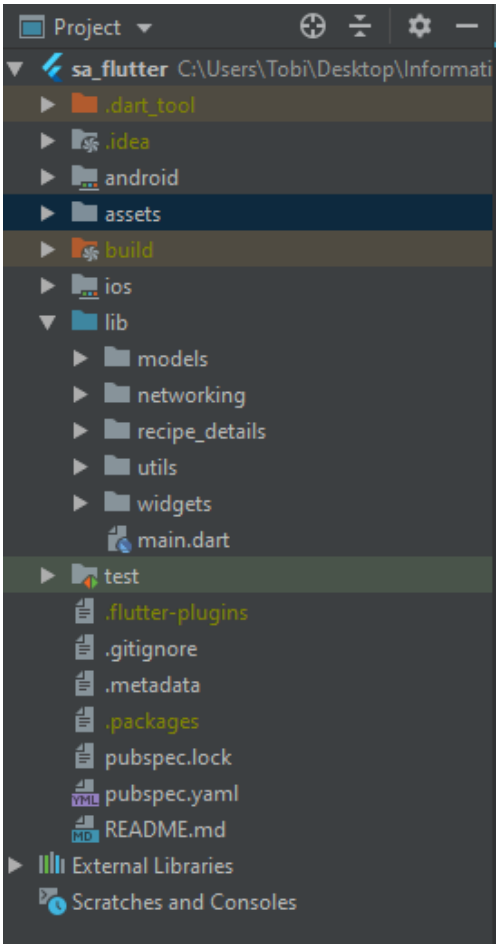

Abbildung 7: Ordnerstruktur der Anwendung

5.2 Aufbau des Prototypen

Der Grundgedanke des Prototypen ist es, die API von spoonacular.com zu nutzen, um die Suche nach Speisen zu ermöglichen und sich die Rezept-Schritte dazu anzeigen lassen zu können. Der Ablauf beim Nutzen der App ist wie folgt:

1. Beim Öffnen der App landet der Nutzer im Bildschirm der **Rezept-Suche**. Hier kann er einen Suchbegriff für eine Speise eingeben und mit Betätigung des *Search*-Buttons die Suche anstoßen.

2. Sobald die App Suchergebnisse erhalten hat, wird eine **Liste an Suchtreffern** angezeigt. Jeder Eintrag besitzt hierbei ein Bild als Preview, sowie Titel und Informationen zu der Kochzeit und ergebenden Portionen. Der Nutzer wird hierfür nicht auf eine weitere Seite weiter navigiert, sondern die bestehende Seite aktualisiert sich. Aus diesem Schritt kann der Nutzer entweder weiter in die **Rezept-Details** eines Treffers oder durch den Button oben rechts in der Appbar zurück zur **Rezept-Suche**.

3. In der **Rezept-Details-Ansicht** bekommt der Nutzer eine Schritt-für-Schritt Anleitung für die Zubereitung des jeweiligen Rezepts. Hierbei sieht er zu jedem Rezept-Schritt die Beschreibung und die dazu benötigten Geräte, Küchenutensilien und Zutaten als kleine Bilder.

Der Ablauf ist in Abbildung 8 noch einmal verdeutlicht:

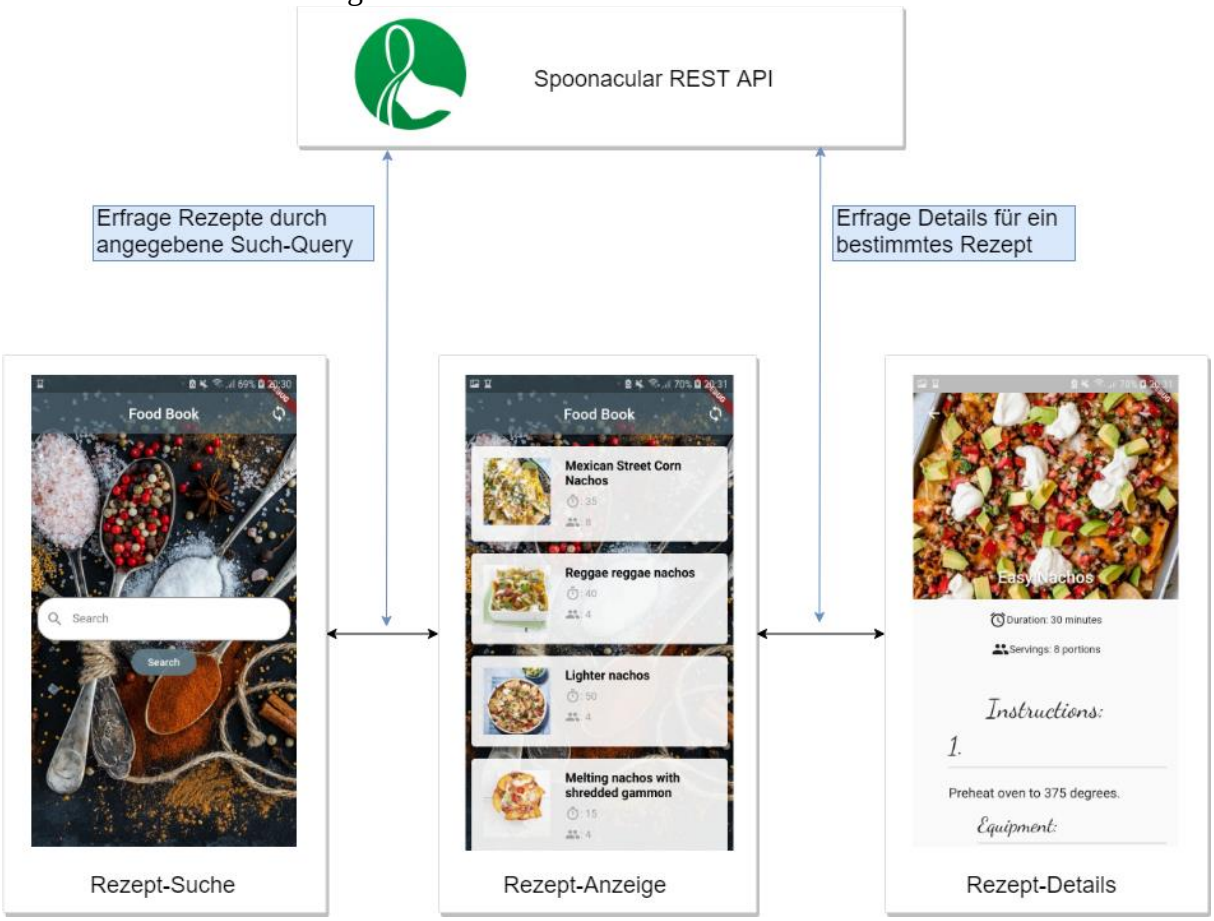

Abbildung 8: Ablauf der Anwendung[33]

5.3 Handling von Netzwerk-Requests

Die blauen Pfeile in Abbildung 9 kennzeichnen die Kommunikation zwischen der App und der REST API von spoonacular.com. Um einen Netzwerk-Request durchzuführen sind im Prototyp zwei Klassen implementiert, **ApiClient** und **ApiRouter**. Die folgende Abbildung 9 verdeutlicht ihr Zusammenspiel:

[33] Selbst erstellte Grafik

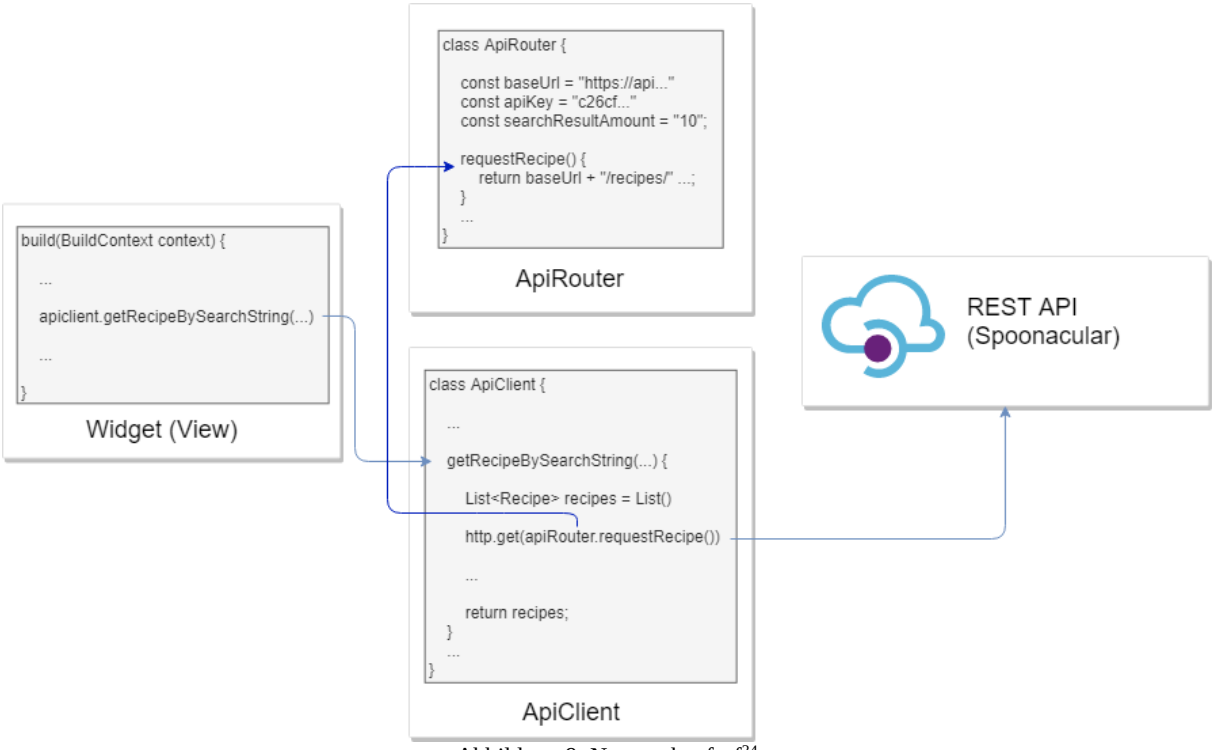

Abbildung 9: Netzwerkaufruf[34]

Die **ApiClient** Klasse stellt Methoden zur Verfügung, welche aufgerufen werden können um eine Liste an Ergebnissen zu einem Suchbegriff oder um die Details zu einem Rezept von der Spoonacular API zu erhalten. Hierfür führt sie erst einen HTTP GET Request aus, um Daten von der API zu bekommen. Die Query wird hierbei von der Klasse **ApiRouter** erstellt, welche verschiedene Parameter (wie den API Key und die Anzahl an gewollten Suchtreffern), als auch die API Endpunkte als Konstanten hinterlegt hat. Anschließend parst der ApiClient die Response zu Objekten, welche als Kärtchen in Form von Suchergebnissen angezeigt werden können und gibt diese zurück in die aufrufende View.

[34] Selbst erstellte Grafik

6 Schluss

In dieser Arbeit wurde die Cross-Platform Technologie Flutter vorgestellt. Verschiedene Cross-Platform Technologien wurden mit Flutter verglichen und ein Fazit gezogen. Jeder der vorgestellten Technologien hat Schwächen und Stärken, sodass für jedes Projekt eine Abwägung getroffen werden muss.

Anschließend wurde der Entwicklungsprozess einer Flutter Applikation betrachtet. Hierbei wurden die Entwicklungs-Tools und Entwicklungsumgebungen genauer analysiert. Des weiteren wurde das 'Hot Reloading', die Debugging Tools und Test Tools erläutert.
Im Anschluss wurde das Deployment- und Laufzeitmodell von Flutter untersucht, wobei intensiv auf die Performance von Flutter eingegangen wird. Außerdem wurde der Build-Prozess und das Hochladen der Applikation in die Stores beschrieben.

Gegen Ende wurde die Architektur von Flutter zum einen im Allgemeinen und zum anderen anhand einer Beispielapplikation, betrachtet. Wie das Handling von Netzwerk Requests in der prototypischen Anwendung implementiert wurde, wurde ebenfalls behandelt.

Abschließend lässt sich aussagen, dass mit Flutter als Cross-Platform Technologie produktiv gearbeitet werden kann. Es ist ratsam, die Anforderungen einer App gründlich zu prüfen, bevor man sich für eine Entwicklung mit Flutter entscheidet. Denn obwohl das Ökosystem an Plugins und Third-Party-Libraries stetig wächst, kann es immer unüblichen Anforderungen geben, für die es keine bereits existierende, etablierte Implementierung gibt. Wenn z.B. 3D-Rendering mit OpenGL gefragt ist, muss dies momentan noch nativ umgesetzt und mithilfe von sogenannten Platform-Channel eingebunden werden, da Flutter durch die Rendering Engine Skia momentan noch auf 2D-Rendering beschränkt ist.

Das Erlernen von Flutter gelingt Dank der deklarativen Erstellung von UI auch ohne Vorkenntnisse in nativer Android- oder iOS-Entwicklung. Auch die Programmiersprache Dart kann nach dem Prinzip „learning by doing" erlernt werden, wenn man bereits Erfahrungen mit anderen objektorientierten Sprachen wie z.B. Java gesammelt hat. Die offizielle Dokumentation unter www.flutter.dev/docs ist sehr ausführlich und gibt teils auch Hilfestellung zu iOS bzw. Android spezifischen Problemen. Die dedizierten Flutter-Entwickler von Google sind trotz der hohen Rate, mit welcher Bugs erstellt werden, stets hilfreich und lassen nicht allzu lange auf eine Antwort warten.

Literaturverzeichnis

[Cla]: G. Clarke. Google shoots Dart at JavaScript. https://www.theregister.co.uk/2011/10/10/google_previews_dart//. Zuletzt aufgerufen: 10.01.2020.

[Ste]: Flutter's Compilation Patterns: https://proandroiddev.com/flutters-compilation-patterns-24e139d14177. Zuletzt aufgerufen: 11.01.2020

[Lelb]: Why flutter uses Dart: https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf. Zuletzt aufgerufen: 11.01.2020

[Flu]: Flutter's Homepage: https://flutter.dev/ Zuletzt aufgerufen 11.01.2020

[Emb]: https://medium.com/flutter/flutter-on-raspberry-pi-mostly-from-scratch-2824c5e7dcb1. Zuletzt aufgerufen 12.01.2020

[Emb2]:https://medium.com/ionicfirebaseapp/flutter-a-portable-ui-framework-for-mobile-web-embedded-and-desktop-ef1e4da1ca8f. Zuletzt aufgerufen 11.01.2020

[Goog]: https://support.google.com/googleplay/android-developer/answer/3131213?hl=de. Zuletzt aufgerufen 17.01.2020

[stat]: Mobile Operating System Market Share Worldwide: https://gs.statcounter.com/os-market-share/mobile/worldwide#monthly-201803-201904-bar, abgerufen am 19.12.2019

[git]: The Engine architecture: https://github.com/flutter/flutter/wiki/The-Engine-architecture, Zuletzt aufgerufen 17.01.2020

[nets]: Where do Cross-Plattform App Frameworks Stand in 2020: https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/, abgerufen am 19.12.2019

[rayg]: Native App Development: https://raygun.com/blog/native-app-development/, abgerufen am 20.12.2019

[mic]: What is xamarin: https://docs.microsoft.com/de-de/xamarin/get-started/what-is-xamarin, abgerufen am 15.01.2020

[fb]: facebook: http://facebook.github.io/react-native/showcase.html, abgerufen am 15.01.2020