Studie av Cross Platform teknik för mobil applikationsutveckling

Emil Kristiansson

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

Sammanfattning

Idag domineras den globala mobilmarknaden av operativsystemen *iOS* och *Android*. *Android* står för över **80%** av marknaden och *iOS* strax under **20%**. I nuläget verkar det inte finnas något utrymme för några konkurrenter [4].

Båda operativsystemen nyttjar separata programspråk och verktyg för utveckling. Detta innebär att man som utvecklare måste göra två olika applikationer om man vill lansera en applikation som ska finnas tillgänglig på hela mobilmarknaden.

I praktiken resulterar det i att en utvecklare måste lägga dubbla resurser vid en utvecklingsprocess.

Under det här examensarbetet har två olika ramverk, som är så kallade *Cross Platform*, undersökts. *Cross Platform* innebär att man endast behöver utveckla en applikation med en kodbas som sedan fungerar på båda operativsystemen. Ramverken som har undersökts är *React Native* och *Flutter*.

Undersökningen har gjorts i samarbete med fintechföretaget *Smart Refill* som utvecklar mobilapplikationer till teleoperatörer, banker och fondbolag. I studien har två enkla prototyp-applikationer byggts i programbiblioteken nämnda ovan (*React Native* och *Flutter*).

Nyckelord: Cross Platform, React Native, Flutter

Abstract

Today the global mobile market are dominated by the two operating systems *iOS* and *Android*. *Android* have over **80%** of the total market and *iOS* just below **20%**. At this moment there are not any other competitor who competes with *iOS* or *Android* [4].

These two operating systems have their own programming language and tools for development. This means that any developer who wants to launch an application for the entire market needs to develop two different applications.

In other words the development process require a double amount of resources.

During this thesis two *Cross Platform* application frameworks have been examined. *Cross Platform* is a technology where a single application with a single code base is compatible with multiple operating systems. The two *Cross Platform* frameworks are *React Native* and *Flutter*.

The thesis has been made in collaboration with the fin-tech company *Smart Refill* that develops mobile applications for Phone-operators, banks and Fund Companies. In the thesis two simple prototype applications have been developed with *React Native* and *Flutter*.

Key words: Cross Platform, react-native, flutter

Förord

På Smart Refill vill jag tacka mina handledare Johannes Ivarsson, Andre Lind och Jonathan Böcker för deras hjälp och vägledning under examensarbetet. Jag vill även tacka min handledare Christin Lindholm för sitt oändliga tålamod samt min examinator Christian Nyberg. Slutligen vill jag tacka min älskade sambo Elise som tagit sig tid att korrekturläsa rapporten flera gånger. Det är äkta kärlek.

Contents

Sammanfattning 2
Abstract 3
Förord 4
1. Inledning 9
1.1. Bakgrund 9
1.2. Syfte 10
1.3. Målformulering 10
1.4. Problemformulering 11
1.5. Motivering av examensarbetet 12
1.6. Avgränsningar 12
2. Teknisk bakgrund 14
2.1. IDE och Visual Studio Code 14
2.2. Native mobilapplikation 15
2.2.1. iOS 15
2.2.2. Android 16
2.3. Cross Platform mobilapplikation 17
2.4. React Native 18
2.5. Flutter 21
2.6. Postman 24
3. Metod 26
3.1. Upplägg 26
3.2. Projektmodell för utveckling 27
3.3. Förstudie 28
3.4. Utveckling 31
3.4.1. Bygga Hello World för respektive ramverk 32
3.4.2. Testa API:et 33

3.4.3. Få Applikationen att hämta data för respektive ramverk.....33
3.4.3.1. React Native.....33
3.4.3.2. Flutter.....34
3.4.4. Design av layouten på applikationen för respektive ramverk.....35
3.4.4.1. React Native.....35
3.4.4.2. Flutter.....38
3.5. Dokumentation.....40
3.6. Handledning på Smart Refill.....41
3.7. Källkritik.....41
4. Analys.....43
4.1. Projektmodell.....43
4.2. Förstudie.....43
4.2.1. Litteraturstudier.....44
4.2.1.1. React Native.....44
4.2.1.2. Flutter.....45
4.2.2. Konceptbild.....46
4.3. Utveckling.....47
4.3.1. React Native.....48
4.3.2. Flutter.....50
4.3.3. Jämförelse av ramverken.....51
4.3.3.1. Behöver man några särskilda verktyg för att kunna utveckla?.....51
4.3.3.2. Vilket programspråk använder man?.....52
4.3.3.3. Hur lång tid tar det för respektive prototyp att kompilera?.....53
4.3.3.4. Är det någon skillnad i filstorlek på prototyperna? 53

4.3.3.5. Vad har andra utvecklare för erfarenhet av respektive ramverk? 53
4.3.3.6. Hur stort är respektive community? 54
4.4. Verktyg 55
4.4.1. Visual Studio Code 55
4.4.2. Postman...55
4.5. Dokumentation 55
5. Resultat...58
5.1. Utvecklingen av prototyperna 58
5.1.1. React Native 59
5.1.2. Flutter 61
5.2. Hur lång tid tar det att hämta data från API:et? ..62
5.2.1. React Native 63
5.2.2. Flutter 63
5.3. Hur skiljer sig prototyperna åt ur ett utvecklarperspektiv?..................................64
5.3.1. Behöver man några särskilda verktyg för att kunna utveckla? 65
5.3.2. Vilket programspråk använder man? 65
5.3.3. Hur lång tid tar respektive prototyp att kompilera?...65
5.3.4. Är det någon skillnad i filstorlek på prototyperna? 66
5.3.5. Vad har andra utvecklare för erfarenhet av respektive ramverk?...66
5.3.6. Hur stort är respektive community?.....................67
6. Slutsats 69
6.1. Reflektioner över etiska aspekter 71
6.2. Framtida utvecklingsmöjligheter 72
7. Källförteckning 73
8. Terminologi 78

8.1.JSON 78
8.2.StackOverFlow 78
8.3.CSS-fil 78
8.4.SCRUM 78
8.5.Asynkron kommunikation...............................79
8.6.Node 79
8.7.JDK 79
9. Appendix 80
9.1.React Native..80
9.2.Flutter 84

1. Inledning

Den här rapporten beskriver ett examensarbete som ingår i utbildningen för högskoleingenjörer i Datateknik vid Lunds universitet, Campus Helsingborg. Examensarbetet har gjorts i samarbete med företaget *Smart Refill*.

1.1. Bakgrund

Smart Refill är ett företag som utvecklar mobila applikationer inom branscherna finans och telekommunikation [1]. Företaget grundades **2006** och började med att leverera en tjänst som gjorde det lättare för mobilanvändare att fylla på sina kontantkort. Idag har *Smart Refill* expanderat till att bli en fullständig *white label mjukvaruleverantör* och applikationsutvecklare till olika aktörer inom Norden, Europa och Asien. Bland dessa kan nämnas *Handelsbanken*, *Resurs Bank*, *Telia* och *Shanghai Commercial & Savings Bank*. Deras huvudkontor ligger i Helsingborg och där jobbar cirka **30** medarbetare. **2017** omsatte företaget ca **668** miljoner kronor och de gjorde en vinst på drygt **8.5** miljoner kronor [2].

Mobilapplikationer från *Smart Refill* utvecklas idag med så kallade *Native* ramverk. Detta innebär att varje applikation bara är utvecklad för en särskild plattform och endast är kompatibel med den plattformen. Med plattform menar man i det här fallet en mobiltelefons operativsystem [3]. *Smart Refill* är intresserade av att gå över till att utveckla applikationer med en annan teknik som kallas för *Cross Platform* ramverk vilket i motsats till *Native* ramverk innebär att en applikation blir kompatibel med flera olika plattformar [4]. *Smart Refill* vill undersöka vad det finns för olika fördelar och nackdelar med att använda sig av *Cross Platform* ramverk. Av dessa är det två stycken särskilda ramverk som är intressanta att undersöka: *React Native* och *Flutter* [5] [6] .

Det som *Smart Refill* efterfrågar är en första studie på dessa två *Cross Platform* bibliotek. Studien går ut på att bygga två prototyper

av mobilapplikationer, en med *React Native* och en med *Flutter*. Applikationerna ska kunna kommunicera med ett öppet *API* [7]. Ett *API* är en specifikation för hur en mjukvara kan kommunicera med en annan mjukvara. Detta är av intresse för *Smart Refill* eftersom de applikationerna som de utvecklar använder och kommunicerar med flera olika *API*:er.

1.2. Syfte

Syftet med examensarbetet är att utveckla två prototyper med var sitt *Cross Platform* ramverk, *React Native* och *Flutter*, för att undersöka de båda ramverkens olika fördelar och nackdelar samt utvärdera vilket av dem som är lämpligast för *Smart Refill*.

Eftersom det idag finns två ledande aktörer på marknaden för mobilplattformar, *iOS* och *Android*, behöver aktörer som utvecklar mobilapplikationer göra två stycken applikationer om man vill täcka hela marknaden [8]. Med ett *Cross Platform* ramverk behövs däremot teoretiskt bara en applikation för att täcka hela marknaden. På grund av detta så borde en undersökning av *Cross Platform* ramverk vara intressant för alla aktörer som utvecklar mobilapplikationer då det idag råder brist på utvecklare i hela IT-branschen [9].

1.3. Målformulering

Målet med examensarbetet är att utveckla en prototyp av en mobilapplikation med ramverket *React Native* och en prototyp av en mobilapplikation med ramverket *Flutter*. I målet ingår även att prototyperna ska kunna kommunicera med ett öppet *API*. Prototyperna kommer att ha nära nog identisk design och funktionalitet. Genom dessa kan en studie och jämförelse av de två ramverken utföras. Punkter som kan studeras vid jämförelsen är *hur lång tid programmet tar att kompilera*, *hur lång tid det tar för respektive prototyp att hämta data från ett öppet API*, *filstorlek*, *hur*

stort nätverk av utvecklare som jobbar med ramverket (så kallat community) och *vilka aktörer på marknaden som använder respektive ramverk idag*.

1.4. Problemformulering

Här följer de frågeställningar som besvaras av examensarbetet:

- Hur kan man utveckla en prototyp av en mobilapplikation med *React Native* som hämtar data från ett öppet *API* och visar datan i prototypen på ett överskådligt sätt i form av en lista?
- Hur kan man utveckla en prototyp av en mobilapplikation med *Flutter* som hämtar data från ett öppet *API* och visar datan i prototypen på ett överskådligt sätt i form av en lista?
- Hur lång tid tar det för respektive prototyp att hämta data från *API*:et och skiljer sig tiden väsentligt mellan prototyperna?
- Hur skiljer sig prototyperna ur ett utvecklarperspektiv?
  - Behöver man några särskilda verktyg för att kunna utveckla?
  - Vilket programspråk använder man?
  - Hur lång tid tar respektive prototyp att kompilera?
  - Är där någon skillnad i filstorlek på prototyperna?
  - Vad har andra utvecklare för erfarenhet av respektive ramverk?
  - Hur stort är respektive community?

1.5. Motivering av examensarbetet

De två dominerande operativsystemen för mobiltelefoner som används idag är *Android* och *iOS*. Därför måste alla som utvecklar mobilapplikationer och vill nå ut till så många användare som möjligt bygga en applikation för respektive operativsystem. Om det visar sig att man kan utveckla en applikation till båda operativsystemen med hjälp av *Cross Platform* teknik som fungerar lika bra som en applikation utvecklad med traditionell *Native* teknik skulle det kunna leda till stora resurssparingar. Eftersom *Smart Refill* är en förhållandevis liten aktör i branschen har man därför gått i tankarna att på sikt gå över till att utveckla sina applikationer med *Cross Platform* teknik.

1.6. Avgränsningar

Tillsammans med *Smart Refill* beslutades vilka avgränsningar som skulle förhållas under examensarbetets gång. Detta gjordes med hänsyn till hur mycket tid som fanns samt vad som skulle var realistiskt att göra erfarenhetsmässigt. Diskussionerna resulterade i följande avgränsningar.

- Applikationerna byggs utan någon särskild säkerhet.
- Applikationerna kommer vara prototyper som ska testa funktionerna i problemformuleringen och inte färdiga applikationer.
- Applikationerna ska hämta data från något *API* som är tillgängligt för allmänheten vilket innebär att det inte kostar pengar.

2. Teknisk bakgrund

I det här kapitlet förklaras de olika teknikerna som använts under examensarbetet. Det börjar med att förklara de båda begreppen *Native* och *Cross Platform* samt förtydligar om vilket *IDE* som använts. *IDE* står för *Integrated Development Platform* och är det verktyg som används för att skriva, kompilera och testa kod [10]. *IDE* tas upp eftersom det bör vara intressant för en person som själv vill testa att utveckla en applikation med något *Cross Platform* ramverk. Därefter så undersöks och beskrivs de båda ramverken *React Native* och *Flutter.* Till sist förklaras även verktygen *Hot Reload* och *Postman,* samt hur de simulatorer som använts under examensarbetet fungerar.

2.1. IDE och Visual Studio Code

Ett *IDE* (*Intgrated Development Environement*) är ett verktyg som underlättar processen för mjukvaruutveckling. *IDE*:et består av en texteditor som underlättar kodskrivning genom att enkelt kunna felsöka koden, en miljö för att enkelt kunna exekvera programmet direkt i verktyget. *IDE*:er brukar också kunna automatisera byggprocesser och använda verktyg för debuggning som är en typ av felsökning i programkod [11]. Det finns olika *IDE*:er som är anpassade för olika programspråk.

Prototyperna i detta examensarbete programmerades i utvecklingsmiljön *Visual Studio Code* [12]. *Visual Studio Code* är ett gratis verktyg från *Microsoft* som idag är populärt för webbutveckling men det används även vid utveckling av andra program så som

mobilapplikationer. *Visual Studio Code* har många så kallade plugins och extensions som man kan anpassa editorn med. Detta möjliggjorde att utveckling för både *React Native* och *Flutter* kunde göras i *Visual Studio Code*.

2.2. Native mobilapplikation

Idag domineras mobilmarknaden av de två plattformarna *iOS* som kommer från företaget *Apple* och *Android* som kommer från *Google* [13] [14]. *iOS* och *Android* är två plattformar som tillhandahåller olika tekniker för hur en utvecklare skapar en applikation. En applikation som utvecklas med dessa teknikerna är en *Native* applikation vilket innebär att den endast fungerar på den ena plattformen. En *Native* applikation som utvecklats för *iOS* fungerar endast på *iOS* och vice versa för *Android*.

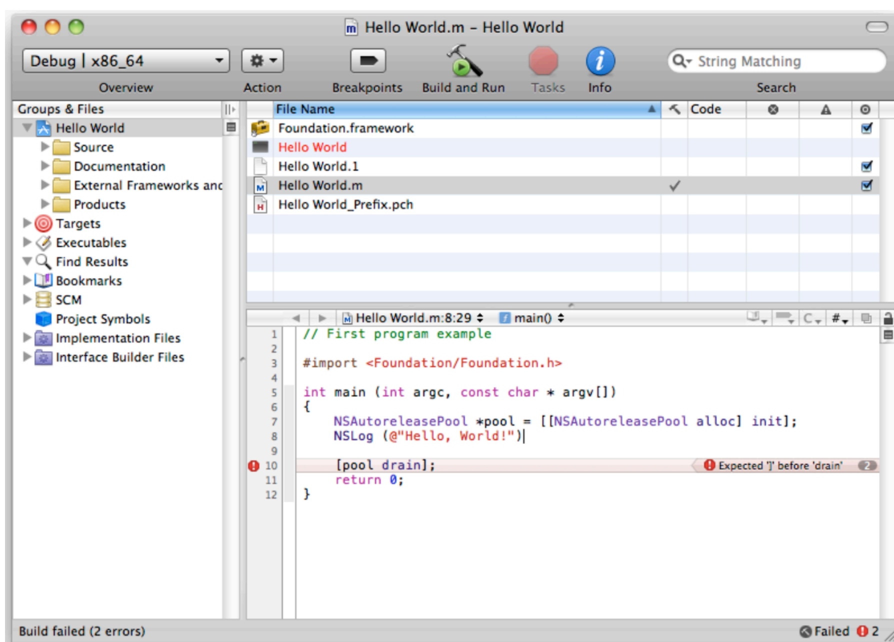
Fördelen med detta är att tekniken som används vid utvecklingen tillhandahålls av samma företag som tillhandahåller mobilens plattform. Detta innebär att tekniken är optimerad för den tänkta plattformen och applikationen som är byggd med *Native* teknik presterar perfekt på den tänkta plattformen [15].

Nackdelen är att man behöver utveckla två applikationer för att kunna täcka hela marknaden. Att utveckla två applikationer istället för en kräver mer resurser och är mer kostsamt.

2.2.1. iOS

Apples operativsystem till mobiltelefoner heter *iOS* [13] som är en förkortning för *iPhone Operating System*. *Native* applikationer utvecklas med programspråken *Objective-C* [16] eller *Swift* [17]. För *Native* utveckling heter verktyget *Xcode* som innehåller *iOS SDK* (*Software Development Kit*) som behövs för utveckling. *Xcode* fungerar endast på datorer från *Apple* med operativsystemet *MacOS* och är idag gratis att ladda ner. *Xcode* tillhandahåller ett *IDE* och

simulatorer av samtliga modeller av *iPhone* och även *iPad* och *Apple TV*. Nedan visas en bild över hur *Xcode* ser ut (se figur 2.1). Applikationer till *iOS* laddas ner via *App Store*. *App Store* är den digitala marknadsplatsen för applikationer till *iOS*. Alla applikationer på *App Store* är granskade av *Apple*.

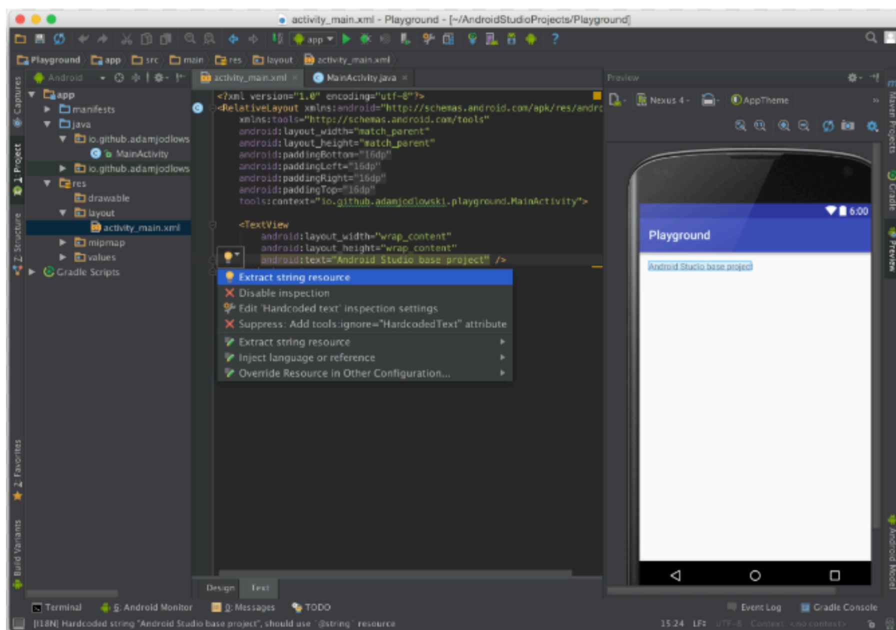


Figur 2.1 - Exempel över *Xcode*

2.2.2. Android

Googles operativsystem till mobiltelefoner heter *Android* och *Native* applikationer utvecklas med programspråken *Java* eller *Kotlin* [18]. För *Native* utveckling heter verktyget *Android Studio* som innehåller *Android SDK* (*Software Development Kit*) som behövs för utvecklingen. *Android Studio* finns till *Windows*, *MacOS* samt *Linux* och är gratis att ladda ner. *Android Studio* tillhanda håller ett *IDE* och simulatorer för olika modeller av mobiltelefoner. Nedan visas en bild över hur *Android Studio* ser ut (figur 2.2). Applikationer till *Android*

laddas ner från via *Google Play*. *Google Play* är motsvarigheten till *App Store* för *iOS*.



Figur 2.2 - Exempel över *Android Studio*

2.3. Cross Platform mobilapplikation

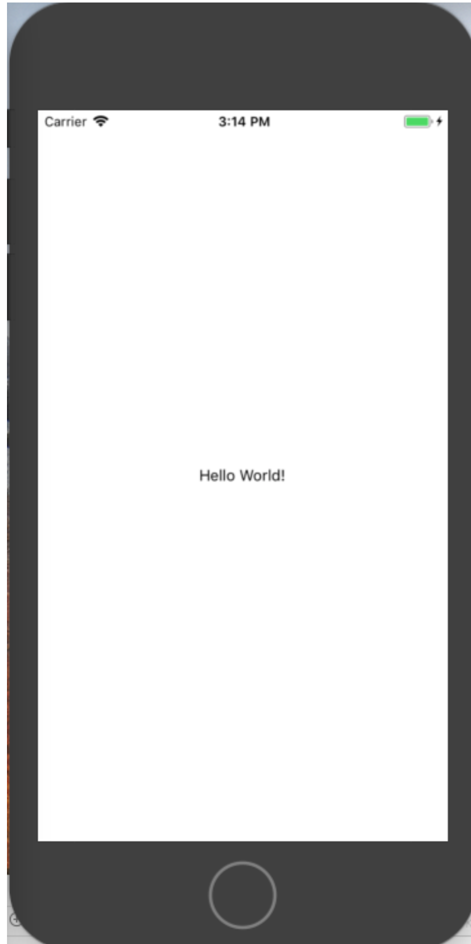
Förutom *Native* finns det olika möjligheter att utveckla mobilapplikationer som fungerar på flera olika plattformar. Bland dessa olika teknikerna kan nämnas *Phonegap*, *Xamarin*, *Ionic*, *NaticeScript*, *React Native* och *Flutter*. Samtliga av dessa har olika sätt för hur en applikation byggs som fungerar på olika plattformar. Det som en utvecklare ska ha i åtanke är att den här tekniken ofta behöver kompromissa med prestandan för att tekniskt fungera på flera plattformar. Ett exempel på detta är om en *Cross Plattform* teknik översätter sin kod från språket som det skrivits i till språket som operativsystemet använder. Då behöver översättningen göras vilket är ett extra steg som tar prestanda. Detta hade inte inträffat med *Native*

teknik men då hade den bara fungerat på ett operativsystem. Eftersom *Android* och *iOS* är två olika operativsystem kan det göras olika typer av kompromisser för att en applikation ska fungera på båda. Det kan exempelvis vara att applikationen inte blir helt integrerad med operativsystemet vid en ny uppdatering av operativsystemet [19].

2.4. React Native

React Native [6] är ett programbibliotek som utvecklats av *Facebook* för att bygga *Cross Platform* mobilapplikationer. Språket man använder till *React Native* kallas för *JSX* och är en blandning av *JavaScript* och *XML* [20]. Applikationer som byggs med *React Native* består av olika beståndsdelar som betecknas *components*.

Nedan visas ett exempel på källkod för en Hello World applikation samt hur det ser ut på en mobilskärm (figur 2.3 och figur 2.4). I bilden över källkoden (figur 2.4) utgår man från rad 4 där huvudklassen startar med metoden *render* som returnerar den grafiska komponenten *view*. *View* hämtar i sin tur data från komponenten *container*. Detta fungerar precis som en *CSS*-fil där utvecklaren kan välja att sätta egna värden på olika attribut så som var texten ska utgå från på skärmen och vilken färg den ska ha. Inuti *view* skrivs sedan texten ut med textattributet *text*. Detta påminner om hur man jobbar med webbutveckling vilket har sin förklaring av att *React Native* bygger på ramverket *React* som är ett av de mest populära ramverken som används vid webbutveckling. I figur 2.3 ser man resultatet av källkoden kompilerad.



Figur 2.3 - Hello World *React Native*

```
App.js

import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello World!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Figur 2.4 - Källkod över Hello World med *React Native*

En fördel med *React Native* är att det, precis som *Flutter*, har *Hot Reload* så man kan se ändringar i realtid under en utvecklingsprocess. Communityn är stor och det finns flera stora aktörer som använder *React Native* till sina mobilapplikationer. Bland dessa kan främst nämnas *Facebook* [21]. En nackdel kan vara att det finns väldigt mycket tredjeparts-kompononenter till *React Native*. Problemen med dessa kan vara att de saknar dokumentation eller att de ej är uppdaterade och därmed innehåller säkerhetshål.

2.5. Flutter

Flutter [5] är ett så kallat programbibliotek som utvecklats av *Google*, med syftet att kunna bygga mobila applikationer till de båda operativsystemen *iOS* och *Android*. Det är *Open Source*, vilket innebär att all källkod är öppen och tillgänglig för allmänheten. *Flutters* kodbas bygger på det objekt-orienterade högnivåspråket *Dart* som lanserades av *Google* **2011**. Syftet med *Dart* är att det ska förenkla processen att bygga avancerade webb-applikationer. *Dart* har en syntax som påminner om andra vanliga programspråk som *Java* och *C#*. Vilket på så sätt gör det lätt för utvecklare att komma igång [22]. *Dart* är ett språk som har många likheter med *Java*. Det är objekt-orienterat och baseras på klasser.

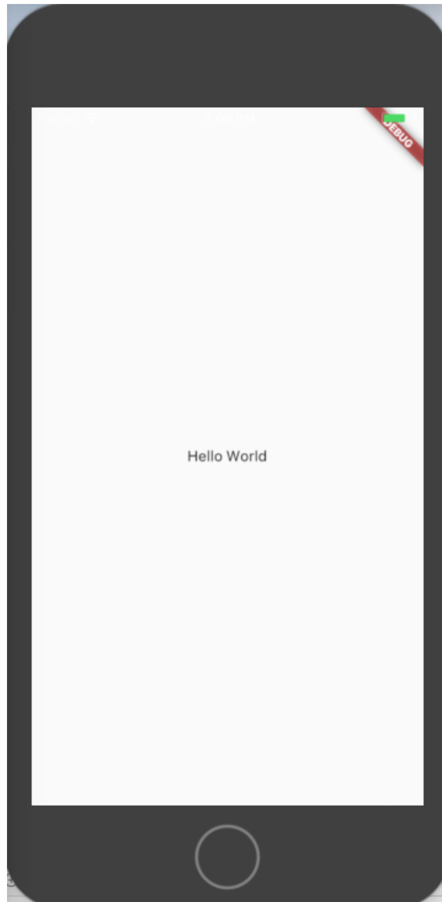
Stommen i en applikation byggd med *Flutter* är något som *Google* döpt till *Widgets*. *Widgets* är klasser skrivna i *Dart* som bestämmer applikationens grafiska layout och funktionalitet. *Flutter* tillhandahåller olika varianter av *Widgets* med olika syften som en utvecklare kan kombinera till en applikation. En applikation består således av en hierarkisk struktur av flera *Widgets*. Om man tar en enkel applikation som visar en lång text som en användare kan skrolla igenom så består den först av en *Widget* med en yta som kan skrollas och sen i den finns en *Widget* som kan visa text. På så sätt blir skroll-widgeten en förälder och text-widgeten ett barn i hierakin.

I bilden nedan visas hur en applikation som skriver ut Hello World (figur 2.5). På rad 4 startas applikationen med en main-metod som kör funktionen *runApp*. *RunApp* skapar upp en ny instans av klassen *GHFlutterApp* och kör den direkt (se rad 6). Inuti *GHFlutterApp* finns Widgeten *build* som börjar på rad 8. Den skapar och returnerar en ny instans av *MaterialApp* som håller widgeterna för den grafiska layouten. Inuti *MaterialApp* finns widgeten *Scaffold* som börjar på rad 11. *Scaffold* innehåller widgeten *body* som talar om att texten ska vara centrerad och *body* innehåller därefter widgeten *child* där texten som ska skrivas ut är. Här kan man se hierakin tydligt

från *Scaffold* till *body* till *child*. Hur applikationen ser ut visas i figur 2.6.

```
1   import 'package:flutter/material.dart';
2   import 'strings.dart';
3
4   void main() => runApp(new GHFlutterApp());
5
6   class GHFlutterApp extends StatelessWidget {
7     @override
8     Widget build(BuildContext context) {
9       return new MaterialApp(
10        title: Strings.appTitle,
11        home: new Scaffold(
12
13          body: new Center(
14            child: new Text('Hello World'),
15          ), // Center
16        ), // Scaffold
17      ); // MaterialApp
18    }
19  }
```

Figur 2.5 - Källkod över Hello World med *Flutter*



Figur 2.6 - Hello World *Flutter*

En fördel med *Flutter* är att det använder verktyget *Hot Reload*. Det innebär att en utvecklare kan se ändringar som gjorts i realtid och hen behöver inte kompilera om varje gång [23]. Vanligtvis tar det några sekunder, upp mot en minut, för programmet att byggas om och tack vare *Hot Reload* behöver utvecklaren inte ens tänka utan processen startas så fort de nya ändringarna har sparats.

Nackdelen är dock att det allmänna intresset (communityn) runt *Flutter* fortfarande är litet och ibland kan det vara svårt att hitta hjälp

från andra utvecklare på exempelvis det stora webb-forumen för utveckling som exempelvis *Stack Overflow*.

2.6. Postman

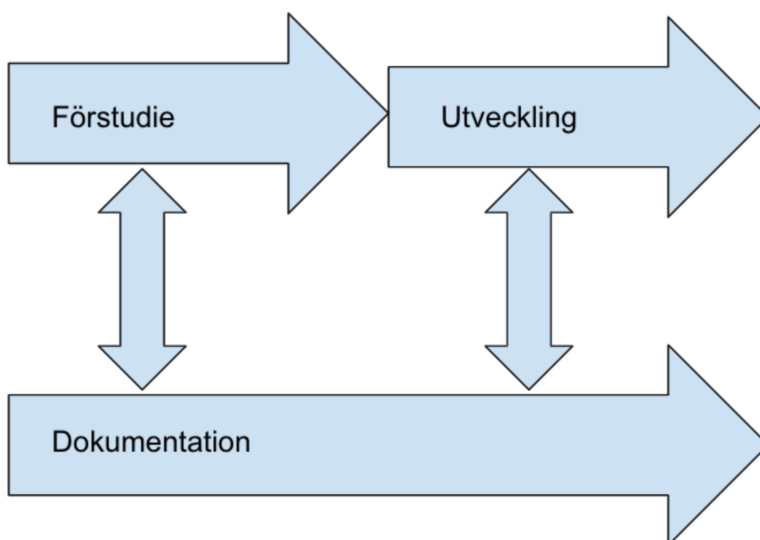
Postman är verktyg som används vid utveckling av *API*:er. Verktyget tillhandahåller ett grafiskt gränssnitt där en utvecklare kan testa ett *API*:s olika funktioner. Detta görs genom så kallade anrop över *HTTP* [24]. Om en utvecklare byggt en applikation som hämtar data från en databas via ett *API* och vill testa om det går att filtrera datan efter olika parametrar så kan detta testas med hjälp av *Postman* först för att verifiera att *API*:et fungerar. *Postman* finns både som gratis- och betalversion [25] [26].

3. Metod

I detta kapitlet förklaras de metoder som använts under examensarbetet. Först introduceras den övergripande projektmodellen och därefter projektmodellens olika faser. Slutligen innehåller det även ett delkapitel om handledningen och ett delkapitel om källkritik.

3.1. Upplägg

I detta kapitel beskrivs vilka faser, steg och metoder som användes under examensarbetet. Var och en av de tre faserna bestod av de tre stegen förstudie, utveckling och dokumentation. Faserna delades upp i två parallella steg där det ena steget bestod av förstudien (se kapitel 3.3) och utvecklingen (se kapitel 3.4) och det andra av dokumentation. Dessa steg illustreras nedan (se figur 3.1). Utvecklingen delades i sin tur upp fyra sekventiella steg som beskrivs i kapitel 3.2. Förstudien och utvecklingen skedde till stor del på *Smart Refills* kontor i Helsingborg.



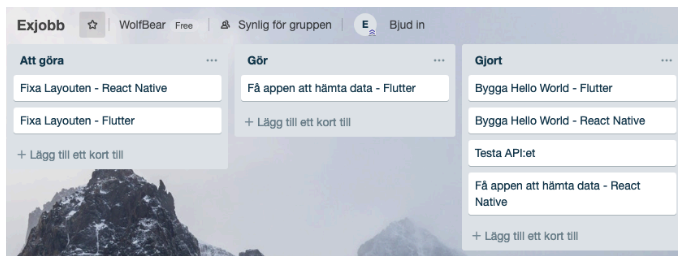
Figur 3.1 - Projektmodell över examensarbetet

3.2. Projektmodell för utveckling

För utvecklingen tillämpades en enkel projektmodell som var inspirerad av *SCRUM* [27]. Den utgick från en enkel *Scrum Board* med de tre kategorierna: *Att göra*, *Gör* och *Gjort*. Utvecklingen delades in i stegen som motsvarade korten (se figur 3.2).

1. Bygga Hello World för respektive ramverk
2. Testa *API*:et
3. Få appen att hämta data för respektive ramverk
4. Fixa layouten på appen för respektive ramverk.

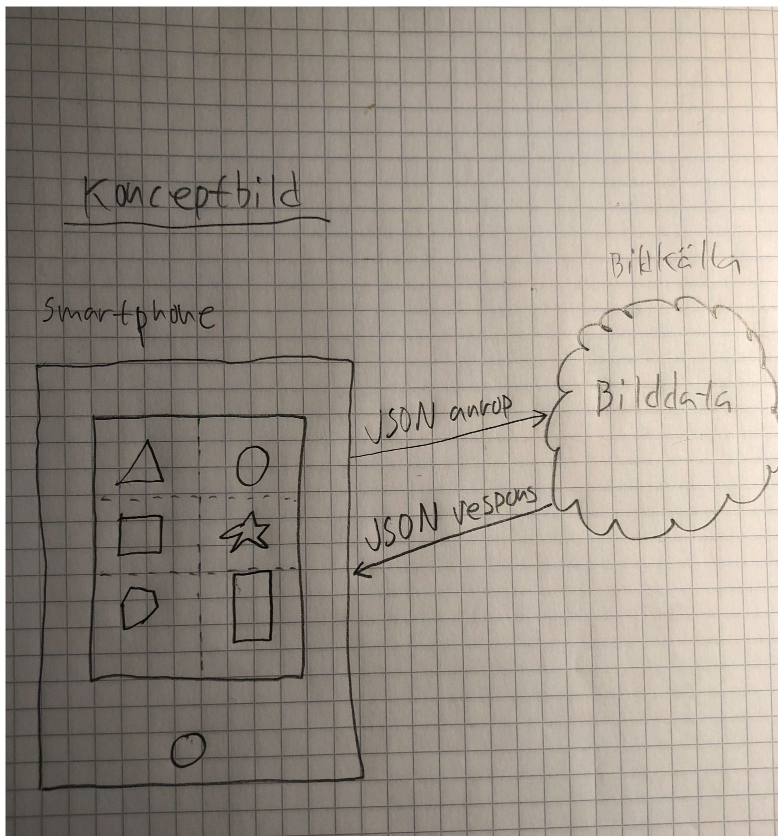
Samtliga fyra delar förklaras i delkapitel 3.4. Applikationerna utvecklades parallellt och eftersom examensarbetet gjordes ensam tillämpades ingen daglig standup.



Figur 3.2 - Bild över den *Scrum Board* som användes

3.3. Förstudie

Examensarbetet började med en förstudie som till en början bestod av att undersöka hur man programmerar med *React Native* och *Flutter*. Undersökningen bestod av litteraturstudier och dokumentation. Detta resulterade sedan i att en enkel konceptbild över hur applikationerna skulle se ut skissades fram med papper och penna (figur 3.3). Bilden blev en prototyp som visade hur den färdiga applikationen skulle se ut och fungera för att stämma med den målformulering som finns i kap 1.3. Detta var för att få en bild över hur det färdiga resultatet skulle se ut. Bilden visar en smartphone som anropar en källa med bilder om att den vill hämta bilder enligt anropet. Bildkällan skickar sen bilderna som visas upp i smartphonen i en vertikal lista med två bilder på varje rad.



Figur 3.3 - Konceptbild över den färdiga applikationen

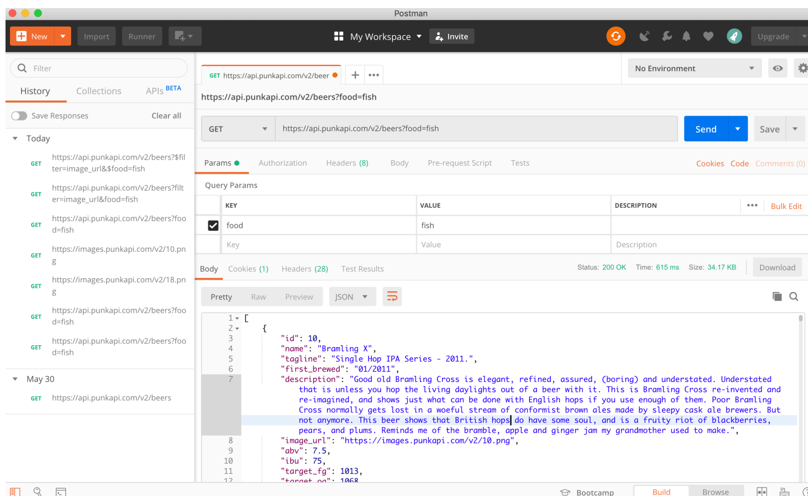
Till en början så studerades de båda officiella hemsidorna för *React Native* och *Flutter* [5] [6]. Ett flertal olika artiklar, både som förklarar tekniken bakom ramverken, som går igenom hur man utvecklar olika sorters applikationer och deras respektive för- och nackdelar lästes samt antecknades. För detta användes till en början två stycken block för respektive ramverk (*React Native* och *Flutter*) där mycket anteckningar och sammanfattningar gjordes utifrån faktan som fanns på ramverkens officiella hemsidor. Båda hemsidorna hade tydliga instruktioner för hur man snabbt kunde komma igång och bygga enkla applikationer. Det första steget här blev att utveckla varsin applikation som kunde visa upp text vilket blev de båda applikationerna som visas i kap 2 (figur 2.2 och figur 2.4). Därefter testades hur man kunde ändra layouten med hjälp av olika funktionaliteter. Andra funktionaliteter som testades var hur man skulle implementera vyer som en användare skulle kunna skrolla igenom samt olika varianter för hur man kunde visa upp olika listor på bästa sätt samt hur man importerade bilder. Anledningen till att detta testades var för att det var funktioner som var tänkta att finnas med i den färdiga prototypen för de båda ramverken. Allt detta antecknades i blocken.

Förutom faktan från de officiella hemsidorna så införskaffades även två stycken nybörjarkurser för respektive ramverk (*React Native* och *Flutter*) från kursföretaget *Udemy* [28] [29]. Ett antal olika artiklar från *Medium*, som är en stor källa inom teknik och IT, lästes och antecknades. Samtliga artiklar som har använts är de som fungerar som referenser i det här examensarbetet.

3.4. Utveckling

Efter förstudien påbörjades själva utvecklingsfasen. Två stycken projekt, ett för varje ramverk, skapades enligt de instruktioner som från de båda ramverkens officiella hemsidor [30] [31]. Ett projekt är i det här fallet ett programpaket bestående av olika filer som ska bli den färdiga applikationen. Projekten genererar man med hjälp av ramverken, *React Native* och *Flutter*. Själva processen för hur man gör detta är enkelt och går ut på att utvecklaren ser till att ha installerat sitt ramverk enligt dess officiella instruktioner. Därefter skapar utvecklaren en mapp på datorn i vilken man vill ha sitt projekt. Slutligen går utvecklaren in i sin terminal och matar in ett kommando som laddar ner projektet till mappen från Internet.

Stort fokus låg under den här fasen på att få kommunikationen med de valda *API*:et att fungera. *API*:et som valdes var öltillverkarens *BrewDogs,* över deras ölsortiment [32]. Deras *API* är gratis och öppet för allmänheten och datan fungerar utmärkt att visa som en lista. För att kunna hämta data från *API*:et behövs en så kallad *GET*-metod. En *GET*-metod är ett anrop som görs över *HTTP*-protokollet där utvecklaren kan filtrera ut vilken data som ska hämtas genom att mata in olika parametrar. Man kan använda dessa för att anpassa metoden och filtrera ut olika typer av data. För att testa detta användes verktyget *Postman*. Bilden nedan visar data på alla ölsorter som passar att dricka till fisk (enligt *BrewDog*). Anledningen till att valet blev att bara visa öl som passar till fisk var för att testa att filtrera datan samt att det gav en kortare lista. Datan har erhållits som ett så kallat *JSON*-objekt (se figur 3.4).



Figur 3.4 - Test av *HTTP*-anrop med *Postman*

Därefter genomfördes utveckling som möjliggjorde att de båda applikationerna kunde hämta *JSON*-objekten i simulatorerna. När detta fungerade så påbörjades utveckling av det grafiska gränssnittet för att få applikationen att visa informationen given av *API*:et.

Nedan så sammanfattas de fyra faserna från den *Scrum Board* som användes under utvecklingen (se delkapitel 3.2):

3.4.1. Bygga Hello World för respektive ramverk

Det första steget i utvecklingsfasen var att testa att bygga en enkel applikation för respektive ramverk (*React Native* och *Flutter*). För att möjliggöra detta måste först varje ramverk installeras på den datorn som ska användas för utvecklingen. Instruktioner för hur dessa installerades fanns på ramverkens officiella hemsidor [30] [31]. Efter att dessa var installerade så skapades två projekt, ett för varje ramverk, som skulle testa hur en utvecklare gör en enkel applikation.

Detta blev de två applikationerna som beskriv i delkapitel 2.4 (figur 2.1 och figur 2.2) och 2.5 (figur 2.3 och 2.4).

3.4.2. Testa API:et

Nästa steg var att hitta ett öppet *API* som kunde användas för att hämta någon data ifrån. Valet föll på ett *API* som tillhandahölls av den skotska öltillverkaren *BrewDog* [32]. För att testa *API*:et användes verktyget *Postman* (se delkapitel 2.6).

3.4.3. Få Applikationen att hämta data för respektive ramverk

Nedan beskrivs hur applikationen för respektive ramverk kommunicerades med *API*:et från *BrewDog* [33].

3.4.3.1. React Native

För att hämta datan från *API*:et användes en metod som döptes till *getData* som börjar på rad 32 (figur 3.5). Här användes en metod från *JavaScript* som kallas för *fetch* och används för att hämta objekt. *Fetch* börjar på rad 33 (figur 3.5) och tar länken till *API*:et som inparameter. På rad 34 och 35 finns därefter två metoder av typen *then*. *Then* är metod som används när man arbetar med ett *Promise* i *JavaScript*. Ett *Promise* garanterar att nästkommande kod inte kommer köras förrän datan har hämtas. Den första *then* metoden gör om data från *API*:et till ett *JSON*-objekt och den andra lägger in det i en lista i applikationen.

```
32  getData() {
33    fetch('https://api.punkapi.com/v2/beers?food=fish')
34      .then(res => res.json())
35      .then(res => this.setState({ beers: res }));
36  }
```

Figur 3.5 - Kopplingen mellan *React Native* och *API*:et

3.4.3.2. Flutter

För att hämta datan från *API*:et användes en så kallad *Future* som är en inbyggd objekttyp för att hämta data asynkront. Den går från rad 53 till rad 59 och döptes till *getJson* (figur 3.6). En sträng av länken till *API*:et som hämtar alla ölsorter som passar till fisk skapas på rad 54 som döptes till *apiUrl* (figur 3.6). På rad 56 skapas ett objekt av typen *HTTP Response* som döptes till *response* som hämtar *apiUrl* (figur 3.6). Slutligen returneras *response* i form av ett *JSON*-objekt bestående av själva bodyn på resultatet från *response* på rad 58 (figur 3.6). Bodyn är den delen som innehåller all data från API:et. Bland dessa kan nämnas den länken till bildfilen som ska användes i nästa steg.

```
53  Future<List> getJson() async {
54    String apiUrl = 'https://api.punkapi.com/v2/beers?food=fish';
55
56    http.Response response = await http.get(apiUrl);
57
58    return JSON.decode(response.body);
59  }
```

Figur 3.6 - Kopplingen mellan *Flutter* och *API*:et

3.4.4. Design av layouten på applikationen för respektive ramverk

Nedan beskrivs hur layouten designades för respektive ramverk.

3.4.4.1. React Native

När den grafiska layouten designades började det med ett objekt av typen *Scrollview* på rad 52 (figur 3.8). *Scrollview* används när man ska visa upp mer information än vad som får plats på en vanlig mobilskärm. *Scrollview* känner av hela informationsslingan och möjliggör att användaren kan skrolla igenom den. I det här fallet användes den eftersom listan med bilder som skulle visa var större än vad som fick plats. Inuti *Scrollviewn* skapades ytterligare två objekt, en *header* och en *container*. Headern bestod av en text och två bilder. Containern bestod därefter av alla de bildfiler som hämtades från *API*:et i en vertikal lista med två bilder på varje rad (figur 3.10).

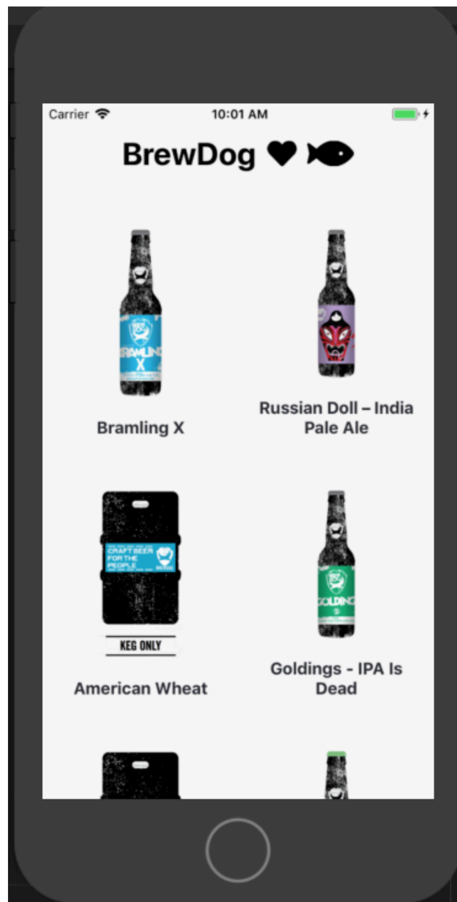
För *React Native* användes en så kallad *Stylesheet* som där all specifikation för layouten bestämdes (figur 3.9). Den fungerar precis som en *CSS*-fil i *HTML*.

```
38    render() {
39      const beers = this.state.beers.map((beer, index) => {
40        return (
41          <View style={styles.beer} key={index}>
42            {beer.image_url && <Image
43              style={{flex:1, height: undefined, width: undefined}}
44              source={{uri: beer.image_url}}
45              resizeMode="contain"
46            />}
47            <Text style={styles.text}>{beer.name}</Text>
48          </View>
49        );
50      });
51      return (
52        <ScrollView style={styles.scroll}>
53          <View style={styles.header}>
54            <Text style={styles.heading}>BrewDog</Text>
55            <Image
56              style={{height: 24, width: 28, marginRight: 10}}
57              source={require('./heart.png')}
58            />
59            <Image
60              style={{height: 24, width: 45}}
61              source={require('./fish.png')}
62            />
63          </View>
64          <View style={styles.container}>
65            {beers}
66          </View>
67        </ScrollView>
68      );
69    }
70  }
```

Figur 3.8 - Källkoden över layouten för *React Native*

```
72  const styles = StyleSheet.create({
73    header: {
74      flex: 1,
75      flexDirection: 'row',
76      justifyContent: 'center',
77      alignItems: 'center',
78      padding: 30,
79    },
80    heading: {
81      paddingRight: 10,
82      fontSize: 30,
83      fontWeight: 'bold',
84    },
85    scroll: {
86      backgroundColor: '#F5F5F5',
87    },
88    container: {
89      flex: 1,
90      flexDirection: 'row',
91      justifyContent: 'space-around',
92      flexWrap: 'wrap',
93    },
94    beer: {
95      paddingTop: 25,
96      paddingBottom: 25,
97      height: 250,
98      width: 150,
99    },
100   text: {
101     paddingTop: 20,
102     color: '#2B2C34',
103     fontSize: 16,
104     fontWeight: 'bold',
105     textAlign: 'center',
106   },
107 });
108
```

Figur 3.9 - Källkoden över *StyleSheet* till *React Native*



Figur 3.10 - Layouten för React Native i Simulatorn.

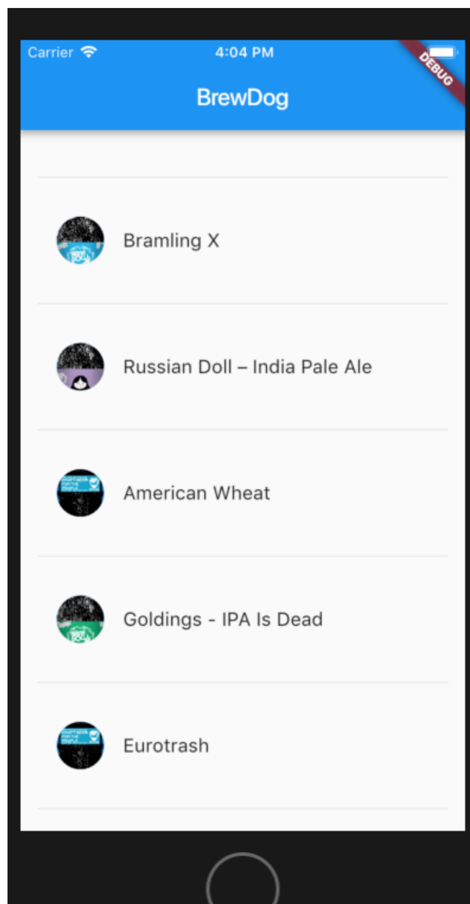
3.4.4.2. Flutter

Den grafiska layouten börjar med widgeten *Scaffold* som innefattar allt som användaren ser på skärmen (figur 3.11). Inuti i den finns två widgetar. Den ena heter *AppBar* som är en motsvarande header med texten: ”*BrewDog*”. Den andra är body som innehåller resten av all layout. I body skapas på rad 31 en widget som får namnet *child* och är av typen *ListView* som används för att skapa listor som en användare

kan skrolla igenom på en mobiltelefon. På rad 35 skapas sedan en widget av typen *itemBuilder* som tar de två parametrarna *context* och *position*. *Context* är själva bildobjektet och *position* är dess index i listan. Sedan returneras objektet. Här valdes att visa objekten i en vertikal lista (figur 3.12). *Divider* anger hur många pixlar varje objekt ska ha som höjd. Sedan anger widgeten *ListTile* på rad 39 att varje objekt visar namnet för respektive *JSON*-objekt på respektive position (index) (rad 40) och att varje bildobjekt som hämtas från samma position ska visas som ett cirkelobjekt.

```
26    runApp(new MaterialApp(
27      home: new Scaffold(
28        appBar: new AppBar(
29          title: new Text('BrewDog'),
30      ),
31      body: new Center(
32        child: new ListView.builder(
33          itemCount: _data.length,
34          padding: const EdgeInsets.all(14.5),
35          itemBuilder: (BuildContext context, int position){
36            return new Column(
37              children: <Widget>[
38                new Divider(height: 50.5),
39                new ListTile(
40                  title: Text("${_data[position]['name']}"),
41              //    leading: Image.network("${_data[position]['image_url']}")
42                  leading: new CircleAvatar(
43                    backgroundImage: NetworkImage(_data[position]['image_url']),
44                  ),
45                )
46              ],
47            );
48          })
49        ),
50      ),
51    ));
52  }
```

Figur 3.11 - Koden för layouten för *Flutter*



Figur 3.12 - Layouten för Flutter i Simulatorn.

3.5. Dokumentation

Dokumentationen gjordes parallellt med utvecklingen och faktasökningen. Detta var både stödanteckningar och minnesanteckningar samt det som blev rapporten. Eftersom det fanns mycket olika fakta från flertalet hemsidor användes en *Google Spreadsheet* (*Googles* motsvarighet till *Microsoft Excel*) för att hålla reda på diverse artiklar och annat som hittades. Anteckningarna

gjordes digitalt i ett *Google Docs*. Här användes en egen mapp för att ha koll på allt som hade med examensarbetet att göra. Även sådant som bilder sparades här. Faktan som dokumenterades var framförallt information från de båda officiella hemsidorna för ramverken [5] [6].

3.6. Handledning på Smart Refill

Smart Refill tillhandahöll två kontakter som var behjälpliga med alla frågor som dök upp. Regelbundna möten hölls ej men eftersom examensarbetet utfördes på *Smart Refills* kontor hölls en kontinuerlig kommunikation efterhand som frågetecken dök upp som kunde diskuteras och lösas.

3.7. Källkritik

Den primära faktan för detta arbete har kommit från de båda utgivarna av ramverken som studerades (*Facebooks* och *Googles*) egna hemsidor. Eftersom båda teknikerna är relativt nya var det svårt att hitta akademiska artiklar. Däremot fanns det många blogg-artiklar från programmerare, utvecklare och ingenjörer som jobbar och testar dessa tekniker dagligen.

Två källor som använts mycket är medium och Hackernoon. Dessa har båda gott rykte i branchen och utvecklare brukar använda dessa när dom letar efter fakta.

Codeforgeeks är populärt forum som berör allt som har med programmering att göra. Deras artiklar om it-relaterade begrepp bör anses som trovärdig.

Två artiklar från nyhetskanalerna Svenska Dagbladet och BBC som inte borde behöva ifrågassättas närmare.

PC Magazine är en tidskrift om datorer och som känns legitim.

O’REILLY är ett av de största utbildningsföretag för it så att hänvisa till dom borde också vara legitimt.

4. Analys

4.1. Projektmodell

Projektmodellen som användes för detta examensarbete tillsammans med brädan på *Trello* var enkel att tillämpa och gav en bra överblick över utvecklingsprocessen. Detta berodde antagligen på att det var ett förhållandevis enkelt utvecklingsprojekt utan så många komplexa delmoment samt att utvecklingen gjordes ensam vilket resulterade i att något verktyg för versionshantering inte användes.

4.2. Förstudie

Under den första fasen av detta examensarbete gjordes en förstudie. Det primära fokuset för examensarbetet låg i att jämföra de två *Cross Platform* ramverken *React Native* och *Flutter.* Förstudien inleds med en diskussion med handledarna från *Smart Refill* där det berättade om sina erfarenheter av respektive ramverk. Deras rekommendation var att den bästa dokumentationen finns på de båda officiella hemsidorna för *React Native* och *Flutter*. Information till förstudien kunde också erhållas från blogg-artiklar och *YouTube*-klipp. I början fanns endast frågorna från frågeställningen och de konstaterades snabbt att dessa behövde konkretiseras mer. Handledarna gav lite exempel på olika *API*:er som tillhandahöll bilder som skulle kunna användas. De tipsade även om bloggportalen *Medium* och berättade att de själva ofta läste artiklar därifrån som tillhandahöll idéer som de brukade använda i sitt dagliga arbete. De visade även hur *Postman* kunde användas. En sak som poängterades från deras sida var att det är viktigt att bilderna som visas inte är för små eftersom tanken är att det ska var synliga på en mobilskärm. En annan viktig sak som skulle hållas i åtanke var att användaren använder sina fingrar på skärmen istället för en muspekaren. Förstudien gjordes till viss del på *Smart Refills* kontor men följde inget strikt upplägg med regelbundna

möten. När konceptbilden började tas fram så visades den några gånger för de båda handledarna som gav sitt godkännande och hade inga invändningar.

Andra tekniker och verktyg som använts under utvecklingsfasen studerades även under förstudien. Det svåra i förstudien var att definiera hur den färdiga applikationen skulle se ut och fungera då kravspecifikationen inte var så tydlig. Detta skapade i början en del förvirring men efterhand som konceptbilden kom fram så försvann det. Inspiration till hur en lista med bilder skulle kunna se ut kom från olika källor men främst från hur andra färdiga applikationer är utformade, så som kontaktlistor och shoppinglistor. I takt med att slutmålet blev mer konkret så blev även förstudien lättare att planera.

4.2.1. Litteraturstudier

4.2.1.1. React Native

Större delen av faktan om ramverket *React Native* togs från dess egen hemsida. Hemsidan var informativ med flertal exempel. Intrycket som gavs var att skaparna av *React Native* vill att det ska vara lätt för utvecklare att komma igång. Då hemsidan underhålls och ägs av utgivaren av ramverket så får man anse att informationen är trovärdig och aktuell.

Eftersom *React Native* är en stor och etablerad spelare inom *Cross Platform* teknik så finns det mycket information att hitta online. Det finns ett stort nätverk av utvecklare som jobbar med ramverket dagligen. Information finns därför relativt enkelt att hitta via artiklar, *YouTube*-klipp samt på exempelvis frågeforumet *Stack Overflow*.

Några av dessa artiklar hittades på *Medium* som är en bloggportal med ett stort utbud av information om programmering, IT och systemutveckling. Det som är extra intressant med dessa artiklar är att

skribenterna, som oftast är erfarna utvecklare, gärna uttrycker sina egna åsikter och erfarenheter kring tekniken ifråga.

Kursen som köptes på utbildningsplattformen *Udemy* förklarade tekniken bakom ramverket vilket är intressant för alla utvecklare men som nybörjare var det kanske lite för mycket teori. För många är det lättare att ta in ny information genom konkreta exempel och praktiska övningar. Därför var kursen inte lika värdefull i en förstudie.

Som nybörjare kändes det tryggt att studera *React Native* då man visste att det fanns hjälp och svar på frågor att hitta online.

I den litterära studien av *React Native* så hittades all information som behövdes för att komma igång och utveckla. Detta var först instruktioner för hur ramverket installerades på en egen dator. Därefter var det olika instruktioner för hur en utvecklare kan bygga och exekvera olika enklare applikationer. Slutligen fanns det också instruktioner för hur mer avancerade komponenter kunde utvecklas. Bland dessa ingick kommunikationen med *API*:et och hur layouten kunde utformas. Överlag upplevdes dessa instruktioner som tydliga då de både innehöll generella instruktioner och exempel.

4.2.1.2. Flutter

Flutter är till skillnad från *React Native* en nyare aktör inom *Cross Platform* teknik. Dock finns det ändå rätt mycket information online. Nätverket är däremot något mindre och det är svårare att hitta svar via exempelvis *Stack Overflow.*

Flutters hemsida underhålls av dess utvecklare på Google och känns uppdaterad. Den var lätt att navigera och många exempel fanns att utforska. Större delen av informationen till förstudien hittades därför här. Det fanns även gott om artiklar på *Medium* som handlade om *Flutter*.

Udemy-kursen var utformad med liknande upplägg som för *React Native*. Den var lärorik men bidrog inte med lika mycket information som de andra källorna.

I det stora hela så skiljde sig inte den litterära studien mellan *React Native* och *Flutter* speciellt mycket. Den enda märkbara skillnaden mellan dem är att det inte finns lika många stora aktörer som använder sig av *Flutter* idag.

I litteraturstudien av *Flutter* var det precis som för *React Native* tydlig information som fanns att tillgå. Instruktionerna visade hur ramverket kunde installeras på datorer med olika operativsystem som var enkla att förstå. En sak som inte var helt tydligt beskrivet var hur man skulle sätta *Flutter* som global sökväg på datorn som det installerades på, vilket behövdes för att installera det. Detta nämndes men det förklarades inte så tydligt som hade önskats. I övrigt så var alla instruktioner tydliga och lättförståeliga med olika exempel över olika tekniker. Här var det precis som i fallet med *React Native* att de som användes här var främst information om kommunikation med *API*:et och utformningen av layouten.

4.2.2. Konceptbild

Handledarna på *Smart Refill* efterfrågade en applikation som skulle visa grafiskt innehåll då det är tydligare och mer underhållande att demonstrera. I samband med den här diskussionen så bestämdes att applikationen skulle visa en lista med bilder som användaren skulle kunna skrolla igenom. Informationen som hämtades från *API*:et skulle visas i en vertikal lista med två bilder per rad. Anledningen till detta val var att bilderna inte skulle bli för små eller för stora på mobilskärmen. För att visualisera hur denna applikation skulle kunna se ut rent grafiskt togs en konceptbild fram.

Handledarna på *Smart Refill* klargjorde att den grafiska layouten kunde ändras och anpassas efterhand under utvecklingen, men att det

var bra att ha en färdig konceptbild så det fanns en målbild att inspireras av. Att layouten senare eventuellt skulle kunna ändras var inget som skulle påverka resultaten i frågeställningen. Handledarna menade att sådana ändringar under ett projekts gång är snarare regel än undantag ute i arbetslivet.

Under utvecklingsfasen så användes konceptbilden som en mall då den grafiska layouten implementerades. Konceptbilden gjorde det enklare att få de två olika applikationerna i *React Native* och *Flutter* att grafiskt likna varandra så mycket som möjligt. I slutändan underlättade det att visa en bild per rad i *Flutter*. Men denna ändring är inte något som påverkar prestandan eller resultatet i undersökningen.

4.3. Utveckling

Med hjälp av enkel och tydlig dokumentation kring de båda ramverken kunde utvecklingsfasen genomföras. Den största källan till information fanns hos de båda ramverkens egna hemsidor som var fyllda med guider och exempel att följa då applikationerna skulle implementeras.

Utöver detta så var de båda kurserna i *React Native* och *Flutter* från utbildningsföretaget *Udemy* till stor hjälp under denna fas för att exempelvis titta på hur kopplingen mot *API*:et kunde göras och hur layouten kunde designas på olika sätt utifrån konceptbilden.

Precis som under de flesta utvecklingsprocesser av mjukvara var även *Stack Overflow* till stor hjälp. Diverse problem som exempelvis när det kom till installationen av ramverken och hur terminalen kunde användas på ett smart sätt fanns svaren tillgängliga hos frågeforumet.

Under de två delkapitlen nedan presenteras en mer djupgående analys av respektive ramverk följt av en analyserande jämförelse av *React Native* och *Flutter*.

4.3.1. React Native

Under utvecklingen av applikationen i *React Native* så bröts fasen ner i flertal mindre steg; val av *IDE*, val av språk, val av simulator, koppling till *API*:et och layouten. Nedan finns en djupare analys kring varje steg och bakgrunden till de val som gjordes.

Utvecklingen av applikationen gjordes med hjälp av verktyget och *IDE*:et, *Visual Studio Code.* Då verktyget är gratis och populärt bland professionella utvecklare i branschen så kändes det som ett enkelt val. *Visual Studio Code* stödde de funktionaliteterna som krävdes för att utveckla applikationen och det levde upp till alla de förväntningar som man har på ett *IDE*.

I val av språk stod det mellan *JavaScript* och *JSX*. Enligt *React Natives* officiella dokumentation så rekommenderades *JSX* som är en kombination av *JavaScript* och *HTML* där utvecklaren kan skriva funktioner och definiera gränssnitt i samma syntax. Detta, i kombination med att de flesta exempel som fanns tillgängliga var skriva i *JSX,* gjorde att valet föll på det språket. Hade *JavaScript* valts istället så hade det antagligen blivit mer tidskrävande då det hade krävts en större kodbas eftersom funktionerna och gränssnittet då varit separerade. Dessutom upplevdes det som att det fanns mindre dokumentation där *JavaScrip*t användes i jämförelse med *JSX*.

Applikationen testades med den *iPhone*-simulator som ingår i programmet *Xcode* som finns tillgänglig för *MacOS*-användare. Simulatorn ger automatiskt tillgång till samtliga modeller av *iOS* och gjorde det därför till ett självklart val jämfört med andra simulatorer på marknaden.

För att hämta datan från *API*:et användes ett eget *API* tillhandahållet från *React Native* som heter *Fetch.* Detta *API* var även rekommenderat av grundarna själva. Två alternativ som även nämndes hette *Axios* och *Frisbee*. *Axios* testades men efter en

diskussion med en av handledarna på *Smart Refill* föll valet istället på *Fetch* eftersom det är mest beprövat.

Layouten till applikationens gränssnitt är uppbyggd med komponenten *Scrollview* som i sin tur innehåller de två komponenterna *header* och *container*. Containern bestod av bildfilerna som lästes in. Dessa låg i en lista med två kolumner. Layoutens utseende bestämdes med en så kallad stylesheet. Scrollviewn valdes eftersom den innehöll både funktionen för att visa listan och funktionen för att skrolla igenom listan. Valet att lägga bilderna som en lista med två kolumner gjordes utifrån konceptbilden. Stylesheeten påminde om hur en webbutvecklare bestämmer en grafisk layout med en *CSS*-fil och då *CSS* är välkänt så föll valet på det utan att undersöka närmare alternativ. Med en del erfarenhet av *CSS* i bagaget så blir *Stylesheet* en smidig övergång och man kommer snabbt igång med att styla sin applikation.

Det tog i genomsnitt ca **0.008** sekunder avrundat att hämta datan från *API*:et. Genomsnittet baserades på tio oberoende tester där tiden för hämtningen loggades. Resultatet på testerna varierade från ca **0.005** sekunder till ca **0.009** sekunder. Eftersom det rörde sig om så pass snabba hämtningar och att de inte varierade så mycket valdes den här avrundningen och att det antogs räcka med tio tester.

Överlag så upplevdes det som enkelt att utveckla med *React Native*. Det var en snabb startsträcka för att komma igång och problem som dök upp tog sällan speciellt lång tid att lösa. Detta berodde antagligen på bra dokumentation samt att det fanns så mycket information att tillgå. Att få en erfaren utvecklare att utveckla med *React Native* borde inte vara svårt.

4.3.2. Flutter

Utvecklingen av applikationen i *Flutter* delades upp i samma steg som för *React Native*. Under detta kapitel analyseras de tekniker som använts.

Precis som för *React Native* användes *IDE*:et *Visual Studio Code* för att dokumentera koden. Eftersom erfarenheten av verktyget redan började under utvecklingen av applikationen i *React Native* så underlättade det att koda denna applikation under samma förutsättningar.

Applikationen skrevs i språket *Dart* vilket är det officiella språket för *Flutter*.

Som simulator användes även här *iPhone*-simulatorn som ingår i *Xcode* för *MacOS*-användare. För att testa applikationen på en mobiltelefon med *Android* behövdes varje modell laddas ner separat till en simulator vilket upplevdes som både tids- och resurskrävande.

Datan från *API*:et hämtades med hjälp av en asynkron metod döpt till *getJson*. Metoden specificerades som något som *Flutter* döpt till *future* som kan användas när data ska hämtas från externa källor.

Layouten byggdes med en widget av typen *Scaffold* som innehåller widgetarna appbar och body. I body finns en widget av typen *ListView* som visar alla bilder. *Scaffold* är den widgeten som i regel all layout utgår från. *ListView* var enligt *Flutters* dokumentation den vanligaste widgeten för att designa listor som skrollas igenom. För den här listan valdes att visa bildobjektet med en vertikal lista för att variera lite från alternativet med *React Native* utan att det skulle påverka prestandan enligt handledarna.

Det tog i genomsnitt ca **3.5** sekunder för applikationen att hämta datan från *API*:et. Genomsnittet baserades på tio oberoende tester där

tiden för hämtningen loggades. Resultatet på testerna varierade från ca **3** sekunder till ca **3.6** sekunder. Eftersom testerna inte varierade så mycket valdes den här avrundningen och att det antogs räcka med tio tester. Värt att nämna här är att hämtningen tog betydligt längre jämfört med motsvarande för *React Native*.

Att utveckla i *Flutter* var på det stora hela inte speciellt svårt eller tidskrävande. Men att hitta information om det var ibland lite svårt. Detta kompenserades dock på det stora hela av att den officiella dokumentation som upplevdes som väldigt heltäckande.

4.3.3. Jämförelse av ramverken

I detta delkapitel analyseras hur det båda ramverken skiljer sig åt ur ett utvecklarperspektiv.

4.3.3.1. Behöver man några särskilda verktyg för att kunna utveckla?

Egentligen så behövs bara en dator med internetuppkoppling för att kunna ladda ner *Flutter* eller *React Native*. Båda ramverken är kompatibla med både *Windows*, *MacOS* och *Linux*. För att kunna utveckla applikationer till *iOS* krävs *MacOS* men för *Android* går det bra med alla operativsystem. Att använda sig av ett *IDE* som till exempel *Visual Studio Code* är en fördel vid utveckling men inget krav då all utveckling kan göras med en vanlig terminal och valfri texteditor.

Tanken med den här frågan var om det kunde vara så att en utvecklare behöver köpa någon särskild licens eller program för utvecklingen men så är inte fallet då allt är gratis. Det enda utvecklaren är beroende av är om hen ska utveckla till *iOS* och inte har tillgång till någon dator med *MacOS* så behöver en sådan införskaffas. Här dras slutsatsen att det inte skiljer något mellan ramverken.

4.3.3.2. Vilket programspråk använder man?

React Native kan skrivas i antingen *JavaScript* eller *JSX*. För *Flutter* använder man *Dart* vilket beror på att *Flutter* använder sig av *Darts* virtuella maskin för att kompilera koden. Någon dokumentation om att utveckla *Flutter*-applikationer med andra språk hittades ej.

Om man kommer från en bakgrund som webbutvecklare så har man troligtvis arbetat med *React* som baseras på *JavaScript* och *JSX*. Ur det perspektivet skulle det vara en snabbare övergång att skriva mobilapplikationer i *React Native*. Om man däremot arbetar som *Java*-utvecklare eller med andra objekt-orienterande programspråk så kan *Flutter* vara det bättre alternativet då syntaxen är mest lik. Men för de flesta utvecklare med många år i branschen så spelar nog inte språket en större roll då man är van att växla mellan olika tekniker. Som nybörjare är det inte någon större skillnad mellan de två olika programspråken. I slutändan bör valet av ramverk, med hänsyn till språket, dras utifrån utvecklarens individuella erfarenheter.

4.3.3.3. Hur lång tid tar det för respektive prototyp att kompilera?

För *Flutter* tog det i genomsnitt **33** sekunder att kompilera applikationen, baserades på tio försök. För *React Native* tog det i genomsnitt **24** sekunder för tio försök. Detta testades genom att en timer startades från det att respektive applikation startades tills dess att den var färdig-exekverad. För *Flutter* låg försöken på lite över **30** sekunder och för *React Native* lite över **20** sekunder. Här kan man konstatera en fördel för *React Native* vilket stämmer överens med resultaten från hur lång tid det tog för respektive ramverk att hämta datan från *API*:et (Se kap 4.1.4 och 4.2.4).

4.3.3.4. Är det någon skillnad i filstorlek på prototyperna?

För *Flutter* var storleken **323,3** MB och för *React Native* var storleken **645,3** MB. *React Native* var **322** MB större. Exakt vad detta beror på är svårt att säga, men jämförelsen i delkapitlet ovan konstaterar att det verkar finnas en korrelation i prestandan och filstorleken eftersom *React Native* var snabbare men också krävde större filstorlek.

4.3.3.5. Vad har andra utvecklare för erfarenhet av respektive ramverk?

När man ska jämföra vad andra utvecklare har för erfarenhet av ramverken så anses båda vara bra att jobba med. Det finns inga direkta för- eller nackdelar som sticker ut och båda verkar vara enkla att komma igång med. Denna slutsats baserades framförallt på olika

blogginlägg. Det svåra var dock att det inte fanns någon bra analys gjord av någon som hade lika mycket erfarenhet av båda ramverken. Anledningen till det beror enligt antaganden på två saker. Det ena är att *React Native* har funnits ett tag och medans *Flutter* är ett nyare fenomen som inte har hunnit bli lika använt av erfarna utvecklare. Det andra är att *React Native* är ett språk som anses vara lätt för webbutvecklare att sätta sig in i då dessa ofta använder sig av liknande ramverk som huvudsakligen kodas med *JavaScript*. Eftersom *Flutter* skiljer sig från dessa och inte använder *JavaScript* så lockar det inte webbutvecklare lika mycket. På grund av detta så skrivs artiklar om *React Native* oftast av webbutvecklare som gillar *JavaScript* medans artiklar om *Flutter* snarare författats av personer som gillar att koda i till exempel *Java*.

4.3.3.6. Hur stort är respektive community?

På *Stack Overflow* så har *React Native* över **56600** frågetrådar medan *Flutter* har över **21100**. På *Github* har *React Native* fått över **79900** stjärnor och repot har hämtats (fork) över **17900** gånger. För *Flutter* så motsvarade siffrorna över **72400** stjärnor och över **8700** hämtningar. En fördel för *React Native* med andra ord [34]. *Stack Overflow* är världens mest spridda forum när det kommer till programmering och vad som trendar där speglar ofta hur verkligheten ser ut. Samma sak gäller för *Github* som är världens största klient för världens största versionshanteringsteknik *git*. Både *Stack Overflow* och *GitHub* har dessutom en tradition av att ha det mesta öppet och publikt för allmänheten. När branscher tittar efter vilka programmeringsspråk och tekniker som ökat och/eller sjunkit i popularitet brukar de kolla på dessa forum.

4.4. Verktyg

Nedan analyseras de två verktygen *Visual Studio Code* och *Postman*.

4.4.1. Visual Studio Code

När utvecklingen påbörjades så användes först ett annat *IDE* som heter *Sublime* men handledarna på *Smart Refill* tipsade om att de själva tidigare hade använt *Sublime* men nu gått över till att använda *Visual Studio Code*. Efter detta byttes *Sublime* snabbt ut mot *Visual Studio Code* och det var enkelt att komma igång. Den stora fördelen som upplevdes med *Visual Studio Code* var att det hade en inbyggd terminal som förenklade utvecklingsprocessen då utvecklaren inte behövde skifta fönster mellan *IDE*:et och terminalen hela tiden.

4.4.2. Postman

Att använda *Postman* var väldigt enkelt då det bara var att mata in URL:en i dess användargränssnitt och ange vilka parametrar som skulle hämtas. Så här i efterhand kan det dock anses som onödigt att använda *Postman* då man hade kunnat testa *API*:et direkt i valfri webbläsare men det var lärorikt att testa då det är ett populärt verktyg i branschen.

4.5. Dokumentation

Under examensarbetets gång skrevs regelbundna minnesanteckningar på daglig basis med papper och penna. Dessa renskrevs sedan digitalt. Till en början så utfördes digitaliseringen av anteckningarna oregelbundet, när det fanns tid över, men allt efter som arbetet fortskred så blev det mer en rutin vilket också gjorde att det sedan skulle bli lättare att hitta bland dem. Att ha alla anteckningar samlade på ett ställe som var lättillgängligt underlättade. Metoden att föra ner

anteckningarna digitalt på ett mer regelbundet sätt kunde ha varit bättre planerat från början, exempelvis genom att sätta ett schema för när det skulle göras veckovis, men metoden som tillämpades fungerade ändå.

5. Resultat

I detta kapitel presenteras resultatet på problemformuleringarna. Inledningen berör resultatet från de två första:

- Hur kan man utveckla en prototyp av en mobilapplikation med *React Native* som hämtar data från ett öppet *API* och visar datan i prototypen på ett överskådligt sätt i form av en lista?
- Hur kan man utveckla en prototyp av en mobilapplikation med *Flutter* som hämtar data från ett öppet *API* och visar datan i prototypen på ett överskådligt sätt i form av en lista?

Följt av resultaten från de resterande problemformuleringarna.

5.1. Utvecklingen av prototyperna

I stora drag så påminner processerna för att komma igång och utveckla till *React Native* och *Flutter* om varandra. Processerna kan kortas ner till följande sex steg:

Steg 1 - Installera ramverket

I det första steget behöver man ladda ner och installera ramverket (SDK för *React Native* och *Flutter*) på datorn som ska användas. Detta var något svårare för *Flutter* än *React Native*. Att installera *React Native* var bara att följa instruktionerna som fanns på deras hemsida. Medan man för *Flutter* var tvungen att sätta *Flutter* som en global sökväg.

Steg 2 - Installera ett IDE

Ingen skillnad då samma *IDE* (*Visual Studio Code*) användes.

Steg 3 - Installera en simulator (*IOS* eller *Android*)

I det här steget så användes *IOS*-simulatorn.

Steg 4 - Skapa ett färdigt projektpaket

Detta görs på liknande sätt för båda ramverken. Utvecklaren går till den sökväg projektet ska ligga i via sin terminal och matar därefter in ett kommando som laddar ner projektpaketet automatiskt till sökvägen.

Steg 5 - Öppna projektpaketet i IDE:et

Därefter öppnas projektet i *IDE*:et.

Steg 6 - Utveckla i IDE:t och testa i simulatorn

I sista steget börjar själva kodskrivningen. Kompileringen av applikationen görs med den inbyggda terminalen i *Visual Studio Code*. Via ett kommando körs applikationen i simulatorn.

Stegen, för *React Native* och *Flutter,* förklaras mer detaljerat i de kommande två delkapitlen. Steg 1 och steg 2 är exakt samma för båda ramverken och därför görs ingen mer förklaring än den under kapitel 5.1.1.

5.1.1. React Native

Det första steget i utvecklingsfasen var att installera *React Native* på den datorn som skulle användas. Till hjälp följdes de instruktioner som finns att tillgå på *React Natives* officiella hemsida. För det här projektet valdes alternativet *React Native CLI Quickstart.* För detta krävs att beroendena *Node*, *Watchman*, *React Native command line interface* och *JDK* är installerade. Dessa installerades i sin tur med

hjälp av en pakethanterare som heter *Homebrew* som fungerar med *MacOS* och *Linux* men inte *Windows*. *Homebrew* är gratis och installeras i terminalen med hjälp av skriptet:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)”
```

Med *Homebrew* installerades därefter alla beroendena. *React Native* installerades slutligen med skriptet:

```
npm install -g react-native-cli
```

Nästa steg var att installera simulatorn. Eftersom det hade bestämts att detta skulle göras för *iOS* så skulle den simulatorn som ingår i *Xcode* användas. Den hämtas från *Mac App Store*. Även *IDE*:et som i detta fallet var *Visual Studio Code* laddades ner och installerades från dess officiella hemsida.

Med allt installerat är det nu dags att skapa ett projektpaket. Det görs genom att mata in skriptet:

```
react-native init valfritt_projektnamn
```

Därefter öppnas projektet med hjälp av *IDE*:et. Projektet innehåller flera olika filer. Filen som är intressant för utvecklaren här är den som heter *App.js*. Det är i den som själva utvecklingen sker och här skrevs programkoden för applikationen.

Källkoden (Appendix 9.1) bestod av metoderna: *constructor(props)*, *componentDidMount*, *getData* och *render*. Den innehöll även en konstant kallad *styles* som användes för layouten.

- *constructor(props)* är applikationens konstruktor.
- *componentDidMount* är en inbyggd metod i *React Natives* livscykel och som körs då applikationen har skapats och renderats färdigt för första gången. Det är vanligt att här hämta data från *API*:er. Funktionen kallar på *getData*.

- *getData* är funktionen som anropar *API*:et och hämtar datan.
- *render* rendrerar den grafiska vyn.

Att bygga och testa applikationen i simulatorn gjordes med skriptet:

```
react-native run-ios
```

5.1.2. Flutter

Det första steget i installationsprocessen är att installera *Flutter* lokalt. Detta görs genom att ladda ner ett paket från *Flutter*s officiella hemsida:

https://flutter.dev/docs/get-started/install/macos

Paketet kommer som en zipfil som får packas upp på valfritt ställe. Nästa steg som bör göras är att uppdatera sökvägen på datorn så att *Flutter* ligger som en global sökväg och kan användas i alla terminalfönster. Detta görs genom att först navigera till

```
$HOME/.bash_profile
```

Därefter ska skriptet:

```
export PATH=”$PATH:[PATH_TO_FLUTTER_GIT_DIRECTORY]/flutter/bin”
```

matas in i terminalen. För *Flutter* finns även ett inbyggt analysverktyg döpt till *Flutter doctor* som används för att undersöka om ens installation av *Flutter* är fullständig eller behöver uppdateras. Det användas med hjälp av skriptet:

```
flutter doctor.
```

Nästa steg är att skapa ett projekt som kan användas för att utveckla i. Detta görs här med skriptet:

```
flutter create valfritt_namn_på_applikationen
```

Vilket skapar upp ett programpaket som öppnas i det valda *IDE*:et. I paketet finns olika filer och den som är intressant här heter *main.dart*. I den skrevs sedan följande metoder som användes för att lösa uppgiften i problemformuleringen:

Källkoden(Appendix 9.2) bestod av metoderna: *main*, *runApp* och *getJson*.

- *main* är applikationens konstruktor.
- *runApp* startar applikationen. Den grafiska vyn ligger även här.
- *getJson* anropar *API*:et och hämtar datan.

Slutligen så körs applikationen genom att först starta simulatorn. Därefter matas skriptet:

```
flutter run
```

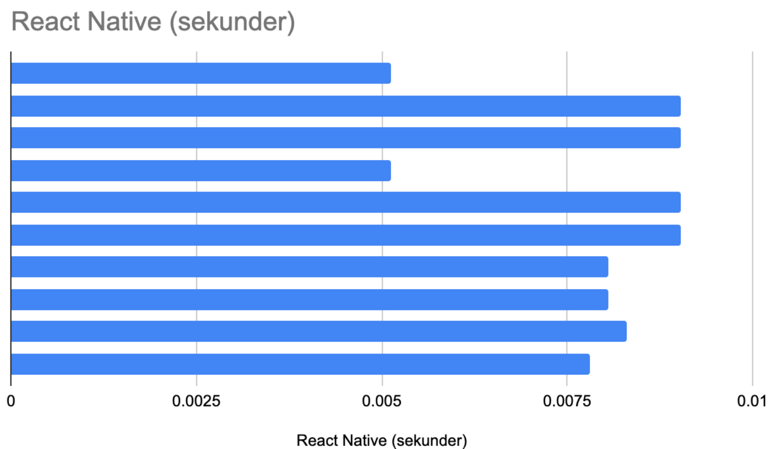
in i terminalen.

5.2. Hur lång tid tar det att hämta data från API:et?

Tiden för att hämta datan gjordes med hjälp av tio tester för respektive ramverk. Testen baserades på hur lång tid det tog för respektive applikation att exekveras och visa datan i simulatorn. Tiderna gavs från kompileringsloggar. Sammanfattningsvis så kan man se att *React Native* var betydligt snabbare. Eftersom det rörde sig om millisekunder så gör det inte så mycket på mindre applikationer men ska man utveckla mer prestandatunga applikationer kan man nog föredra *React Native* eftersom det då kan bli stora skillnader.

5.2.1. React Native

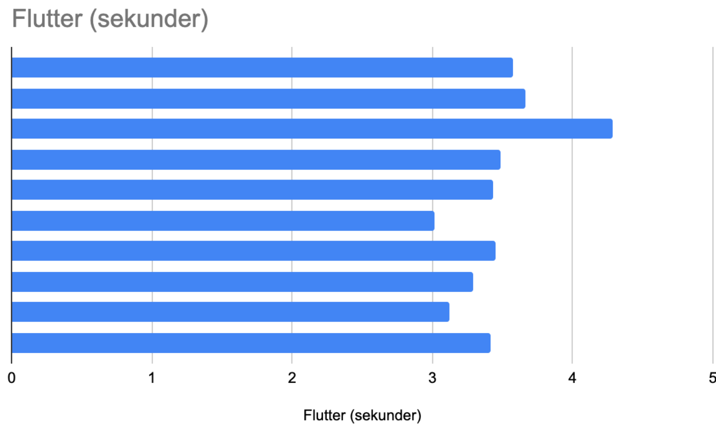
Det tog i genomsnitt **0.008** sekunder för applikationen att hämta datan. Här var det en klar fördel för *React Native* som i snitt är nästa **438** gånger snabbare än *Flutter*. Samtliga tester illustreras i diagrammet nedan (figur 5.1).



Figur 5.1 - Tider för *React Native* att hämta data från *API*:et

5.2.2. Flutter

Det tog i genomsnitt **3.5** sekunder för applikationen att hämta datan. Här var det förvånande att *Flutter* var så pass mycket långsammare än *React Native* och en klar nackdel för *Flutter*. Tiden för testerna illustreras av diagrammet nedan (figur 5.2).



Figur 5.2 - Tider för *Flutter* att hämta data från *API*:et

5.3. Hur skiljer sig prototyperna åt ur ett utvecklarperspektiv?

Efter att ha byggt de två applikationerna kan man redan se olika saker som skiljer ramverken åt. Först och främst så är *React Native* något smidigare att installera där det enda som behövde göras var ladda ner *HomeBrew* och sedan installera allt med hjälp av det ixnstallationsscript som tillhandahölls på dess hemsida. För *Flutter* var man tvungen att ställa in den globala sökvägen själv vilket *React Native* gör automatiskt. Analysverktyget *flutter doctor* är en liten fördel men i förhållande till att *React Native* var smidigare att installera och att det dessutom var betydligt snabbare i att kommunicera med *API*:et är något som skiljer ramverken åt till *React Native*s fördel.

5.3.1. Behöver man några särskilda verktyg för att kunna utveckla?

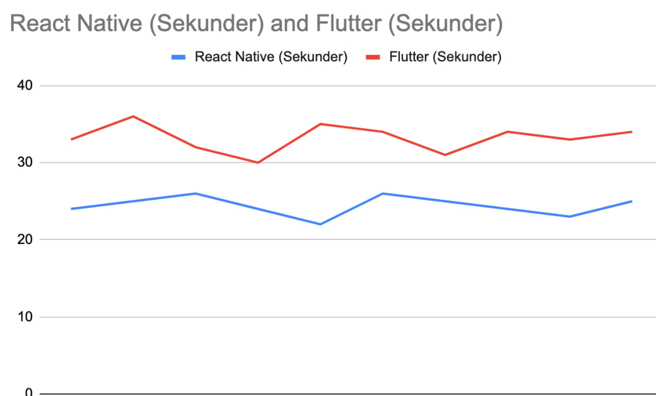
För att kunna utveckla respektive applikation behövs en dator med ramverket installerat på. En simulator för att testa applikationen kan också komma till hands. Ett *IDE* är inte nödvändigt men rekommenderas starkt. Det finns *IDE*:er att välja på som är gratis som t.ex *Visual Studio Code*. Här skiljer verkligen ingenting mellan ramverken.

5.3.2. Vilket programspråk använder man?

Dart är programmeringsspråket för *Flutter*. *JSX* eller *JavaScript* används för att utveckla i *React Native*. *Dart* anses av utvecklare, som ett språk som är lätt för programmerare med en bakgrund av andra objekt-orienterade språk, så som *Java* eller *C#*, att sätta sig in i. Det är dock fortfarande ett relativt okänt språk och utvecklare som ska jobba med det behöver nog oftast skolas in lite först. *JavaScript* är ett av de populäraste språken idag och det finns många utvecklare som besitter kunskaper inom det, vilket borde ge *React Native* en klar konkurrensfördel jämfört med *Flutter*.

5.3.3. Hur lång tid tar respektive prototyp att kompilera?

Applikationen utvecklad med *Flutter* tog det i snitt **33** sekunder att kompilera. *React Native* applikationen tog **24** sekunder att kompilera och var därmed cirka **1.38** gånger snabbare. Detta baserades på tio olika försök för respektive ramverk. Här kan vi återigen se en klar fördel för *React Native* som illustreras i figur 5.3. Tiden kunde erhållas från *IDE*:et *Visual Studio Code*.



Figur 5.3 - Kompileringstider för *React Native* och *Flutter*

5.3.4. Är det någon skillnad i filstorlek på prototyperna?

Prototypen byggd med *Flutter* var **323,3** MB stor. Prototypen byggd med *React Native* var **645,4** MB stor. *React Native* var således **322** MB större. Här är första gången *Flutter* visar på en fördel gentemot *React Native*. Vad detta exakt beror på är svårt att säga. Eftersom smartphones kan ha begränsat diskutrymme på sina hårddiskar så kan detta vara något att ta i beaktning om man ska utveckla en applikation.

5.3.5. Vad har andra utvecklare för erfarenhet av respektive ramverk?

Det är svårt att avgöra men utvecklare som har sitt kompetensområde inom webbutveckling verkar föredra *React Native* medans utvecklare som främst jobbar med andra delar än webbutveckling är mer intresserade av *Flutter*. Av att läsa olika artiklar från personer i branschen verkar det som att många är eniga om att det finns många fördelar med *React Native*. En av fördelarna är att det finns fler

utvecklare som kan jobba i *JavaScript* än som kan jobba med *Dart* [35]. Att det finns flera stora applikationer idag som körs med *React Native* och att det finns ett större community kring ramverket är något som också verkar uppskattas [36].

5.3.6. Hur stort är respektive community?

På *Stack Overflow* finns **53300** trådar om *React Native* och **17700** trådar för *Flutter*. På *Github* har *React Native* **78200** stjärnor och *Flutter* har **67500**. *React Natives* repo har hämtats ca **17500** gånger och *Flutters* har hämtats ca **67500** gånger. Det är kanske orättvist att jämföra då det *React Native* har fått leva under en längre tid och därför hunnit etablera sig mer. *Flutter* växer stadigt och har en trogen fanskara men i förhållande till *React Native* så ligger det efter i popularitet.

6. Slutsats

Examensarbetet resulterade i två stycken prototyper som använder det öppna *API*:et för öltillverkaren *Brewdog*. Tiden att hämta datan var så mycket kortare för *React Native*s vilket var förvånande. Att bygga applikationerna var inte speciellt svårt med något av ramverken. Vid första anblick känns det inte som att det skulle vara någon större omställning för en erfaren utvecklare som jobbat med ett *Native* ramverk att gå över till *React Native* eller *Flutter*. Det som framförallt skiljer dem åt är att *React Native* har ett större community vilket innebär att det oftast är lättare att hitta information och hjälp. Om några år kommer *Flutter* antagligen ta en större andel då ramverket redan har ett stabilt växande community. Är man dessutom en erfaren utvecklare som är van vid att testa nya programmeringsspråk och verktyg så kanske man inte har samma behov av ett utbrett nätverk.

Båda är rätt snarlika när de delar upp hela det grafiska gränssnittet utifrån olika ”beståndsdelar” och utformar appen efter dessa. Att kunna bryta ner en applikation på detta sätt underlättar för en utvecklare då man kan fokusera på en del i taget.

En fördel med båda teknikerna jämfört med *Native* är att de tillämpar så kallad *Hot Reload* vilket innebär att man som utvecklare direkt får feedback när man gör en ändring i sin applikation. Detta innebär att en kompilering sker på bara någon sekund vilket, jämfört med för en *Native* applikation då kompilering ibland kan ta flera minuter om man har gjort stora ändringar i sin kod, sparar mycket värdefull tid.

En fördel med *React Native* är att tröskeln för att gå över till detta ramverk anses vara lägre för webbutvecklare eftersom de ofta jobbar med snarlika ramverk och bibliotek exempelvis *React*, *Vue* och *Angular*, där alla baseras på *JavaScript*. På så sätt öppnar det upp ett

större spektrum av utvecklare för företag så som *Smart Refill*, som vill utveckla *Cross Platform* applikationer, att rekrytera jämfört med om företagen bara skulle fokusera på att rekrytera mobil-utvecklare.

För utvecklare som däremot tidigare använt ett *Native* språk så kan nog steget till *Flutter* kännas naturligare. En annan fördel med *Flutter* var att storleken på de färdiga applikationerna verkar vara mindre. *Flutter* borde även vara optimalt för utveckling till *Android* eftersom *Google* ansvarar för både *Flutter* och *Android*. Detta är dock inget som har visat sig under examensarbetet men det känns ändå naturligt att det borde vara så.

Att skriva kod till applikationerna visade sig vara relativt enkelt. Båda ramverken hade bra guider som gjorde att det var lätt att komma igång och programmera. Det var enkelt att installera ramverken och *IDE*:et, *Visual Studio Code*, användes till båda utvecklingsprocesserna. För någon som inte har speciellt mycket erfarenhet av något av ramverken så är det rätt enkelt att sätta sig in i dem och komma igång.

Examensarbetet gick ut på att se om företag som utvecklar *Native* kan gå över till *Cross Platform*. Detta beror på vad för typ av applikationer företaget i nuläget utvecklar. Är det avancerade 3D-applikationer, kommer förmodligen inte *Cross Platform* vara rätt väg att gå eftersom *Flutter* och *React Native* har som primärt syfte att utveckla applikationer i 2D. Utvecklar man däremot konventionella applikationer så bör man som företag titta på dessa ramverk. Att få en utvecklare att gå över till *React Native* eller *Flutter* bör som sagt inte vara något problem. Det man främst ska ha i åtanke om man väljer mellan dem är att: *Flutter* har mer solida egna komponenter medan *React Native* har rätt många komponenter som kommer från tredjepartstillverkare. Det behöver inte vara dåligt men kan kännas mindre tillförlitligt. Däremot så har *React Native* funnits längre och

har ett större community om man tittar på *Stack Overflow* eller *GitHub*. *Flutters* community växer dock så om något år är det mycket möjligt att det ser helt annorlunda ut.

Ytterligare en sak att ha i åtanke är att *Flutter* kommer från *Google* som ju naturligtvis har ett intresse av att ta andelar från *Apple* och bli fortsatt dominerande på marknaden. Det vill säga *Flutter* kommer från det ena av de två dominerande bolagen på mobilmarknaden.

I nuläget har *React Native* en dominerande ställning samtidigt som *Flutter* växer snabbt. Men vad skulle hända om *Flutter* plötsligt slutar växa? Skulle *Google* i så fall tappa intresset för *Flutter* och kanske inte bry sig om att utveckla det längre? I så fall hade man kanske satsat stora resurser på ett ramverk som inte längre underhålls. Historiskt kan man ju se liknande exempel på när *Google* försökt ta andelar från andra aktörer till exempel som när de försökte lansera *Google Plus* som tänkt *Facebook*-dödare. Men så blev inte fallet och nu verkar det inte som att *Google* prioriterar sitt sociala nätverk. Skulle detta även kunna hända med *Flutter*?

Ska ett företag idag välja en *Cross Platform* teknik så borde det välja *React Native* framför *Flutter*. Kombinationen av att det har ett större community än *Flutter* samt att det använder *JavaScript* är två mycket starka argument som avgör detta val till *React Native*s fördel.

6.1. Reflektioner över etiska aspekter

Det verkliga syftet med de två *Cross Platform*-teknikerna är egentligen att aktörerna bakom dem vill bli en marknadsledande spelare som äger hela kretsloppet, det vill säga som står bakom all mjukvara. *React Native* och *Flutter* kommer från *Facebook* och

Google som precis som sina konkurrenter vill bli dominerande på marknaden. Ramverken är gratis och *Open Source* men i bakgrunden så finns det en stor aktör som är intresserad av att dominera och vill att andra aktörer gör sig beroende av dem. Det måste inte nödvändigtvis vara dåligt men det är viktigt att inte glömma bort. Det andra lösningarna som finns idag tillhandahålls dock också av andra aktörer så som *Oracle* med *Java* eller *Apple* med *Swift* och *Objective-C*. Det gör inte respektive val av teknik bättre eller sämre eftersom alla har samma agenda och intresse av att tjäna pengar. I bakgrunden finns det alltid ett affärsmannamässigt syfte och även om alla dessa aktörerna påstår att de vill göra världen bättre så finns det alltid ett ägarintresse någonstans i bakgrunden som man inte bör vara naiv inför.

6.2. Framtida utvecklingsmöjligheter

Eftersom examensarbetet endast resulterade i två prototyper som testades i en lokal simulator så finns det flera framtida utvecklingsmöjligheter. Ett första steg skulle kunna vara att få prototypen att fungera på en riktig smartphone. En annan möjlighet skulle kunna vara att ändra applikationen så att användaren först får en lista över möjliga parametrar att mata in för på så sätt få ut flera olika listor. Ett exempel på detta skulle kunna vara en lista över alla de maträtter som finns att välja på i *API*:et för att på så sätt påverka vilka ölsorter som kommer tillbaka efter anropet. Ytterligare ett sätt man skulle kunna göra är att bygga samma applikation med ett *Native* ramverk och jämföra prestandan mellan dessa.

7. Källförteckning

[1] Smart Refill. *Om oss. 2019.* https://www.smartrefill.se/about-us/ (Hämtad 2019-03-23)

[2] Alla bolag. Sökning på Smart Refill. *Smart Refill.* https://www.allabolag.se/5566689922/smartrefill-i-helsingborg-ab (Hämtad 2019-03-23)

[3] Hackernoon. *Native-app-development-vs-hybrid-app-development 2019.* https://hackernoon.com/native-app-development-vs-hybrid-app-development-dd83122a738c (Hämtad 2019-03-25)

[4] Codeburst. *native-vs-cross-platform-app-development-pros-and-cons 2019.* https://codeburst.io/native-vs-cross-platform-app-development-pros-and-cons-49f397bb38ac (Hämtad 2019-03-25)

[5] Flutter. Officiell startsida. 2019. https://flutter.dev/ (Hämtad 2019-03-26)

[6] React Native. Officiell startsida. 2019. https://facebook.github.io/react-native/ (Hämtad 2019-03-26)

[7] Medium. Artikel om vad ett API är. *what-is-an-api-in-english-please.* 2019. https://www.freecodecamp.org/news/what-is-an-api-in-english-please-b880a3214a82/ (Hämtad 2019-04-6)

[8] Medium. Artikel om React Native och Flutter. *react-native-or-flutter-which-should-i-choose.* 2019. https://medium.com/flutter-community/react-native-or-flutter-which-should-i-choose-48567ae2e5e1 (Hämtad 2019-04-6)

[9] Svenska Dagbladet. Artikel om att det är brist på utvecklare. *brist-pa-programmerare.* 2019. https://www.svd.se/brist-pa-programmerare--saknas-en-miljon-i-eu (Hämtad 2019-04-6)

[10] Brookshear, J. Glenn., Brylow, Dennis. 2015. Computer Science An Overview. 12. uppl. Pearson Education Limited. 334

[11] BBC. Artikel om IDE. https://www.bbc.co.uk/bitesize/guides/zgmpr82/revision/4 (Hämtad 2019-04-07)

[12] Visual Studio. Officiell dokumentation. https://code.visualstudio.com/docs (Hämtad 2019-04-07)

[13] Apple. Officiell Hemsida om utveckling. *ios* . https://developer.apple.com/ios/ (Hämtad 2019-04-07)

[14] Android. Officiell Hemsida. https://www.android.com/ (Hämtad 2019-04-07)

[15] Hackernoon. Artikel om Native utveckling och hybrid utveckling. *native-app-development-vs-hybrid-app-development.* https://hackernoon.com/native-app-development-vs-hybrid-app-development-dd83122a738c (Hämtad 2019-04-10)

[16] Apple. Dokumentation om Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html (Hämtad 2019-04-10)

[17] Apple. Dokumentation om Swift. https://developer.apple.com/swift/ (Hämtad 2019-04-10)

[18] Hackernoon. Artikel om Java och Kotlin. https://hackernoon.com/java-vs-kotlin-its-time-to-expand-android-development-f08e3d6a72b6 (Hämtad 2019-04-10)

[19] PC Magazine. Definition av Cross Platform. https://www.pcmag.com/encyclopedia/term/40495/cross-platform#fbid=aHfb3ldkqPq (Hämtad 2019-04-12)

[20] React Native. Introduktion av JSX. https://reactjs.org/docs/introducing-jsx.html (Hämtad 2019-04-11)

[21] Facebook. Lista på vilka som använder React Native. https://facebook.github.io/react-native/showcase (Hämtad 2019-04-14)

[22] O'reilly. Artikel om Dart. http://radar.oreilly.com/2012/03/what-is-dart.html (Hämtad 2019-04-14)

[23] Hackernoon. Artikel om att utveckla Cros Platform. https://hackernoon.com/flutter-pros-and-cons-for-seamless-cross-platform-development-c81bde5a4083 (Hämtad 2019-04-14)

[24] Brookshear, J. Glenn., Brylow, Dennis. 2015. Computer Science An Overview. 12. uppl. Pearson Education Limited. 189

[25] Postman. Offiell hemsida där Postman kan laddas ner. https://www.getpostman.com/postman?_ga=2.116792847.863258506.1559469143-870507050.1559217489 (Hämtad 2019-04-15)

[26] Geeksforgeeks. Introduktion om hur postman används. https://www.geeksforgeeks.org/introduction-postman-api-development/ (Hämtad 2019-04-15)

[27] Scrum. Förklaring om vad Scrum är. https://www.scrum.org/resources/what-is-scrum (Hämtad 2019-05-12)

[28] Udemy. Hemsida för kursen *learn-flutter-dart-to-build-ios-android-app*. https://www.udemy.com/course/learn-flutter-dart-to-build-ios-android-apps/ (Hämtad 2018-07-12)

[29] Udemy. Hemsida för kursen *React-native-the-practical-guide*. https://www.udemy.com/react-native-the-practical-guide/ (Hämtad 2018-07-12)

[30] Facebook. Dokumentation över hur React Native installeras. https://facebook.github.io/react-native/docs/getting-started (Hämtad 2018-07-12)

[31] Flutter. Dokumentation över hur Flutter installeras. https://flutter.dev/docs/get-started/install/macos (Hämtad 2018-07-12)

[32] Brewdog. Officiell hemsida om företaget. https://www.brewdog.com/about (Hämtad 2018-07-15)

[33] Brewdog. Dokumentation över dess officiella API. https://punkapi.com/documentation/v2 (Hämtad 2018-07-15)

[34] Stackshare. Jämförelse mellan React Native och Flutter. https://stackshare.io/stackups/flutter-vs-react-native (Hämtad 2019-08-11)

[35] Medium. Artikel om jämförelse av React Native och Flutter. https://medium.com/@adhithiravi/react-native-vs-flutter-what-are-the-differences-b6dc892f0d34 (Hämtad 2019-08-11)

[36] Hackernoon. Artikel om jämförelse av React Native och Flutter. https://hackernoon.com/react-native-vs-flutter-which-is-preferred-for-you-bba108f808 (Hämtad 2019-08-11)

8. Terminologi

8.1. JSON

JSON står för *Javascript Object Notation* och är ett standardformat för att representera strukturerad data i *JavaScript*. *JSON* används för skicka data mellan applikationer över internet.

8.2. StackOverFlow

StackOverflow.com är ett frågeforum för alla typer av frågor relaterade till programmering.

8.3. CSS-fil

En *CSS*-fil används för att anpassa det grafiska innehållet på en hemsida. Filen kan definiera storlek, färg, typsnitt, indentering, inramning och lokalisering av olika element.

8.4. SCRUM

Scrum är ett agilt ramverk som kan användas för att genomföra ett projekt av komplex karaktär. Syftet med att använda detta är att få en grupp att effektivt samarbeta och lösa svåra uppgifter. Detta görs genom att projektet som genomföras delas upp i olika tidsintervaller, kallat sprintar, som planeras i förväg. Varje dag under projektets gång har projektgruppen en genomgång av vad varje deltagare gör för att få feedback och eventuell hjälp från övriga deltagare. Genomgången kallas för en daily standup. De uppgifterna som sedan inte hinner bli färdiga under sprinten hamnar därefter i projektets backlog. Därefter påbörjas en ny sprint med nya uppgifter som tagits fram från backloggen.

8.5. Asynkron kommunikation

Asynkron kommunikation innebär att en komponent interagerar med en annan komponent utan den andra komponenten ger direkt feedback på interaktionen. Som exempel kan man jämföra med två mobiltelefoner som kommunicerar med varandra. Ett sms är en typ av asynkron kommunikation medans att ringa ett samtal istället är ett exempel på synkron kommunikation som är motsatsen till asynkron kommunikation.

8.6. Node

Node är ett gratis verktyg som exekverar *JavaScript.* I *Node* ingår en pakethanterare kallad *npm* som används för att installera beroenden.

8.7. JDK

Står för *Java Development Kit* och är den plattform som installeras för att kunna bygga och exekvera kod skriven i *Java*.

9. Appendix

9.1. React Native

```
import React, {Component} from 'react';
import {Platform, StyleSheet, Text, View,
Image, ScrollView} from 'react-native';
import 'react-native-console-time-polyfill';
const instructions = Platform.select({
  ios: 'Press Cmd+R to reload,\n' + 'Cmd+D or
shake for dev menu',
  android:
    'Double tap R on your keyboard to reload,
\n' +
    'Shake or press menu button for dev menu',
});
export default class App extends Component {
  constructor(props) {
    super(props);
    console.time(`${this.constructor.name}
init`);
    console.timeEnd(`${this.constructor.name}
init`);
    this.state = {
      beers: []
    }
  }
```

```
  componentDidMount() {
    this.getData();
  }
  getData() {
    fetch('https://api.punkapi.com/v2/beers?
food=fish')
      .then(res => res.json())
      .then(res => this.setState({ beers:
res }));
  }

  render() {
    const beers = this.state.beers.map((beer,
index) => {
      return (
        <View style={styles.beer} key={index}>
          {beer.image_url && <Image
            style={{flex:1, height: undefined,
width: undefined}}
            source={{uri: beer.image_url}}
            resizeMode="contain"
          />}
          <Text style={styles.text}
>{beer.name}</Text>
        </View>
      );
    });
    return (
      <ScrollView style={styles.scroll}>
```

```
        <View style={styles.header}>
          <Text style={styles.heading}
>BrewDog</Text>
          <Image
            style={{height: 24, width: 28,
marginRight: 10}}
            source={require('./heart.png')}
          />
          <Image
            style={{height: 24, width: 45}}
            source={require('./fish.png')}
          />
        </View>
        <View style={styles.container}>
          {beers}
        </View>
      </ScrollView>
    );
  }
}
const styles = StyleSheet.create({
  header: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    padding: 30,
  },
  heading: {
```

```
    paddingRight: 10,
    fontSize: 30,
    fontWeight: 'bold',
  },
  scroll: {
    backgroundColor: '#F5F5F5',
  },
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'space-around',
    flexWrap: 'wrap',
  },
  beer: {
    paddingTop: 25,
    paddingBottom: 25,
    height: 250,
    width: 150,
  },
  text: {
    paddingTop: 20,
    color: '#2B2C34',
    fontSize: 16,
    fontWeight: 'bold',
    textAlign: 'center',
  },
});
```

9.2. Flutter

```
import 'package:flutter/material.dart';
import 'dart:async';
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'dart:convert' show jsonDecode;
import 'package:time_log/time_log.dart' as
time_log;
void main() async {
List _data = await getJson();
String _description = "";
for(int i = 0; i < _data.length; i++) {
  print("Name: ${_data[i]['name']}");
  print("image url: ${_data[i]
['image_url']}");
}
_description= _data[0]['description'];
  runApp(new MaterialApp(

    home: new Scaffold(
      appBar: new AppBar(
        title: new Text('BrewDog'),
    ),
    body: new Center(
      child: new ListView.builder(
        itemCount: _data.length,
        padding: const EdgeInsets.all(14.5),
        itemBuilder: (BuildContext context,
int position){
```

```
          return new Column(
            children: <Widget>[
              new Divider(height: 50.5),
              new ListTile(
                title: Text("${_data[position]
['name']}"),
                leading: new CircleAvatar(
                  backgroundImage:
NetworkImage(_data[position]['image_url']),
                ),
              )
            ],
          );
        })
      ),
    ),
  ));
}
Future<List> getJson() async {
  String apiUrl = 'https://api.punkapi.com/v2/
beers?food=fish';
  http.Response response = await
http.get(apiUrl);
  return jsonDecode(response.body);
}
```