Rocket League Companion

von

Andreas Klar

Mobile und verteilte Systeme

Technische Hochschule Bingen

2018

Inhalt

Einführung

Die Android Applikation „Rocket League Companion“ ist ein Projekt für das Modul „Mobile und Verteilte Systeme“ an der TH Bingen aus dem Jahr 2018.

Die Zielgruppe besteht aus Spielern des Videospiels „Rocket League“, die sich für ihre eigene Spielstatistik und Entwicklung interessieren. Die App zeigt dem Nutzer sämtliche relevanten Daten, die aus dem Spielerprofil zu ermitteln sind. Das Hauptaugenmerk liegt dabei auf der Entwicklung des einzelnen Spielers und seiner persönlichen Statistik.

Was ist Rocket League?

Rocket League ist ein sehr schnelles Multiplayerspiel, bei dem der Spieler ein Auto steuert und versucht den Spielball im gegnerischen Tor unterzubringen. Gespielt wird 3vs3, 2vs2 oder 1vs1 auf einem mit Fußball vergleichbarem Spielfeld. Allerdings mit Wänden und einer Decke.

Die Autos sind mit einem Raketenantrieb ausgestattet und können in der Luft frei gesteuert werden. Dadurch können geübte Spieler kurze Strecken durch die Luft fliegen und präzise Schüsse und Pässe abgeben.

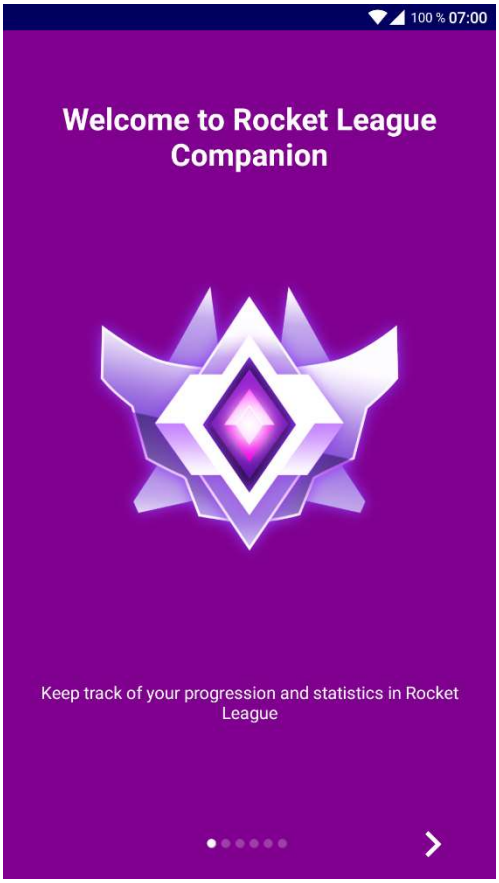
Ein Match dauert 5 Minuten, mit Stoppuhr. Bei Spielbeginn und nach Toren werden Ball und Spieler auf Startpositionen gesetzt und eilen zum Ball in die Mitte des Spielfelds. Das Spiel endet, wenn die Zeit abgelaufen ist und der Ball den Boden berührt. Bei Unentschieden wird erneut angestoßen und „Golden Goal“ gespielt, wer als erstes trifft gewinnt.

Ein Beispiel Clip aus der Universitätsliga:

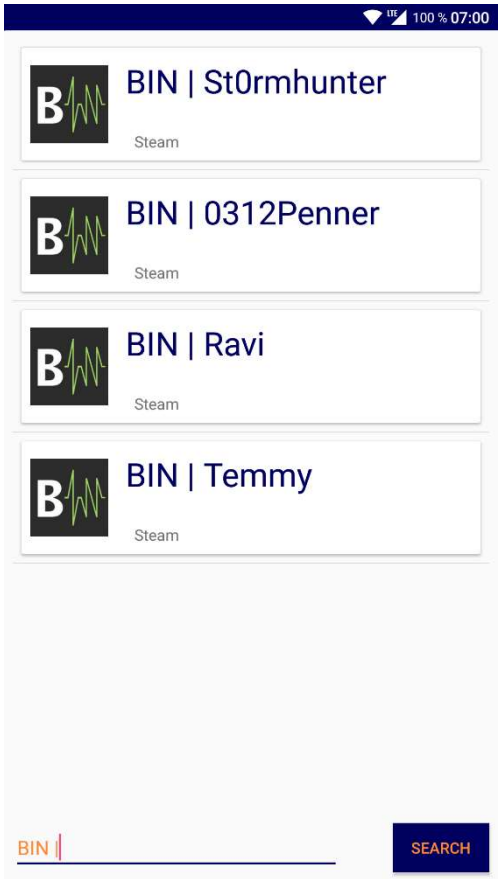
https://clips.twitch.tv/ThoughtfulPatientHareBabyRage

Mit „Rocket League Companion“ kann der Nutzer seinen Spielstil analysieren, seine Trefferrate und sein Ranking Verlauf betrachten und daraus Schlüsse bezüglich seines Trainings oder anderer äußeren Umstände ziehen.

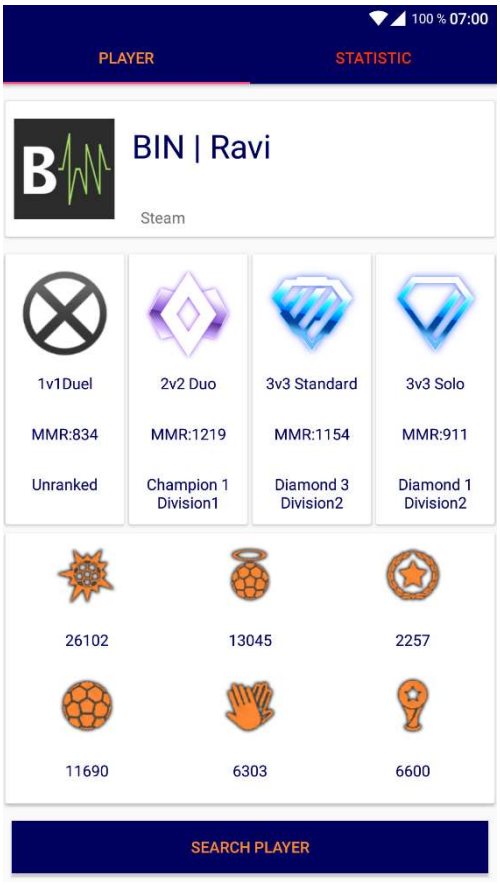
Überblick



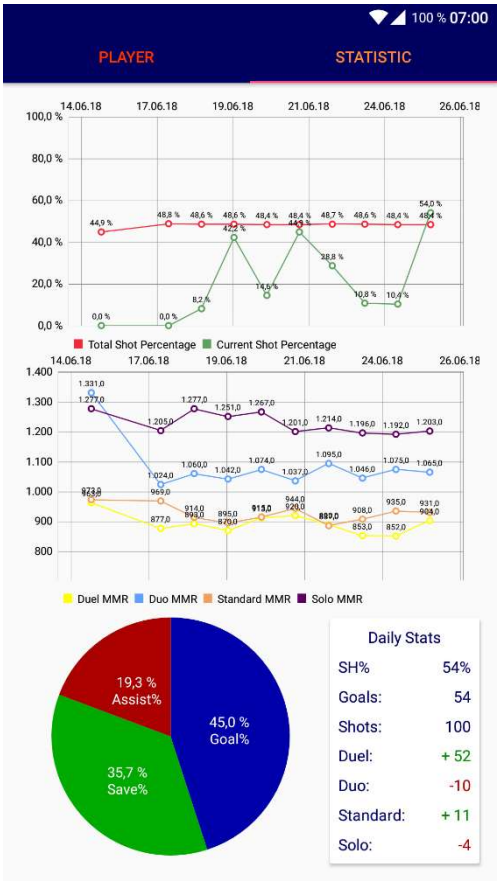
Beim Erststart erhält der Nutzer eine kurze Übersicht über die Funktionen und Anzeigen der App



Mithilfe der Suche kann der Spieler sein Profil finden und auswählen



Auf der Hauptseite findet der Spieler rohe Daten zu sich, wie Ranking und Statistiken. In jeder CardView befindet sich der aktuelle Rang mit passendem Bild, samt Name und genauem Ranking. Im unteren Segment sind Tore, Paraden, MVPs, Schüsse, Assists und Siege aufgelistet.



Hier findet der Spieler einen Verlauf seiner Trefferrate an diesem Tag und insgesamt, sowie seines Rankings in den verfügbaren Warteschlangen. Zuletzt gibt es ein Kuchendiagramm, an dem er seinen Spielstil erkennen kann und eine CardView, die tägliche Veränderungen anzeigt.

Funktionsweise

Suche

Die Suche funktioniert simpel. Der Nutzer gibt einen Suchbegriff ein, der dann per passendem Request an den API-Server gesendet wird. Dieser antwortet dann mit einem JSONArray das Spieler beinhaltet die der Server für passend erachtet. Daraus generiert die App Player Objekte, die dann in der ListView angezeigt und ausgewählt werden können. Durch Antippen eines Spielers wird dieser gesetzt und man gelangt in die Spielerübersicht. Einmal ausgewählt Spieler werden beim erneuten öffnen der Suche direkt als Auswahl angeboten.

Spielerübersicht

Nach der Auswahl wird direkt eine Anfrage an den Server gestellt, um Details zum Spieler zu ermitteln, sie wird dann in einem festen Zyklus wiederholt.

Die PlayerOverview hat einen Spieler, dieser Spieler kennt seinen Namen, ID, Statistiken…

Er hat außerdem eine Map mit Timestamps, diese Timestamps sind Schnappschüsse der Statistiken dieses Spielers. In der Map wird für jeden aktiven Tag ein Timestamp gespeichert. Ist bereits einer vorhanden wird er ersetzt.

Würden für jede Anfrage ein zusätzlicher Timestamp erzeugt und angezeigt werden, hätte man innerhalb von einer Stunde etwa 100 Einträge ohne Aussagekraft. Die Hälfte ohne Änderungen und die anderen mit extremen Trefferraten. Deswegen wird genau ein Timestamp pro Tag gespeichert und angezeigt.

Die Map ermöglicht es den nächst älteren Timestamp zu einem gegebenen zu finden. Dies ist nötig, um Tageswerte zu ermitteln. Also wie viele Tore wurden heute erzielt, wie hoch war die tägliche Trefferrate und so weiter.

In jedem Zyklus wird erneut eine Anfrage gestellt und mit der Antwort der Spieler und das Interface geupdatet.

Speichern

Einmal ausgewählte Spieler werden lokal gespeichert, um eine Schnellauswahl zu ermöglichen. Viele Nutzer werden sich selbst und ihre Freunde betrachten wollen, deswegen wird die Auswahl gespeichert und bei der Rückkehr auf die SearchActivity angezeigt.

Spielerobjekte werden per ObjektOutputstream mit ihrer ID abgelegt. Ein Spielerwechsel löscht damit nicht alle gesammelten Daten und sie können beim erneuten Auswählen wiederverwendet werden.

Bei der Auswahl eines Spielers in der Suche, versucht die App bestehende Daten zu Laden und diese danach anzuzeigen. Danach werden aktuelle Daten vom Server angefragt, verarbeitet und direkt gespeichert.

Layout

Die App teilt sich in zwei Activities. Die erste ist die SearchActivity, in der der Nutzer seinen Spielaccount suchen und Auswählen kann. Sie beinhaltet ein Textfeld, einen Button, eine Ladeanzeige, wenn gesucht wird und eine ListView zum Anzeigen der Resultate. Die Anordnung ist mit einem ConstraintLayout umgesetzt. Konnte kein Spieler gefunden werden wird der Nutzer per Toast darüber informiert. Auf den Einträgen der Liste sieht der Nutzer die Profilbilder, den Namen und die Plattform der Suchresultate.

Die zweite Activity ist der Kern der App. Sie besteht aus zwei Fragmenten in einem TabbedLayout. Zum einen die Spielerübersicht und zum anderen die Statistikanzeige.

Die Spielerübersicht besteht aus Cards in denen verwandte Elemente gruppiert sind. Dadurch werden die Daten in einzelne Bereiche geteilt. Sowohl in den Cards als auch die Anordnung der Cards verwende ich wieder das ConstraintLayout. Oben befindet sich die PlayerCard mit Bild, Namen und Plattform des Spielers, darunter vier Cards zu jeweils einer Warteschlange. Mit Rating, Liga, Beschreibung und Bild. Im unteren Bereich werden die rohen Daten, wie Schüsse, Tore, Siege… aufgelistet, jeweils mit dem im Spiel dazugehörigen Logo.

Die Statistikseite besteht aus drei Charts. Oben werden Prozentwerte angezeigt, in der Mitte absolute Rating Werte und im untern Bereich ein Verhältnis von Toren, Assists und Paraden. Die Graphen zeigen einen Zeitabschnitt über die aktuell vorhandenen Daten an mit jeweils einem Tag Puffer an den Enden.

In den Prozentgraphen werden die Gesamttore/Gesamtschüsse und Tagestore/Tagesschüsse (Shooting Percentage) angezeigt. Der Spieler sieht so wie gut er an einem Tag getroffen hat im Vergleich mit vorherigen Tagen und mit seinem Durchschnittswert.

Der mittlere Graph zeigt einfach die letzten Ratings eines Tages für jede der vier Warteschlangen an.

Das Kuchendiagramm zeigt die prozentualen Anteile von Toren, Assists und Paraden an deren Summe.

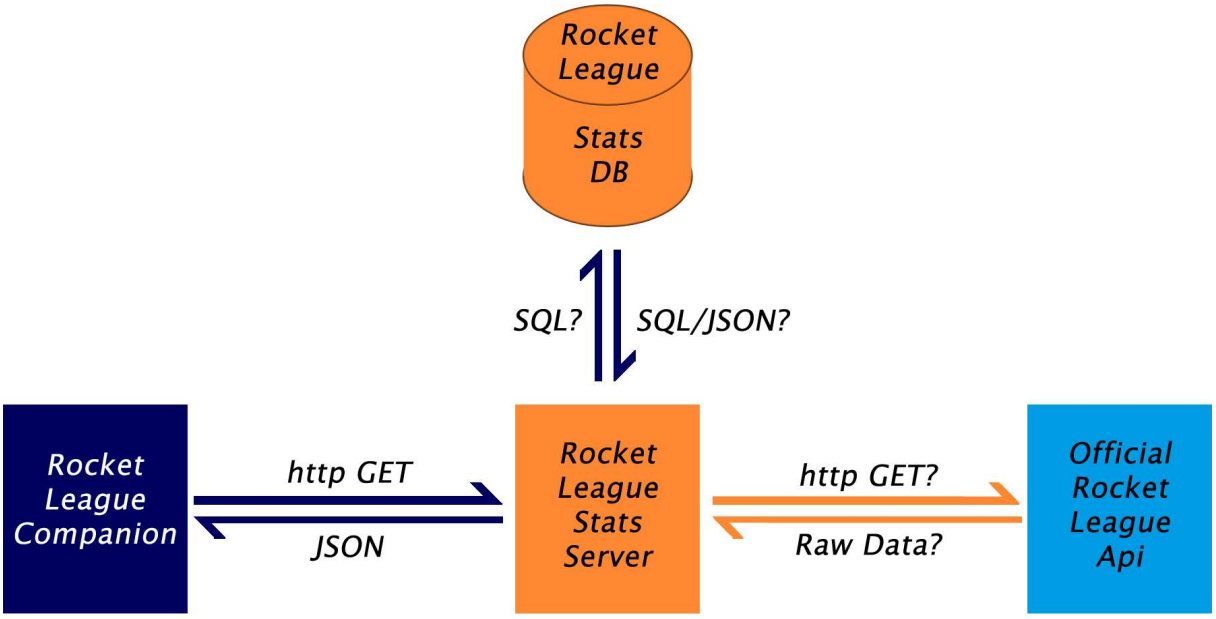
API und Vernetzung

System Architektur

Ich verwende die Daten der API von www.rocketleaguestats.com. Es existiert eine offizielle API des Entwicklers von Rocket League, allerdings befindet sich diese in der Closed Beta und man bekommt schwer Zugriff darauf. Die offizielle API scheint nur rohe Daten anzubieten, die schwer zu verarbeiten sind. Deswegen verwende ich die API eines anderen Anbieters.

Rocketleaguestats verwendet allerdings die offizielle API, dies ist daran zu erkennen, dass keine neuen Daten geladen werden können, wenn der Entwickler Psyonix in den Stoßzeiten den API Zugriff blockiert, um Serverlast zu sparen. Zusätzlich gibt es bei jeder Response von rocketleaguestats einen Eintrag „nextupdate“ und „updatedat“. Daraus schließe ich das zyklisch Daten von der offiziellen API angefragt und in einer Datenbank gespeichert werden.

Es ergibt sich folgende Vernetzung dadurch:



Die App startet asynchron einen http-Request, um aktuelle Daten zu erhalten. Der Server holt die angeforderten Daten aus seiner lokalen Datenbank und sendet sie zurück. Die App verarbeitet diese und aktualisiert das Interface. Dieser Vorgang wird zyklisch wiederholt.

Dabei handelt es sich um eine zentralisierte Client-Server Architektur. Die App fungiert als Client, der API-Server als Server. Da Daten von einer einzigen Schnittstelle bezogen werden müssen, ist dies die einzig sinnvolle Systemarchitektur dafür. Präziser dürfte es sich sogar, um eine Multitier Architektur handeln, allerdings habe ich keinen Einblick hinter die API.

Die Spielersuche und die Datenaktualisierung für einen Spieler sind asynchrone Anfragen. Die Suche wird vom Nutzer ausgelöst, die Aktualisierung zyklisch nachdem ein Spieler gewählt wurde.

Die App sendet die Anfrage an den Server und erwartet dann die Antwort, allerdings in einem Hintergrundthread, damit die App auch weiterhin bedient werden kann. Wenn einer Antwort erhalten wurde, versucht sie die empfangenen Daten zu parsen und aktualisiert die Interfaceelemente.

Software Architektur

Die App hat intern eine objektbasierte Architektur, die zentral von der aktiven Activity gelenkt wird. Die Umsetzung einer geschichteten Architektur ist nicht möglich, da es zwei agierende Schichten gibt, zum einen der Nutzer über das Interface, zum anderen die Programmlogik mit ihren Hintergrundaufgaben.

Den aktivsten Teil nimmt die PlayerOverview Activity ein, hier wird sich der Nutzer hauptsächlich aufhalten. Sie beinhaltet die Daten und Anzeigen zum ausgewählten Spieler, den zyklischen Timer für neue Anfragen, speichert und läd Daten auf und von dem Gerät.

Die beiden Fragmente der Overview enthalten Methoden um sich selbst zu aktualisieren, dafür werden ihnen der aktuelle Player und der einzutragende Timestamp von der PlayerOverview übergeben.

Das Player Objekt updated sich per JSONObjekt selbst und enthält eine statische Methode(in Kotlin companion) die aus einem JSONObjekt einen Player generieren kann. Dies wird in der SearchActivity verwendet.

Publish & Subscribe Pattern

Die App verwendet keine Publish & Subscribe Pattern, da immer die zwei Fragmente „Player“ und „Statistic“, sowie das Player Objekt geändert werden müssen. Es ist nicht notwendig eine Registrierungsmöglichkeit für andere Komponenten bereitzustellen.

Frameworks

Verwendete Frameworks

Volley

Im Android Projekt „Geonet“ (erhältlich im Playstore) habe ich das Framework Volley kennengelernt. Mit ihm lassen sich einfach kleine Daten Mengen mit http-Requests anfordern und verarbeiten. Ich habe es wegen seiner sehr kompakten Schreibweise und einfachen Verwendung dem vorgeschlagenen „OKHTTP“ vorgezogen und vor allem, weil ich damit schon gute Erfahrungen gemacht habe und keine Probleme hatte.

Glide

Ich verwende Glide, um die Profilbilder der Nutzer zum einen in der Suche und zum anderen auf der Profilübersicht anzuzeigen. Es vereinfacht dies extrem, da sowieso zu jedem Spieler eine Avatar URL vorhanden ist und ich so mit einer Zeile die URL auslesen und mit einer weiteren Zeile das Profilbild aktualisieren kann.

MPAndroidChart

Anstelle von AChartEngine. MPAndroidChart ist ein wesentlich aktuelleres Framework, als AChartEngine, um Graphen und Diagramme anzuzeigen. Für meine App war es essentiell einen Verlauf von Werten anzuzeigen, weshalb habe ich zwingend eine Graphen Anzeige benötigt. Auch wenn das Styling und Initialisieren etwas umständlich ist, war es doch die beste Lösung für mein Anwendungsfall, die ich finden konnte.

Nicht verwendete Frameworks

Butterknife

Ich verwende Butterknife nicht, da dich die App in Kotlin implementiert habe und dort direkt auf UIObjekte zugegriffen werden kann, nachdem sie importiert wurden. Die Sprache beinhaltet also praktisch alle Vorteile, die dieses Framework mit sich bringt.

Icepick

Icepick ist leider ohne weiteres nicht in Kotlin verwendbar. In der App war es notwendig den Spieler zu speichern, sobald er in der Suche ausgewählt wurde. Immer wenn neue Daten vom Server empfangen werden, speichert die App den Spieler erneut. Beim erneuten Öffnen der App wird der letzte gewählt Spieler geladen.

„Service“

Ich verwende einen fixedRateTimer, aus der Standardbibliothek von Kotlin, dieser läuft auch weiter, wenn das Smartphone gesperrt ist, oder die App im Hintergrund. Dadurch habe ich keine Verwendung für ein Service Framework.

Probleme und Lösungen

Timestamps

Der anspruchsvollste Teil des Projekts war die Verwaltung der Timestamps. Welche Daten müssen vorhanden sein? In welcher Form sind sie am einfachsten und zuverlässigsten verwendbar? Und wie stellt man einen guten zeitlichen Zusammenhang her?

Ich habe mich früh entschieden einen Timestamp pro Tag zu verwenden, um die bereits angesprochenen Ungenauigkeiten und Extremwerte zu eliminieren und die gespeicherten Daten zu minimieren. Deshalb habe ich eine Map anstelle einer Liste verwendet um Dubletten einfach zu ersetzen und trotzdem eine Reihenfolge zu behalten.

Das erste große Problem, das ich hatte war den letzten Eintrag vor einem gewissen Timestamp zur Berechnung von Tageswerten zu ermitteln. Bei den meisten Anfrage befindet sich der letzte Eintrag einen oder wenige Tage davor, die Binäresuche die ich dafür zunächst verwenden wollte wäre also eine eher schlecht Wahl gewesen. Dieses Problem war allerdings mit einer TreeMap einfach zu lösen, da sie eine lowerEntry Methode besitzt, direkt den passenden Eintrag liefert.

Kotlin

Bei diesem Projekt habe ich das erste Mal Kotlin verwendet, was zu Beginn natürlich auch einige Probleme, Verzögerungen und Recherchen mit sich brachte. Mit einer kleinen Einarbeitungszeit war es allerdings auch kein Problem und ich werde Kotlin in Zukunft für alle weitern Apps verwenden, da viele Dinge wesentlich kompakter und schneller umzusetzen sind, als in Java.

ConstraintLayout

Viel Zeit habe ich auch auf das Interface und die Einarbeitung in das ConstrainLayout verwendet. Es bietet einige Tücken und Besonderheiten, die ich erst herausfinden musste. Zum Beispiel das es nicht ausreicht eine Verbindung von A -> B zu ziehen, sondern diese auch noch in A <- B aktiviert werden muss.

Graphen

Das Graphen Framework MPAndroidChart stellte mich auch vor eine Entscheidung. Leider verschiebt sich beim dynamischen Hinzufügen von Knoten mit neuer X-Koordinate das Hintergrundraster. Dies fällt allerdings nur bei der schnellen Generierung von Dummy Daten auf (etwa ab dem 4. Eintrag), da im Normalfall frühesten alle 24 Stunden ein Knoten erzeugt wird fällt dieser Bug nur auf, wenn die App mehr als 72 Stunden aktiv wäre. Es lässt sich durch erneutes Initialisieren des Graphens beheben, darunter leidet allerdings die Leistung. Ich habe mich entschieden die reguläre Verwendung zu optimieren.

Code Format & Lint

Ich habe regelmäßig den Code Lint konform formatiert. Zudem habe ich alle sinnvollen Warnungen behoben. Drei unnötige sind unterdrückt und mit einer Begründung kommentiert.

Fazit

Rocket League Companion ist eine App, die ich in meiner Freizeit weiterverwenden und gegebenen Falls erweitern möchte. Insbesondere in der UI Gestaltung habe ich viele Erfahrungen gewonnen. Das ConstraintLayout hat mir nach etwas Einarbeitungszeit sehr gut gefallen und ich werde es in Zukunft häufig verwenden.

Die vorgeschlagenen Frameworks habe ich wie oben erwähnt nur zum Teil verwendet. AppIntro und Glide nehmen mir viel Arbeit ab, aber sind natürlich nur kleine Teile der App. Volley dagegen wird häufig aufgerufen, auch wenn es lediglich in wenigen Methode vorkommt. Mit Glide und Volley lassen sich viele Webaufgaben einfach und schnell umsetzten. Neu für mich war die Verwendung von http-Headern, die aber auch problemlos funktioniert hat.

Etwas umständlich war die Verwendung von MPAndroidChart. Die Dokumentation ist teilweise unpräzise, unvollständig und könnte mehr Beispiele haben. Allerdings ist es das passendste Framework, das ich gefunden haben, welches meinen Anforderungen entspricht. Die entstandenen Graphen gefallen mir persönlich gut, allerdings geht bei dynamischen Änderungen zum Teil das Hintergrundraster und die Labels kaputt. Dies ist leider ein bekannter Bug und wird vom Autor des Frameworks leider nicht gefixt.

Insgesamt bin ich sehr zufrieden mit dem Arbeitsablauf und dem Resultat. Die Oberfläche und die Funktionen sind so geworden wie ich es mir vorgestellt habe.