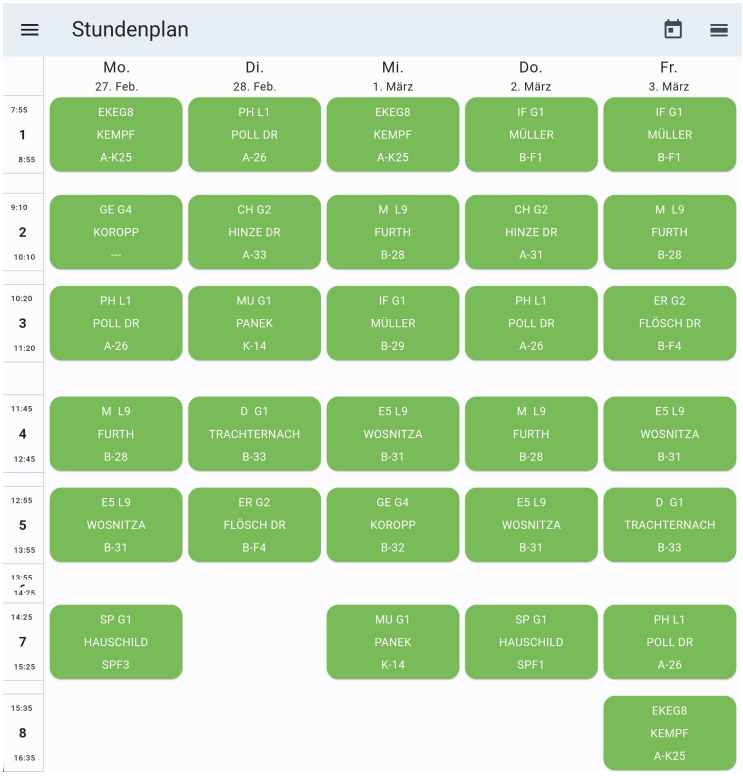
CJD Königswinter Q1

2022/23

Programmierung und Erläuterung einer App zur individuellen Nutzung des Stundenplanprogramms am CJD Königswinter



Verfasser: **Laslo Hauschild**

Kurslehrer: Volker Müller

22. Februar 2023

Inhaltsverzeichnis

1 Einleitung

1.1 Problem

Das CJD Königswinter nutzt die Stunden- und Vertretungsplansoftware *Untis* und die zugehörige App *Untis Mobile*. Die Schule stellt jeder Schülerin und jedem Schüler in einer Klasse/Stufe die gleichen Zugangsdaten zur Verfügung, um auf den Stundenplan zuzugreifen. In diesen sind die entfallenden Stunden integriert. Dies funktioniert in der Unter- und Mittelstufe gut, alle Schülerinnen und Schüler haben schließlich dieselben Stundenpläne. Allerdings ist aufgrund des Kurswahlsystems in der Oberstufe und den dadurch vielen parallelen Kursen dieses System weniger geeignet, weil dadurch die Ansicht der Stunden extrem unübersichtlich wird. Der eigene Stundenplan und die darin entfallenden Stunden sind sehr schwer lesbar. Das ist bei knapp 90 Kursen in einem Jahrgang nicht überraschend. Trotzdem möchte man unbedingt über den neuen Vertretungsplan informiert sein. Das kostet im Moment sehr viel Zeit. Das soll sich mit dieser Facharbeit ändern.

1.2 Lösungsidee

Schülerinnen und Schüler sollen die Möglichkeit bekommen, in einer App die Kurse auszuwählen, die sie gewählt haben. Nur diese werden dann angezeigt. Dafür habe ich eine neue App programmiert, die auf die Daten des Untis-Server zugreifen kann und gleichzeitig eine Option hat, Stunden herauszufiltern. In dieser Facharbeit

1. werden die Anforderungen und Ziele an diese App genannt und die darauf basierende Wahl der Technologie und andere Entscheidungen begründet;
2. wird ein theoretisches Konzept der App entworfen;
3. wird dokumentiert, wie ich Untis und SOL-Connect analysiert habe, um den Zugriff meiner App auf die Daten des Untis-Servers zu ermöglichen und das Ergebnis dargestellt;
4. wird die Implementierung erläutert und
5. werden bei der Implementierung auftretende Probleme und die dazugehörigen Lösungen durch die Wahl bzw. Entwicklung von Algorithmen und Datenstrukturen erklärt.

1.3 Vorhandende alternative Apps

In der Informatik gilt der Grundsatz „Never reinvent the wheel“. In einer Suche nach Alternativen zur Untis Mobile-App auf github.com, der größten Open-Source-Plattform weltweit [14], findet man insbesondere zwei Apps:

- BetterUntis (Android Native, Kotlin) [10]
- SOL-Connect (Flutter, Dart) [13]

Beide unterstützen die in dieser Facharbeit angestrebte Personalisierung nicht. Denkbar ist lediglich, die Apps zur Gewinnung von Erkenntnissen über die Kommunikation zwischen dem Untis-Server und der App heranzuziehen, da beide mit diesem Server kommunizieren können und zusätzlich öffentlich lesbaren Code haben.

2 Theoretisches Konzept

Zunächst muss ein theoretisches Konzept für die Umsetzung der App geplant werden. Dafür werden die Ziele definiert und die Technologie zum Erstellen einer App gewählt.

2.1 Definition der Ziele

Die Ziele lassen sich in zwei Bereiche kategorisieren:

- Anforderungen des Benutzers
- Anforderungen an die Technologie für den Entwicklungsprozess

Für den Benutzer ist es insbesondere wichtig, dass die App den Stundenplan und die entfallenen Stunden korrekt anzeigt. Des Weiteren muss sie, um individuell genutzt zu werden, Stunden benutzerdefiniert herausfiltern können, wenn z. B. der Schüler/die Schülerin diese Kurse nicht gewählt haben. Wichtig ist auch, dass die App sowohl auf Android als auch auf iOS läuft, damit alle Schüler sie nutzen können. Sie sollte außerdem auf alle handelsüblichen Bildschirmgrößen passend dargestellt werden und auf allen gängigen Geräten flüssig laufen.

Die Entwicklung der App sollte möglichst einfach und intuitiv sein. Dazu sollten Fehler direkt bei dem Programmieren zum Beispiel durch eine kompilierte Programmiersprache gefunden werden, um das Risiko für Sicherheitslücken und andere Fehler zu minimieren.

2.2 Wahl der Technologie

Mithilfe dieser Ziele muss vor der Entwicklung der App eine Technologie gewählt werden. Man kann entweder eine *native* App entwickeln oder ein *Framework* nutzen. *Nativ* bedeutet, dass die Apps direkt auf der von Android/iOS bereitgestellten Plattform laufen, während *Frameworks* die vom Betriebssystem bereitgestellte Plattform nutzen und selbst darauf laufen. So bilden sie ihre eigene Plattform. Moderne Apps werden insbesondere mit *Kotlin (Nativ)*, *React (Framework)* und *Flutter (Framework)* erstellt. Davon muss für die Entwicklung der App eine ausgewählt werden.

- Kotlin ist eine Alternative zu Java und ersetzt dieses in der klassischen Art, native Android-Apps zu entwickeln. Deswegen laufen Kotlin-Apps auch nur auf Android

und erfüllt damit nicht die obige Voraussetzung, dass die Apps auch auf iOS und anderen Betriebssystemen laufen sollte. [5]

- React ist ein von Facebook erstelltes Javascript-Framework. Javascript ist keine kompilierte Sprache. Um die Benutzeroberfläche zu erstellen, nutzt es HTML und CSS. Es läuft auf allen Betriebssystemen. Des Weiteren ist React auf Stabilität fokussiert. Das bedeutet, dass es nur sehr selten Updates für React gibt, um das Risiko für Bugs zu minimieren, welche oft von neuen Funktionen verursacht werden. [5]

- Flutter ist ein von Google erstelltes Framework für alle Betriebssysteme. Es nutzt nur die kompilierte Programmiersprache Dart. Dies erleichtert den Entwicklungsprozess enorm, da nur eine Sprache genutzt werden muss, nicht wie bei z. B. React *JavaScript*, *HTML* und *CSS* [8]. Flutter erleichtert durch die Integration von Googles Design System *Material Design* die Entwicklung einer App. Außerdem hat eine mit Flutter geschriebene App nahezu dieselbe Geschwindigkeit wie eine native Android-App. [12]

Kotlin ist Aufgrund der Voraussetzung, dass die App auf allen gängigen Betriebssystemen laufen muss, keine Option. React funktioniert sehr gut, wenn man bereits eine Webseite hat, und die HTML- und CSS-Dateien wiederverwenden kann. Dies ist hier nicht der Fall. Weil Flutter außerdem nur eine kompilierte Programmiersprache benutzt und eine sehr gute Geschwindigkeit aufweist, habe ich mich für Flutter und die dazugehörige Programmiersprache Dart entschieden.

2.2.1 Flutter

Vor der Erläuterung der Entwicklung der App stelle ich hier die wichtigsten Begriffe im Bezug zu dem Framework Flutter vor:

- **Widget** - Ein Widget ist eine Komponente, welche auf dem Bildschirm angezeigt werden kann. Ein Widget kann zum Beispiel ein Stück Text, ein Button oder eine Spalte sein. Weil viele Widgets aus verschiedenen Elementen bestehen, haben viele Widgets sogenannte *children*. Diese *children* sind selbst auch Widgets. Eine Spalte hat zum Beispiel eine Liste von *children*, das sind dann die Widgets sie untereinander anzeigt.

- **State** - Die Daten, welche Widgets benutzen, nennt man State. Ein State könnte zum Beispiel die wichtigsten Informationen über Unterrichtsstunde sein, und ob diese regulär stattfindet. Ein Widget, welches diese Stunde anzeigt, entscheidet dann anhand des States, die Farbe auf Rot zu ändern, wenn die Stunde ausfällt.

- **State-Management-System** - Ein State-Management-System verwaltet und kombiniert State mit den Widgets. Flutter hat sehr viele verschiedene State-Management-Systeme [16]. Ein State-Management-System muss sicherstellen, dass alle Widgets Zugriff auf den State haben und bei einer Veränderung des States alle Widgets erneuern, damit keine veralteten Informationen angezeigt werden. Es gibt keine eindeutige Wahl, wie man dies umsetzt. Flutter empfiehlt aber u. a. *Riverpod* [16].

2.2.2 Riverpod

Um die App zu entwickeln, muss also auch ein State-Management-System gewählt werden. Hier habe ich mich aufgrund der oben genannten Empfehlung für *Riverpod* entschieden.

Riverpod [18] ist ein State-Management-System, welches auf *Providern* basiert.

> „ A provider is an object that encapsulates a piece of state and allows listening to that state. [17] “
>
> Übersetzung des Autors: Ein Provider ist ein Objekt, das einen Teil des Zustands einkapselt und den Zugriff auf diesen Zustand ermöglicht.

Provider können überall in der App erstellt werden. Jedes Widget bekommt dann die Möglichkeit, diesem Provider „zuzuhören“. Das bedeutet, dass Widgets, welche diesen Daten „zuhören“, bei Veränderung der Daten automatisch erneuert werden. Des Weiteren kann jedes Widget den Wert im Provider mit dessen Methoden verändern.

So muss ich mich so nicht mehr um die Aktualisierung der Widgets kümmern, dies geschieht vollautomatisch und ist so auch schneller, da Riverpod Widgets nur neu rendert, wenn sich die spezifisch abgefragten Daten auch geändert haben. Dies macht die Programmierung der App einfacher, was ein Ziel dieser App ist.

Die Provider, die ich benutzt habe, sind die sogenannten *StateNotifierProvider* und haben zwei Teile: Einen *Notifier* und einen *State*. Der State sind die Daten selbst, der Notifier hat Methoden, welche dazu genutzt werden können, den State zu verändern. Dies könnte zum Beispiel die Methode *logout* im *UserSessionProvider* sein. Diese Methoden legen dann einen neuen State fest, woraufhin alle Widgets, die von diesen Daten abhängen, neu gerendert werden. Während der Notifier immer am Namen zu erkennen ist, ist der State eine einfache Klasse, welche die Daten speichert. Bei dem eben genannten *UserSessionProvider* ist das zum Beispiel *UserSession*.

2.2.3 Projektstruktur

Zuletzt muss noch die Ordnerstruktur des Projekts gewählt werden, um die Übersichtlichkeit des Quellcodes zu gewährleisten.

Der gesamte Quellcode liegt, wie von Flutter vorgegeben, im Ordner *lib*. Dort ist auch die Hauptdatei *main.dart* zu finden.

Die App muss zunächst die Stundenplandaten von Untis-Server laden. Dann muss sie die Kurse des Benutzers abfragen und speichern. Damit der Code übersichtlich bleibt, sollen alle Dateien der Benutzeroberfläche in einem Ordner gespeichert sein. Die Provider, welche Daten speichern und ggf. vom Server abfragen, sollen ebenfalls in ihrem eigenen Ordner liegen. Deswegen sind in *lib* folgende Ordner enthalten:

- *core* - Hier in */core/api/providers* liegen alle Provider, die für die Kommunikation zwischen der App und dem Untis-Server zuständig sind. Diese Provider haben Methoden, um den Benutzer anzumelden, die Daten zu laden und zu verarbeiten. Die Daten werden durch Klassen bzw. ihre Instanzen gespeichert. Diese liegen in */core/api/models*.
- *filter* - Hier liegt der Provider, der die Kurse speichert, welche angezeigt werden sollen, um die individuelle Einstellung zu ermöglichen.
- *settings* - Damit der Benutzer einige Einstellungen wie z. B. die Veränderung der Helligkeit vornehmen kann, werden diese Daten in Providern in diesem Ordner gespeichert.
- *ui* - kurz für *User Interface* beinhaltet alle Widgets, welche dem Benutzer angezeigt werden.
- *util* - Code, der oft benötigt wird, aber nicht eindeutig zugeordnet werden kann, liegt hier.

Hiermit ist die Wahl der Technologie und die grobe Planung abgeschlossen.

3 Lösungsfindung

3.1 Methoden zum Reverse-Engineering

Bevor die App jedoch implementiert werden kann, muss der Zugriff auf die Daten auf dem Untis-Server ermöglicht werden. Die App muss nämlich Daten vom Untis-Server abfragen, damit die Stunden korrekt angezeigt werden können. Dafür muss ich zunächst herausfinden, wie Untis Mobile funktioniert, um dieses Verhalten dann zu imitieren.

Dafür muss man Untis Mobile *reverse-engineeren*. Dies ist der Prozess, eine App zu analysieren, um Informationen über die Funktionsweise zu bekommen. Man kann die Kommunikation zwischen App und Server analysieren und man kann den Quellcode anschauen. Bei Open-Source-Apps ist dafür kein Reverse-Engineering nötig, weil Untis Mobile allerdings keine Open-Source-Software ist, muss man die *apk* (die Datei, in der der Code in kompilierter Form vorliegt und auf dem Handy ausgeführt wird) dekompilieren.

3.1.1 Http-Traffic

Um die Kommunikation zwischen App und Server zu analysieren, hat man verschiedene Möglichkeiten. Zunächst kann man die Kommunikation zwischen App und Server abfangen. Dies funktioniert aufgrund der heutzutage standardmäßig verwendeten SSL-Verschlüsslung nicht mehr zwingend, da so die Daten meist unlesbar sind. Die zweite Option ist deswegen, eine verschlüsselte Verbindung nicht zwischen Untis-Server und der App, sondern zwischen mir und der App aufzubauen. Dafür muss ich behaupten, ich sei webuntis.com. Wahrscheinlich wird die App dann die Verbindung abbrechen, weil für diese Behauptung ein Zertifikat nötig ist, welches von einer *Certificate Authority*, kurz *CA* ausgestellt wird. Dieses Zertifikat wird nur dann ausgestellt, wenn mir webuntis.com tatsächlich gehört. Das ist offensichtlich nicht der Fall.

> „Every TLS client has a list of root CAs that it trusts, and to successfully receive an HTTPS request, you must be able to present a certificate for the target hostname that includes a trusted root CA somewhere in its chain.“ [4]
>
> Übersetzung des Autors: Jeder TLS-Client hat eine Liste von Stammzertifizierungsstellen, denen er vertraut. Um eine HTTPS-Anfrage erfolgreich zu empfangen, müssen Sie ein Zertifikat für den Ziel-Hostnamen vorlegen

können, dass eine vertrauenswürdige Stammzertifizierungsstelle irgendwo in seiner Kette enthält.

Jetzt stellt sich die Frage, wie man an ein vertrauenswürdiges CA-Zertifikat herankommt. Dies kann erreicht werden, indem man eine *CA* erfindet und sich selbst das benötigte Zertifikat ausstellt. Wenn Untis Mobile nicht speziell dagegen abgesichert ist, benutzt sie Android-Systemzertifikate, um zu prüfen, ob eine *CA* vertrauenswürdig ist. Diese Systemzertifikate lassen sich bei Android-Emulatoren so bearbeiten, dass meine *Certificate Authority* als vertrauenswürdig angesehen wird. So kann dann der Datenverkehr überwacht werden.

Die Umsetzung dieses theoretischen Konzepts ist einfach - das Programm *HTTP Toolkit*, kombiniert mit über einen durch Android Studio/IntelliJ Idea installierten *Android Emulator* übernimmt vollautomatisch die Injektion der Zertifikate sowie das Abfangen und Weiterleiten der Anfragen.

Zunächst habe ich einen Android-Emulator mit Android Studio installiert, und die Untis Mobile Apk von ApkPure [9] heruntergeladen und per Drag-and-Drop auf dem Emulator installiert. Dann habe ich Http-Toolkit installiert und das Abfangen der Requests gestartet, wie es in Abbildung 1 zu sehen ist. Der Prozess wird auch in dem Tutorial von Http Toolkit beschrieben [3].

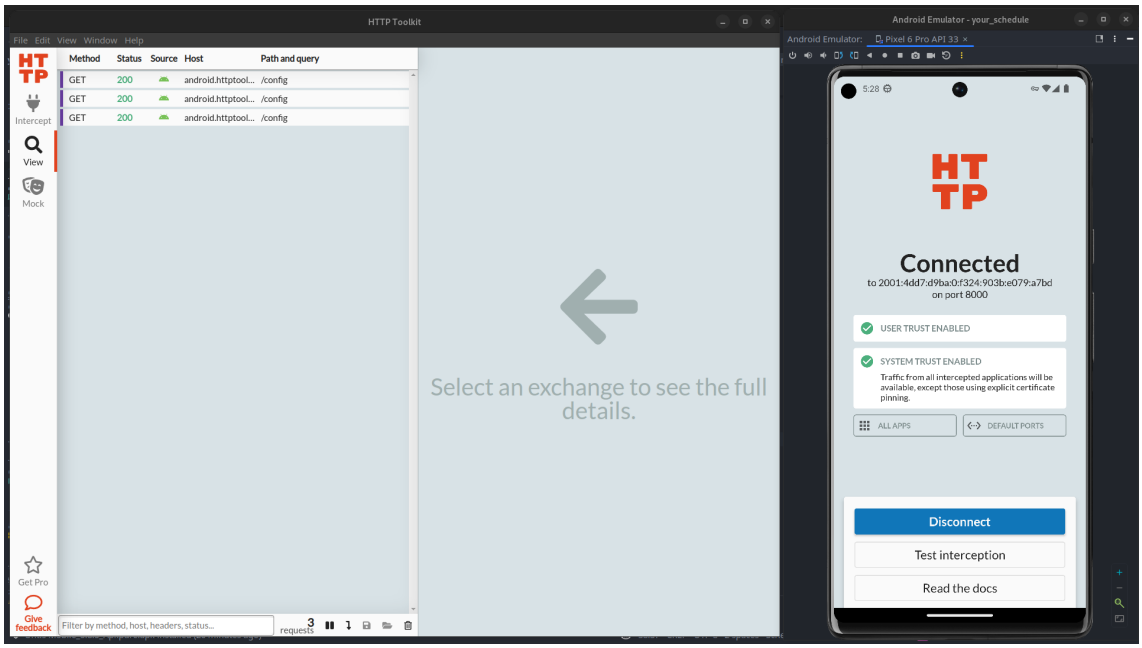


Abbildung 1

So kann später unter anderem herausgefunden werden, wie Untis Mobile mit dem Server kommuniziert und so die Abfrage der Daten des Untis-Servers in meiner App ermöglicht werden.

3.1.2 Quellcode

Um herauszufinden, wie eine App funktioniert, kann man den Quellcode analysieren. Untis Mobile ist nicht Open-Source, das heißt, der Code ist nicht öffentlich. Allerdings kann man die Apk-Datei, in der der Code in kompilierter Form auf dem Handy vorliegt, wie oben beschrieben herunterladen und diese dann in Javacode umwandeln [6]. Dies hat einen entscheidenden Nachteil: Ein dekompilierter Quellcode ist oft schwer zu lesen, nicht zuletzt, da bei dem Kompilieren wichtige Teile aus dem Programm, die zwar keinen Nutzen für den Computer, sehr wohl aber für die Verständlichkeit für den Menschen entfernt werden: Die Kommentare. Des Weiteren sind die kompilierten Dateien oft so verändert, dass es sehr schwierig wird, den Code zu verstehen.

3.2 Dokumentation der Durchführung

3.2.1 Untis Mobile

Mit den beschriebenen Methoden kann ich analysieren, wie Untis Mobile und die zugehörigen Server kommunizieren. Nach dem Login sendet die App ungefähr 20 Requests hauptsächlich an *herakles.webuntis.com*. Abbildung 2 zeigt hauptsächlich zwei Arten von URLs in den Requests:

- `/jsonrpc_intern.do?school=cjd-k\%C3\%B6nigswinter&...`
- `/WebUntis/api/rest/view/v1/...`

POST	200	schoolsearch.webuntis.com	/schoolquery2?v=a5.8.5
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	push.webuntis.com	/api/register/
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
GET	200	herakles.webuntis.com	/WebUntis/api/rest/view/v1/dashboard/cards/status?school=cjd-...
GET	200	herakles.webuntis.com	/WebUntis/api/rest/view/v1/messages/status?school=cjd-k%C3...
GET	200	herakles.webuntis.com	/WebUntis/api/rest/view/v1/trigger/startup?school=cjd-k%C3%...
GET	200	herakles.webuntis.com	/WebUntis/api/rest/view/v1/dashboard/cards/status?school=cjd-...
GET	200	herakles.webuntis.com	/WebUntis/api/rest/view/v1/messages/status?school=cjd-k%C3...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...
POST	200	herakles.webuntis.com	/WebUntis/jsonrpc_intern.do?school=cjd-k%C3%B6nigswinter&...

Abbildung 2

3.2.2 SOL-Connect

Des Weiteren kann man sich Open-Source-Alternativen zu Untis Mobile anschauen, die auch genutzt werden können, um sich bei Untis einzuloggen, wie zum Beispiel die oben genannte App *SOL-Connect* [13]. Hier ist der Quellcode öffentlich verfügbar und kann so weiter genutzt werden. Nach dem Installieren der App auf einem Smartphone klappt der Login zunächst nicht, stattdessen wird eine Fehlermeldung angezeigt. Eine Analyse der Requests mit dem oben beschriebenen Prozess zeigt, die Anfragen gehen an *hepta.webuntis.com*. Untis Mobile hat aber alle Anfragen an *herakles.webuntis.com* geschickt. Diese URL ist für verschiedene Schulen unterschiedlich und hier falsch.

Um dieses Problem zu lösen, habe ich mir den (für diese App öffentlichen) Quellcode [13] heruntergeladen. Es fällt auf, dass die globale Variable *apiBaseUrl* fest auf *hepta.webuntis.com* gesetzt ist [7]. Durch die Änderung der Variable funktioniert die App. Ich habe die Autoren der App über den Bug informiert. Diese haben angekündigt, dies in einer zukünftigen Version zu beheben. Dann habe ich die Requests der funktionierenden App auf dieselbe Art abfangen. Die SOL-Connect App kommt mit deutlich weniger Requests aus - Es reichen gerade einmal 5 Anfragen, um den Stundenplan anzuzeigen, wie in Abbildung 3 ablesbar.

Aufgrund dessen und weil der Source-Code öffentlich ist, habe ich mich entschieden, SOL-Connect genau zu analysieren und meine App mithilfe der Ergebnisse zu erstellen.

POST	200	?	herakles.webuntis.com	/WebUntis/jsonrpc.do?school=cjd-k%C3%B6nigswinter
GET	200	?	herakles.webuntis.com	/WebUntis/api/token/new
GET	200	?	herakles.webuntis.com	/WebUntis/api/rest/view/v1/app/data
GET	200	?	herakles.webuntis.com	/WebUntis/api/rest/view/v1/timegrid
POST	200	?	herakles.webuntis.com	/WebUntis/jsonrpc.do?school=cjd-k%C3%B6nigswinter

Abbildung 3

3.3 Ergebnis

Aus *Http-Toolkit* geht das Request-Schema (Abbildung 4) hervor. Es zeigt, welche Anfragen nötig sind, um die Stundenplandaten vom Server zu erhalten.

Abbildung 4

Den Code, der die Authentifizierung und Datenverarbeitung durchführt, zu kopieren, ist aber keine Option. Die SOL-Connect-App ist für die Integration der sogenannten SOL-Phasierung gemacht [13]. Meine App hat mit der SOL-Phasierung nichts zu tun und soll indviduelle Stunden filtern. Auch wenn SOL-Connect auch Flutter und Riverpod benutzt, entspricht der Codestyle nicht der Empfehlung der Ersteller von Riverpod, da sie *ChangeNotifierProvider* benutzen.

> „Using ChangeNotifierProvider is discouraged by Riverpod (...)“ [11]
>
> Übersetzung des Autors: Die Verwendung von ChangeNotifierProvider wird von Riverpod nicht empfohlen (...)

Das Ergebnis der Analyse ist, dass der Code als Inspiration gut geeignet ist, aber nicht als Grundlage für meine App. Jetzt kann die App implementiert werden.

4 Implementierung

Grundsätzlich muss die App folgendes machen:

1. Zunächst muss der Benutzer authentifiziert werden. Dafür gibt er den Benutzernamen, das Passwort, die Schule sowie die Domain ein, an die die Anfragen geschickt werden sollen. Für das CJD ist das, wie oben beschrieben *herakles.webuntis.com*. Andere Schulen könnten andere Domains benutzen. Dies muss auch auf dem Handy gespeichert werden, damit er diese Daten zukünftig nicht mehr eingeben muss. Jetzt kann nach dem oben beschriebenen Request-Schema die Authentifizierung beginnen.

2. Daraufhin muss der Stundenplan mit den folgenden Requests geladen werden.

3. Dann kann der Benutzer aus einer Liste alle Kurse auswählen, die er gewählt hat. Dies muss ebenfalls gespeichert werden.

4. Jetzt kann der Stundenplan angezeigt werden. In dieser Anzeige sollte auch die Option bestehen, nur einen Tag oder die ganze Woche angezeigt zu bekommen.

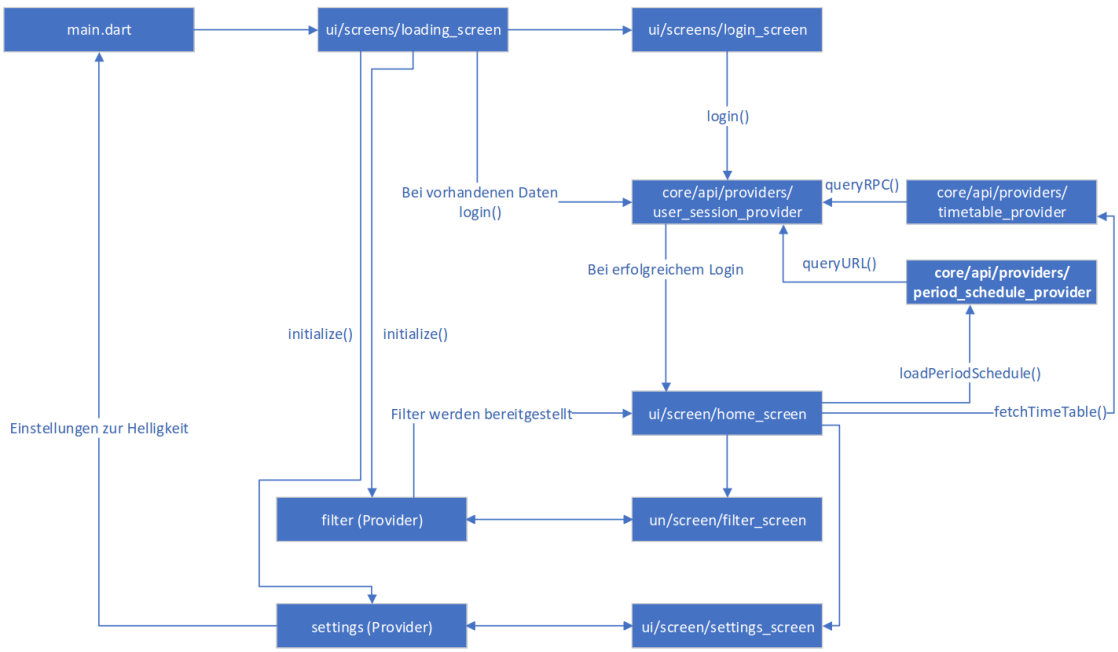


Abbildung 5

Die Implementierung hiervon ist als UML-Diagramm in Abbildung 5 dargestellt. Es stellt den Ablauf des Codes dar, ausgehend von main.dart. Die Einzelheiten sind im Code mit Kommentaren erläutert, wenn Teile unklar sind.

Hier werde ich Probleme und Entscheidungen darstellen.

4.1 Datenstrukturen

4.1.1 Datenstruktur des Stundenplans

Die Stunden selbst müssen in einer Datenstruktur gespeichert werden. Bei der Wahl der Datenstruktur spielt insbesondere eine Rolle, in welcher Form die Daten benötigt werden. Diese Operation muss dann möglichst einfach ohne weitere Algorithmen umsetzbar sein.

Die Schulstunden werden von der Wochen- und Tagesansicht benötigt. Deswegen sollten die Stunden einer Woche und eines Tages möglichst in einem Objekt zusammengefasst sein. Also speichert der *TimeTableProvider* Instanzen von *TimeTableWeek*. Diese speichern dann wieder Instanzen von *TimeTableDay*. Die *TimeTablePeriod*-Objekte sind dann in *TimeTableDay* gespeichert.

Dann bleibt noch die Frage, in welcher Datenstruktur alle diese Objekte die untergeordneten Objekte speichern soll. Am häufigsten wird der Benutzer die Woche, die angezeigt werden soll, definieren. Dann ist also eine Datenstruktur ideal, in der möglichst schnell das zu dieser Woche zugehörige *TimeTableWeek*-Objekt gefunden werden kann.

Deswegen habe ich mich für eine *Map* entschieden. Sie ordnet jeder Woche eine *TimeTableWeek*-Instanz zu und erlaubt direkten Zugriff auf eine *TimeTableWeek*-Instanz, wenn man die dazugehörige Woche nennt. Wenn jetzt also der Stundenplan einer bestimmten Woche angezeigt werden soll, können die hierfür benötigten Daten direkt aus der Map mit einer Laufzeit von O(1) angefragt werden. Auch wenn dies zeitlich aufgrund der geringen Datengröße keinen Einfluss haben dürfte, wird der Code dadurch intuitiver, weil Maps aufgrund deren Laufzeit für solche Anwendungsfälle optimiert sind [2].

Die *TimeTableWeek*-Instanzen selbst speichern die *TimeTableDay*-Instanzen aus denselben Gründen in einer Map, zugeordnet zu dem jeweiligen Tag.

TimeTableDay kann allerdings keine Map benutzen. Dies hat mehrere Gründe: Zum einen gibt es keine eindeutigen Kategorien mehr, in die man die *TimeTablePeriod*-Instanzen sortieren könnte. Diese haben nämlich sowohl eine Start- und eine Endzeit, wodurch Überschneidungen möglich sind. Damit ist es nicht mehr klar, wie man die *TimeTablePeriod*-Instanzen zuordnen sollte. Zum anderen gibt es keinen Fall, wo man nur ein paar bestimmte *TimeTablePeriod*-Instanzen in der App braucht. Die Anzeige der Stunden braucht immer alle Stunden. Damit ist es am sinnvollsten, sie in einer Liste zu speichern. Weil jedoch die Anfangszeiten oft eine Rolle spielen, liegt es nahe, diese nach Startzeit zu sortieren. *TimeTableDay* speichert die *TimeTablePeriod*-Instanzen also in einer nach

Startzeit sortierten Liste.

4.1.2 Datenstruktur der Filter

Die herauszufilternden Stunden im Stundenplan werden in *filter/filter.dart* verwaltet. Sie werden als eine Art Blacklist gespeichert. Das bedeutet, Kurse, die der Benutzer herausfiltern möchte hat, werden gespeichert, während andere Kurse, welche dem Benutzer angezeigt werden sollen nicht gespeichert werden. Um dann herauszufinden, ob ein Kurs angezeigt werden soll, muss lediglich geguckt werden, ob dieser Kurs in der Blacklist erhalten ist. Wenn dies der Fall ist, dann wird der Kurs nicht angezeigt. Damit ist die wichtigste Operation *contains*. Das Set hat den entscheidenden Vorteil, dass die Zeitkomplexität dieser Operation O(1) ist [15]. Des Weiteren spielt die Reihenfolge keine Rolle, dies ist beim Set auch gegeben.

4.2 Algorithmus zur Stundenanzeige

Die letzte Herausforderung der App ist die Stundenanzeige selbst. Die Anordnung von zeitlich parallelen Stunden ist nicht trivial, wenn diese unterschiedliche Start- und Endzeiten haben, z. B. bei mehrstündigen Klausuren oder außerplanmäßigen Aktivitäten. Die Stunden sollen dann in der Breite möglichst gleichmäßig verteilt sein, ohne dabei Lücken zu lassen. Ein Algorithmus zur Lösung dieses Problems ist hier [1] beschrieben. Diese ist allerdings für ein anderes Framework, die Lösungsidee habe ich benutzt und meinen eigenen Algorithmus entworfen. Er funktioniert wie folgt:

1. Das Layout ist ein unbegrenztes Raster.
2. Die Stunden können in Stundenblöcke unterteilt werden. Diese Blöcke starten immer genau dann, wenn die Startzeit der nächsten Stunde nach der Endzeit der vorherigen Stunde liegt. Dann sind sie in der Anzeige völlig unabhängig.
3. Jede Stunde ist eine Spalte breit, die vertikale Position ist durch die Start- und Endzeiten bestimmt.
4. Platziere jede Stunde so weit links wie möglich, ohne dabei eine andere Stunde zu überschneiden.
5. Wenn alle Stunden platziert sind, berechne für jeden Stundenblock die tatsächliche Breite der Stunden, indem man den Platz durch die Anzahl der Spalten teilt.
6. Fülle die leeren Lücken im Raster auf, indem die anliegenden Stunden nach rechts erweitert werden. [1]

Die horizontale Position wird durch die Anzahl der Spalten $\#_{columns}$ definiert. Sie haben in einem Stundenblock folgende Breite:

$$horizontalPos(\#_{columns}) = width \div \#_{columns}$$

So haben alle Spalten dieselbe Breite. Jetzt können die Position und Größe der Spalten ausgerechnet werden. Die Stunden sind nach dem obigen Algorithmus Spalten zuzuordnen und haben so ihre Größe.

Die vertikale Position wird durch den Startzeitpunkt gegeben. Um einen Zeitpunkt z in eine Position auf dem Bildschirm umzurechnen, nutze ich folgende Formel:

$$verticalPos(z) = height \times \frac{z - start_{day}}{end_{day} - start_{day}}$$

Den Zähler des Bruchs kann man sich als den Zeitpunkt vorstellen, der z wäre, wenn der Schultag um 0 Uhr beginnen würde. Dies ist nötig, da ich die Zeit vor der ersten Stunde nicht anzeige und somit nicht zur Höhe gehört. Dies teile ich durch die Länge des Schultages, d. h. $end_{day} - start_{day}$. Also gibt der Bruch z im Verhältnis zur Länge des Schultages an. Weil $height$ die gesamte Länge des Tages einnimmt, ist das Produkt der Höhe und dem Bruch die Position, wo z platziert sein muss.

5 Fazit

Im Rahmen dieser Facharbeit habe ich eine App mit über 3700 Zeilen Code entwickelt und veröffentlicht, welche den Abruf des digitalen Stunden- und Vertretungsplans des CJD Königswinter einfacher und benutzerfreundlicher macht. Dies ist für die Schülerinnen und Schüler der Oberstufe von großem Nutzen.

Die App wurde mithilfe der Analyse der App *Untis Mobile* erstellt und nutzt weiterhin das Stundenplanprogramm *Untis*. Dies ist sehr vorteilhaft, weil so alle Schüler von der Verbesserung profitieren können, ohne dass die Schule Veränderungen anstoßen muss. Andererseits ist meine App abhängig von Untis, welche jederzeit die Funktionsweise der Server verändern können und meine App dann nicht mehr funktionieren würde. Offen in der Implementierung ist noch eine Speicherung der Stundenplandaten, sodass der Benutzer auch ohne Internet auf die letzten geladenen Daten zugreifen kann. Der Code dieser App ist Open-Source, damit jeder den Code, den ich hier geschrieben habe, weiterverwenden darf.

Das Hauptziel der Facharbeit war es, eine App zu erstellen, die jedem erlaubt, seinen Stundenplan individuell auf dem Smartphone anzuzeigen und so auch entfallende Stunden schneller zu sehen. Jetzt läuft die App auf Android und iOS und ist im Google Playstore mit der Suche „Stundenplan Laslo Hauschild“ zu finden. Im AppStore landet die App voraussichtlich erstmal nicht, weil die Kosten von 100 USD pro Jahr für mich den Nutzen deutlich übersteigen. Im Play Store können sich künftig alle Schülerinnen und Schüler des CJD Königswinter die App einfach herunterladen, sich anmelden und ihre Kurse auswählen. Damit muss keiner mehr seine Stunden in der Untis Mobile-App suchen. Im Moment testen bereits 15 Personen diese App. Das Ergebnis dieser Facharbeit ist also ein Erfolg.

6 Literaturverzeichnis

[1] Markus Jarderot. *Visualization of calendar events. Algorithm to layout events with maximum width*. 2012. URL: https://stackoverflow.com/a/11323909/16052812.

[2] Ronan McClorey. *Hash Maps to Improve Time Complexity*. 2021. URL: https://javascript.plainenglish.io/hash-maps-to-improve-time-complexity-bc7ca692fcc.

[3] Tim Perry. *Intercept and edit HTTP traffic from (almost) any Android app*. 2021. URL: https://httptoolkit.com/blog/inspect-any-android-apps-http/.

[4] Tim Perry. *Intercepting HTTPS on Anroid*. 2021. URL: https://httptoolkit.com/blog/intercepting-android-https/.

[5] Scarlett Rose. *Flutter vs. React Native vs. Kotlin: Which One is Driving 2021 and Why?* 2021. URL: https://code.likeagirl.io/flutter-vs-react-native-vs-kotlin-which-one-is-driving-2021-and-why-4c3b38efa912.

[6] Tanuj Soni. *Reverse Engineer Your Favorite Android App*. 2021. URL: https://medium.com/helpshift-engineering/reverse-engineer-your-favorite-android-app-863a797042a6.

[7] Laslo Hauschild. *Changed Domain*. 2022. URL: https://github.com/floodoo/SOL-Connect/commit/ef6fbf653007672b641bdfb33bd71c444a7cd9c8.

[8] Obumuneme Nwabude. *Why you should use Flutter for Your Projects*. 2022. URL: https://www.freecodecamp.org/news/why-you-should-use-flutter/.

[9] URL: https://apkpure.com/de/untis-mobile/com.grupet.web.app/download.

[10] *BetterUntis. An alternative mobile client for the Untis timetable system.* URL: https://github.com/SapuSeven/BetterUntis.

[11] *ChangeNotifierProvider*. URL: https://riverpod.dev/docs/providers/change_notifier_provider.

[12] *Flutter*. URL: https://flutter.dev.

[13] Florian Gersch Philipp und Schmitz. *SOL-Connect*. URL: https://github.com/floodoo/SOL-Connect.

[14] *Github*. URL: https://en.wikipedia.org/wiki/GitHub.

[15] *HashSet E class*. URL: https://api.dart.dev/be/175791/dart-collection/HashSet-class.html.

[16] *List of state management approaches*. URL: https://docs.flutter.dev/development/data-and-backend/state-mgmt/options.

[17] *Providers*. URL: https://riverpod.dev/docs/concepts/providers.

[18] Remi Roussele. *Riverpod. A Reactive Caching and Data-binding Framework*. URL: https://riverpod.dev/.

7 Anhang

Die benutzen Webseiten sowie der Quellcode der App sind aus ökologischen Gründen auf **laslo-hauschild.eu/facharbeit** zu finden. Für Zugangsdaten zum Testen der App kontaktieren sie mich gerne unter lhauschild@cjd-koenigswinter.net.

8 Selbstständigkeitserkärung

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Ort, Datum	Laslo Hauschild