



Compose
Multiplatform

2022

Bachelor-Thesis: Compose BigLazyTable



Livio Näf, Marco Sprenger
FHNW Hochschule für Technik
25.03.2022

Compose BigLazyTable

**Hochwertige Tabelle für grosse Datenmengen
mit Compose for Desktop**

Bachelor-Thesis

Näf Livio, Semester 7

livio.naef@students.fhnw.ch

Sprenger Marco, Semester 7

marco.sprenger@students.fhnw.ch

FHNW

Hochschule für Technik

Informatik

Betreuer: Dr. Dieter Holz

Windisch, 25. März 2022

Abstract

Viele Desktop Geschäfts-Applikationen heutzutage bestehen aus einer Tabelle, welche alle Datensätze zeigt und einem dazu geeigneten Formular für die Bearbeitung der Daten. Diese Applikationen müssen entsprechende Funktionalitäten für den End-Benutzer bieten, damit dieser effizient und angenehm damit arbeiten kann.

Für Entwickler gibt es in älteren Programmiersprachen, wie Java mit *JavaFX* entsprechende Bausteine, um solche Applikationen zu bauen. Meist sind diese Applikationen speziell auf eine Firma angepasst und kommen in einem Design daher, welches nicht mehr so zeitgemäss ist. Zudem muss bei einer *JavaFX* Lösung das *Lazy Loading*, sowie eine Verbindung mit einem Formular, trotzdem selbst programmiert werden. Mit *Compose BigLazyTable* wurde mit Kotlin und *Compose for Desktop* eine Library entwickelt, mit welcher Entwickler unkompliziert eine Basis-Applikation erstellen können, bestehend aus einer Tabelle und einem Formular für die Bearbeitung der Daten. *Compose for Desktop* verwendet für die UI Bausteine standardmässig Material Design, was ein deutlich moderneres Aussehen bietet. Die Tabelle ist mit Filtern, Sortiermöglichkeiten und einem *Lazy Loading* Mechanismus ausgestattet. Kombiniert mit einem Formular der Library *Compose Forms*, bietet *Compose BigLazyTable* die notwendigen Funktionalitäten, welche man von *JavaFX* gewohnt ist.

Als Ziel der FHNW (Fachhochschule Nordwestschweiz) war es zu analysieren, wie weit die Entwicklung von *Compose for Desktop* schon vorangeschritten ist und ob in Zukunft im Unterricht anstelle von Java und *JavaFX*, neu Technologien wie Kotlin und *Compose for Desktop* eingesetzt werden können.

Vorwort

An dieser Stelle möchten wir uns bei all denjenigen bedanken, die uns während der Anfertigung dieser Projektarbeit unterstützt und motiviert haben.

Zuerst geht ein Dank an Dr. Dieter Holz, welcher unsere Projektarbeit betreut hat. Für die konstruktive Kritik und die hilfreichen Anregungen bei der Erstellung dieser Arbeit möchten wir uns herzlich bedanken.

Zudem möchten wir uns bei Prof. Dierk König bedanken, welcher in unseren Meetings immer hilfreiche Inputs geliefert hat.

Inhaltsverzeichnis

ABSTRACT	3
VORWORT	3
1 EINLEITUNG.....	6
1.1 AUSGANGSLAGE.....	6
1.2 ZIELE DES PROJEKTS	7
1.3 RESULTAT	8
1.4 STRUKTUR DES BERICHTS	9
1.5 PROJEKT-LINKS.....	9
2 TECHNISCHE EVALUATION	10
2.1 KOTLIN	10
2.1.1 Vergleich Kotlin und Java	10
2.1.2 Kotlin Coroutines.....	11
2.1.3 Kotlin Eigenheiten	13
2.2 COMPOSE	15
2.2.1 Compose Eigenheiten	15
2.2.2 Compose Multiplatform	17
2.3 PAGING MECHANISMUS	18
3 METHODIK	19
3.1 KANBAN.....	19
3.1.1 Grundlagen.....	19
3.1.2 Umsetzung.....	21
3.1.3 Fazit	22
3.2 CODEVERWALTUNG	22
3.3 SONARQUBE	23
3.4 SONARLINT	24
3.5 AUFARBEITUNG DER DATEN	25
4 LÖSUNGSaufbau	27
4.1 SOFTWARE-ARCHITEKTUR.....	27
4.1.1 Data	28
4.1.2 Model.....	36
4.1.3 Controller	39
4.1.4 View	44
4.2 LAUFZEIT-MESSUNGEN.....	51
4.3 SOFTWARE-TESTING	52
4.3.1 Logging.....	52
4.3.2 Tests	54
4.4 GENERALISIERUNG.....	63
4.5 MÖGLICHE ERWEITERUNGEN	66
5 SCHLUSSFOLGERUNG	68
6 REFLEXION	69

7	LITERATURVERZEICHNIS	70
8	ABBILDUNGSVERZEICHNIS.....	71
9	TABELLENVERZEICHNIS	73
10	ANHANG.....	74
10.1	GLOSSAR.....	74
10.2	FILTER VERSIONEN	75
10.3	SKIZZE GRUNDAUFBAU COMPOSE BIGLAZYTABLE	79
11	EHRlichkeitSERKLÄRUNG	80

1 Einleitung

1.1 Ausgangslage

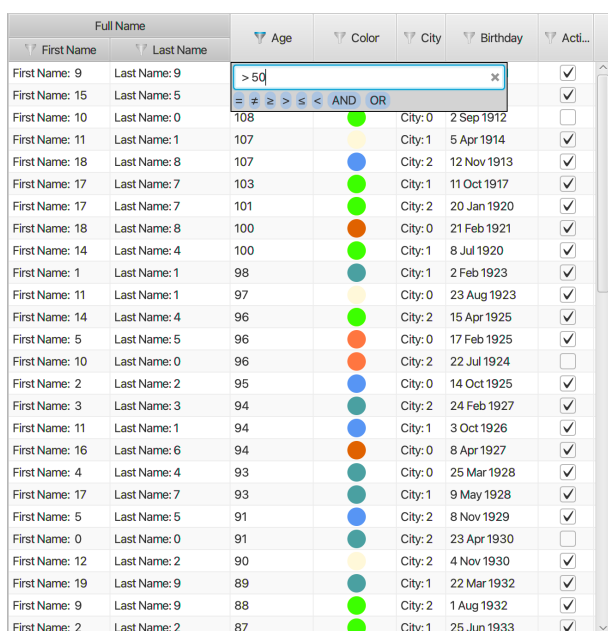
Jetpack Compose ist ein modernes UI-Toolkit von Google für die Entwicklung von nativen Android Apps. Die Firma JetBrains hat diesen Ansatz weiterentwickelt und verfolgt mit *Compose for Desktop* und *Compose for Web* das Ziel, mit dem gleichen deklarativen Ansatz native Desktop- und Web-Applikationen in Kotlin zu entwickeln. Compose verspricht, die Entwicklung gegenüber bestehenden Technologien zu vereinfachen und zu beschleunigen. *Compose for Desktop* wurde Ende 2020 als Alpha-Release veröffentlicht. Zum Zeitpunkt dieser Projektarbeit ist bereits der erste Stable-Release mit der Version 1.1.0 verfügbar. Compose beschränkt sich auf die UI-Teile einer Applikation. Fragestellungen, welche die Bereitstellung von sehr vielen Daten betreffen, werden nicht oder nur im Ansatz berücksichtigt. [19]

Neben *Compose for Desktop* gibt es noch 2 weitere Ausprägungen von Compose. *Jetpack Compose* zur Entwicklung nativer Android-Apps, welches bereits in der Industrie eingesetzt und stark von Google gepusht wird. Mit *Compose for Web* wird noch das fehlende Stück der *Compose* Reihe abgedeckt. Dies ist aber erst als Technology Preview Version veröffentlicht. *Compose* verspricht einen hohen UI Code-Anteil zwischen einer Android-, einer Desktop- und einer Web-Applikation teilen zu können.

Eine geeignete Möglichkeit zur Bearbeitung von Daten bildet die *Compose Forms Library* eines Vorgängerprojekts, die einfach angebunden und eingesetzt werden kann. [16]

JavaFX bietet bereits ein Baustein für eine TableView mit verschiedenen Funktionalitäten. Jedoch muss der Entwickler ein *LazyLoading* selbst implementieren und es gibt keine Standard-Library für eine Verbindung des Tables und einem Formular.

Auftraggeber und Betreuer des Projekts ist Dr. Dieter Holz, Dozent an der FHNW und erfahrener Entwickler im Bereich Java, sowie der *Compose*-Library.



Full Name		Age	Color	City	Birthday	Acti...
First Name	Last Name					
First Name: 9	Last Name: 9					✓
First Name: 15	Last Name: 5					✓
First Name: 10	Last Name: 0	108		City: 0	2 Sep 1912	
First Name: 11	Last Name: 1	107		City: 1	5 Apr 1914	✓
First Name: 18	Last Name: 8	107		City: 2	12 Nov 1913	✓
First Name: 17	Last Name: 7	103		City: 1	11 Oct 1917	✓
First Name: 17	Last Name: 7	101		City: 2	20 Jan 1920	✓
First Name: 18	Last Name: 8	100		City: 0	21 Feb 1921	✓
First Name: 14	Last Name: 4	100		City: 1	8 Jul 1920	✓
First Name: 1	Last Name: 1	98		City: 1	2 Feb 1923	✓
First Name: 11	Last Name: 1	97		City: 0	23 Aug 1923	✓
First Name: 14	Last Name: 4	96		City: 2	15 Apr 1925	✓
First Name: 5	Last Name: 5	96		City: 0	17 Feb 1925	✓
First Name: 10	Last Name: 0	96		City: 2	22 Jul 1924	
First Name: 2	Last Name: 2	95		City: 0	14 Oct 1925	✓
First Name: 3	Last Name: 3	94		City: 2	24 Feb 1927	✓
First Name: 11	Last Name: 1	94		City: 1	3 Oct 1926	✓
First Name: 16	Last Name: 6	94		City: 0	8 Apr 1927	✓
First Name: 4	Last Name: 4	93		City: 0	25 Mar 1928	✓
First Name: 17	Last Name: 7	93		City: 1	9 May 1928	✓
First Name: 5	Last Name: 5	91		City: 2	8 Nov 1929	✓
First Name: 0	Last Name: 0	91		City: 2	23 Apr 1930	
First Name: 12	Last Name: 2	90		City: 2	4 Nov 1930	✓
First Name: 19	Last Name: 9	89		City: 1	22 Mar 1932	✓
First Name: 9	Last Name: 9	88		City: 2	1 Aug 1932	✓
First Name: 2	Last Name: 2	87		City: 1	25 Jun 1933	✓

Abbildung 1: JavaFX basierte Tabelle mit Sortier- und Filtermöglichkeiten

1.2 Ziele des Projekts

Mit Compose BigLazyTable soll eine Library entwickelt werden, die den Umgang mit grossen Datenmengen End-to-End effizient unterstützt. Entwickler sollen einfach und mit wenig Aufwand die Library in ihre bestehende Applikation integrieren können.

Die Library sollte eine UI-Komponente anbieten, die eine grosse Menge an Daten tabellarisch darstellen kann. Dabei sollen gewohnte Funktionalitäten, wie kontinuierliches Scrolling, Sortieren und Filtern integriert sein. Für ein optimales Handling mit grossen Datenmengen, sollte ein *Paging-Mechanismus* eingebaut werden, um Daten möglichst spät zu laden und nur nötige Daten im Memory zu haben. Der End-Benutzer soll die Möglichkeit haben, die Daten zu modifizieren. Im Hintergrund sollte eine geeignete Struktur aufgebaut werden, welche nur notwendige Daten bei einem Service abfragt und diese Zwischenspeichert. Die Library sollte mit jeder beliebigen Datenquelle eingesetzt werden können. Ein Merge-Mechanismus muss dafür sorgen, dass bearbeitete Daten nicht verlorengehen beim Scrollen.

Ziel ist es, zur Veranschaulichung der Library eine Demo-Applikation bereitzustellen, welche mit mehr als 100'000 Datensätzen arbeitet.

Als übergeordnetes Ziel von Dr. Dieter Holz und der FHNW, gilt es mit diesem Projekt zu beurteilen, wie fortgeschritten *Compose for Desktop* bereits ist. Dabei soll beurteilt werden, ob Kotlin mit *Compose for Desktop* eine Alternative für den Unterricht an der FHNW sein kann und die aktuell unterrichteten Sprachen Java und JavaFX im Studiengang iCompetence ablösen könnte.

1.3 Resultat

Es wurde eine Library erstellt, welche ein Entwickler einfach in seine bestehende Geschäftsapplikation einbauen kann. Das UI besteht aus einer Tabelle, welcher alle Datensätze anzeigt und einer *Compose Forms* Komponente, welche einen in der Tabelle ausgewählten Datensatz genauer anzeigt und die Möglichkeit der Bearbeitung bietet.

Für ein schnelles Laden der vielen Datensätze wurde im Hintergrund ein selbstentwickelter *Paging-Mechanismus* gebaut. Dieser lädt nicht alle Daten auf einmal, sondern in Paketen, welche als Pages bezeichnet werden. Für den End-Benutzer ist eine Page im UI sichtbar. Je zwei Pages sind davor und danach schon geladen. Die restlichen Datensätze werden erst bei einem Scroll-Vorgang in einen noch nicht geladenen Bereich, geladen.

Die Tabelle bietet zudem Funktionalitäten, wie Filtern und Sortieren an. Der End-Benutzer hat die Möglichkeit über einen Klick auf den Header einer Spalte, die Daten unsortiert, aufsteigend oder absteigend sortiert anzeigen zu lassen. Die Filter sind für jede Spalte separat verfügbar. Mit einer vordefinierten Syntax, welche im Library-README dokumentiert ist, kann der End-Benutzer verschiedene Filter auf den Spalten anwenden. Wie aus anderen Anwendungen gewohnt, können Filter auch kombiniert werden, sowie gleichzeitig mit dem Sortieren angewendet werden.

Ein angewählter Datensatz wird neben der Tabelle im *Compose Forms* angezeigt. Dort gibt es diverse Möglichkeiten, diesen Datensatz zu bearbeiten. Die Änderungen werden sofort auch in der Tabelle sichtbar und bleiben auch persistiert, wenn man in der Tabelle weg scrollt. Über das *Compose Forms* kann zudem die Sprache eingestellt werden. Dies wirkt sich auf das *Compose Forms* sowie auch auf die Header Zeile in der Tabelle aus.

Die Auswahl der Datenquelle ist dem Entwickler freigestellt. Er muss lediglich ein von Compose BigLazyTable zur Verfügung gestelltes Paging-Service-Interface implementieren, welches alle Funktionalitäten für das Filtern, Sortieren und den *Paging-Mechanismus* abdeckt.

Filter	Filter	Filter	Filter	Filter
ID	No. of edits	Name	Modified at	Track artist
172371	15	Ok	1455560800	Future
10355	17	It's Okay	1508112000	The Paper Kites
312498	35	It's Okay	1474243200	Cage The Elephant
415851	12	It's Okay	1485043200	Solange
747102	9	It's Okay	1475020800	ZAYN
10684	9	Ok	1496016000	Billy Bragg
70112	44	Ok	1449360000	Carrie Underwood
109352	89	Ok	1482451200	Lupe Fiasco
179821	13	Ok	1469836800	Joy Division
185739	37	Ok	1507593600	NF
226196	13	Ok	1507852800	Caravan Palace
242683	23	Ok	1500681600	Seether
255102	19	Ok	1508630400	King Harvest
287215	54	Ok	1496275200	Kendrick Lamar
296013	73	Ok	1461283200	alt-J
318452	14	Ok	1444521600	Drake
372573	12	Ok	1503792000	Rich Homie Quan
388186	21	Ok	1419033600	The 1975
482288	12	Ok	1471132800	Just A Gent
503600	47	Ok	1509148800	50 Cent
513020	12	Ok	1501372800	Machine Gun Kelly
522280	33	Ok	1468281600	HUNNY
625100	21	Ok	1500595200	J. Cole
642419	22	Ok	1505692800	Dirty Heads
645210	13	Ok	1418947200	Boastio Boys
652189	14	Ok	1454544000	Wes Walker

Abbildung 2: Compose BigLazyTable kombiniert mit einem Compose Forms

Für einen Entwickler biete Compose BigLazyTable deutliche Vorteile im Vergleich zu einer eigenen Lösung. Die Entwicklungszeit wird durch den Einsatz der Library stark reduziert. Für die Implementation muss ein Service geschrieben werden, welcher mit der vom Entwickler gewählten Datenquelle zusammenarbeitet. Um die notwendige Funktionalität abzudecken, kann dazu einfach ein Service-Interface von Compose BigLazyTable verwendet werden, welches die geforderte Funktionalität beschreibt und den Aufbau vorgibt. Neben dieser Implementation muss der Entwickler nur noch ein paar Konfigurationen für *Compose Forms* einstellen, wie die angezeigten Labels oder der Aufbau des Models, welches hinter dem Formular steckt. Hier bietet *Compose Forms* auch Interfaces an und eine ausführliche Dokumentation.

Für eine funktionierende Applikation mit einer einfachen Datenquelle dahinter, braucht ein Entwickler ungefähr zwei Tage für die Implementation. Dies ist im Vergleich zu den 720 Stunden dieser Projektarbeit deutlich weniger, was für den Einsatz von Compose BigLazyTable spricht.

1.4 Struktur des Berichts

Der Hauptteil gliedert sich in drei Kapitel. Im Kapitel [Technische Evaluation](#) werden technische Entscheidungen zum Projekt beschrieben. Anschliessend wird im Kapitel [Methodik](#) erläutert, welche technische Infrastruktur verwendet und wie das Projekt organisiert wurde. Im letzten Teil [Lösungsaufbau](#) wird aufgezeigt, wie die komplette Software-Architektur aufgebaut ist.

Daran anschliessend folgt eine [Schlussfolgerung](#) zum Projekt und eine [Reflexion](#) des Projektteams.

Am Ende sind das [Literaturverzeichnis](#), das [Abbildungsverzeichnis](#), das [Tabellenverzeichnis](#), der [Anhang](#) und die [Ehrlichkeitserklärung](#) zu finden. Fachbegriffe und Abkürzungen sind kursiv geschrieben und werden im [Glossar](#) an erster Stelle im Anhang erklärt.

1.5 Projekt-Links

Library Compose BigLazyTable (Code): <https://github.com/FHNW-IP5-IP6/ComposeBigLazyTable.git>

Verwendung der Library: <https://github.com/FHNW-IP5-IP6/ComposeBigLazyTable#readme>

Kanban-Board (Issues): <https://github.com/FHNW-IP5-IP6/ComposeBigLazyTable/projects/1>

2 Technische Evaluation

In diesem Kapitel wird aufgezeigt, welche Programmiersprachen, Frameworks und Programmier-Mechanismen verwendet wurden. Im ersten Teil ist ein Vergleich von Kotlin und Java und es wird aufgezeigt, welche Features von Kotlin im Projekt genutzt wurden. Danach folgt eine Beschreibung zu *Compose Multiplatform*. Zum Schluss wird noch das Thema *Paging-Mechanismus* genauer erläutert.

2.1 Kotlin

In diesem Unterkapitel wird Kotlin genauer beschrieben. Es wird nur auf die Sprach-Features eingegangen, welche in Compose BigLazyTable zum Einsatz gekommen sind. Als Einstieg wird ein kurzer Vergleich zwischen Kotlin und Java gemacht und die Vorteile von Kotlin beschrieben.

Kotlin ist eine Open-Source Programmiersprache der Firma JetBrains. Die Sprache wurde zum ersten Mal im Jahr 2011 veröffentlicht. Ähnlich den Sprachen Java, C++ und C, ist Kotlin eine statisch typisierte High-Level-Programmiersprache und unterstützt sowohl die objektorientierte wie auch die prozedurale Programmierung. Es werden Java Virtual Machine (JVM), Android, JavaScript und Native unterstützt. Kotlin wird von Google stark gepusht und ist aktuell in der stable Version 1.6.10 verfügbar, welche ebenfalls bei Compose BigLazyTable zum Einsatz kommt.

2.1.1 Vergleich Kotlin und Java

Kotlin ist sehr pragmatisch gehalten und bietet im Vergleich zu Java eine einfachere Syntax, während die Funktionalität praktisch gleichbleibt. Die Sprache unterstützt vollständig Funktionen höherer Ordnung, unterschiedliche Funktionstypen und Lambda-Ausdrücke, was vor allem in der funktionalen Programmierung gebraucht wird. Auch eine objektorientierte Programmierung, wenn nötig in Kombination mit funktionaler Programmierung, ist kein Problem. [7]

Folgende Vorteile bietet Kotlin:

- **Volle Kompatibilität mit Java**
Kotlin bietet Support zu allen vorhandenen Java-Frameworks und Libraries. Es können einfach aus Kotlin-Funktionen, Klassen und Methoden von Java aufgerufen und genutzt werden. Zudem ist die Integration von Kotlin in Build-Systeme wie Maven, Gradle und andere problemlos möglich.
- **Einfache Sprache**
Durch die einfache Syntax von Kotlin, ist die Sprache leicht zu erlernen. Es sind nicht zwingend spezifische Kenntnisse erforderlich für die Programmierung mit Kotlin.

- **Gute IDE Unterstützung**

Kotlin kann sehr einfach in die beliebtesten IDEs integriert werden. Zudem bietet IntelliJ eine automatische Konvertierung von Java-Code nach Kotlin und zurück an.

- **Nullsicherheit**

Kotlin erlaubt keine NullPointerExceptions beim Kompilieren oder Ausführen von Code. Dies ist einer der Hauptvorteile von Kotlin gegenüber Java. Datentypen können in zwei Gruppen aufgeteilt werden: die null-fähigen (Fragezeichen am Ende der Deklaration) und die nicht nullfähigen, welche den Wert „null“ nicht annehmen dürfen.

- **Einfache Syntax**

Die Produktivität von Entwicklern kann gesteigert werden durch die einfache Syntax von Kotlin. Im Vergleich zu Java werden deutlich weniger Codezeilen benötigt, was die Programmierzeit und Entwicklungskosten senkt. Es ist möglich mit Kotlin den Code bis zu 40% zu reduzieren.

2.1.2 Kotlin Coroutines

Kotlin bietet eine eigene Umsetzung von Multithreading, die sogenannten Coroutines (Coroutines). Andere Programmiersprachen lösen das Problem von Asynchronität und Parallelität mit Hilfe von Threads. Threads verursachen jedoch einen Overhead und benötigen zudem auch Callbacks, welche den Code schnell unübersichtlich machen. Ähnlich zu Threads, können Coroutines auch parallel ausgeführt werden, warten und untereinander Daten austauschen. Der Unterschied zu Threads besteht darin, dass Coroutines kaum Arbeitsspeicher verbrauchen und nicht an das Betriebssystem gebunden sind, sondern vom Entwickler selbst gesteuert werden. Programme mit Coroutines können sequenziell und ohne Callbacks geschrieben werden. Anstatt einen Thread zu starten und über ein Callback auf das Ergebnis zu warten, kann in der nächsten Zeile mit dem Ergebnis einfach weitergearbeitet werden.

Um Coroutines verwenden zu können, muss Kotlin mindestens in der Version 1.3 verfügbar sein. Zudem muss zusätzlich die „coroutines-core“-Dependency hinzugefügt werden.

```
suspend fun showMails() {  
    val emails = retrieveMails()  
  
    emails.forEach {  
        println(it)  
    }  
}  
  
suspend fun retrieveMails(): List {  
    ... //get mails from the internet  
    return ...  
}
```

Abbildung 3: Beispiel von sequenzieller Programmierung mit Coroutines.

Wie in [Abbildung 3](#) zu sehen ist, bietet Kotlin das Keyword „suspend“ an. Dies kennzeichnet eine suspending function und ist ein Hinweis, dass diese Funktion asynchron ist und nicht sofort ein Ergebnis liefern muss. Die Funktion `retrieveMails()` stellt eine potenziell blockierende Operation dar, zum Beispiel einen Network Call. Ohne die Kennzeichnung mit `suspend` würde diese Funktion den aufrufenden Thread so lange blockieren, bis die Funktion abgeschlossen ist. Man müsste sie also zum Beispiel in einem eigenen Thread ausführen. Speziell für suspending functions ist, dass diese nicht aus einer normalen Funktion aufgerufen werden können. Sie müssen über einen Coroutine Builder gestartet werden.

```
fun main(args: Array) {  
    launch { showMails() }  
}
```

Abbildung 4: Coroutine Builder `launch{}` für den Aufruf einer suspending function.

Wird nun mit der Funktion aus [Abbildung 4](#) die Anwendung gestartet, wird keine Ausgabe gemacht. Dies liegt daran, dass `launch` die suspending function aufruft, sich aber nicht für das Ergebnis interessiert. Coroutines verhalten sich wie Daemon-Threads, das heisst sie halten den Prozess, aus dem sie gestartet wurden, nicht am Leben. Die `main`-Funktion würde also terminieren, bevor alle Mails abgeholt worden sind. Damit dies verhindert werden kann, muss irgendwo auf das Ergebnis gewartet werden, zum Beispiel mit einem `join()` und dem Coroutine Builder `runBlocking{}.`

```
fun main(args: Array) {  
    val job = launch { showMails() }  
    println("You've got mail:")  
  
    runBlocking {  
        job.join()  
    }  
}
```

Abbildung 5: Coroutine Builder `"runBlocking{}"` und `"join()"` für das Warten auf die suspending function.

Die `join`-Funktion in [Abbildung 5](#) wartet darauf, dass der Job beendet wird. Sie ist selbst eine suspending function und blockiert darum auch nicht den aktuellen Thread. Damit dies korrekt funktioniert, muss sie über den Coroutine Builder `runBlocking{}` ausgeführt werden. Um eine Verbindung zwischen der asynchronen und der synchronen Welt herzustellen, bietet es sich deshalb an, direkt die `main`-Funktion als Einstiegspunkt in die gesamte Applikation als blocking zu definieren.

```
fun main(args: Array) = runBlocking {  
    val job = launch { showMails() }  
    println("You've got mail:")  
  
    job.join()  
}
```

Abbildung 6: `"main"`-Funktion als blocking definiert.

Coroutines sind mit diesen Features deutlich leichtgewichtiger als Threads. Es können problemlos 100'000 Coroutines gleichzeitig gestartet werden, ohne einen `OutOfMemoryError` zu bekommen. Würden 100'000 Threads gleichzeitig gestartet werden, würde dies ziemlich sicher zu einem `OutOfMemoryError` führen, da ein Thread je nach Plattform zwischen 256KB und 1024KB benötigt. [8]

2.1.3 Kotlin Eigenheiten

2.1.3.1 Lazy Delegate / Lazy Evaluation

Damit ein Block von Code der immer das gleichen Ergebnis liefert, nicht immer wieder neu ausgeführt wird, gibt es das Keyword `lazy`. Mit `lazy` wird Lazy evaluiert. Damit kann anhand eines Ausdrucks wie `val color by lazy { getColor(Blue) }` verhindert werden, dass jedes Mal wenn die `color` Variable gesetzt wird, die Funktion `getColor` ausgeführt wird. Somit wird nur das erste Mal der Wert von `getColor` evaluiert und anschliessend gespeichert. Dieser Mechanismus kann von grossem Vorteil sein, wenn ein Funktion aufgerufen wird, die längere Zeit zum Prozessieren braucht. Anstelle immer wieder neu warten zu müssen, kennt man den Wert der Funktion bereits. Dieser Mechanismus konnte an einem Ort in *Compose Forms* eingebaut werden, um zu verhindern, dass ständig eine Funktion für das Erhalten der Farbe, aufgerufen wurde. Die `get` Aufrufe konnten von mehr als 10'000 Aufrufen zu gerade einmal 13 Aufrufen verbessert werden. Im Kapitel [Zustand merken](#) wird auf einen ähnlichen Mechanismus im *Compose* Umfeld eingegangen.

2.1.3.2 Scope Functions: `apply`, `let`, `with`

Kotlin bietet sogenannte Scope Functions an, die einen Codeblock im Kontext eines Objekts ausführen können. Beim Aufruf einer solche Funktion wird ein temporärer Gültigkeitsbereich erzeugt, in welchem auf das Objekt ohne seinen Namen zugegriffen werden kann. Neben `apply`, `let` und `with` gibt es auch noch weitere, welche aber nicht im Projekt verwendet wurden.

`Let` wird verwendet, um Code nur auszuführen, wenn ein Objekt, auf dem der Code aufgerufen wird, nicht null ist. Mehr dazu im nächsten Kapitel [Null Safety](#). Ausserdem dient `let` dazu einen Ausdruck als Variable im lokalen Gültigkeitsbereich entweder als `it` oder als benannter Parameter einzuführen. In [Abbildung 27](#) zu sehen.

`Apply` wird verwendet, wenn auf einem Objekt mehrere Sache konfiguriert werden sollen. Ohne `apply` würde eine Variable erstellt und dann auf dieser die Konfigurationen aufgerufen werden. In Kotlin können solche Konstrukte mit `apply` ersetzt und innerhalb des `apply` Block das Objekt konfiguriert werden. Da `apply` das Objekt selbst zurückgibt, kann immer noch eine Variable gemerkt werden, um auf diese zu einem späteren Zeitpunkt weitere Änderungen zu vollziehen. In [Abbildung 67](#) ist ein Beispiel von `apply` zu sehen und ebenfalls, dass innerhalb eines `apply` Blocks mit `this` auf das Objekt zugegriffen werden kann.

`With` macht im Grunde genommen etwas ähnliches wie `apply`. Der Unterschied ist, dass `apply` auf dem Objekt aufgerufen wird. `With` hingegen ist keine Extension Funktion und das Objekt wird als Parameter mitgegeben. Ein weiterer Unterschied zu `apply` ist, dass der Rückgabewert von `with` Unit ist. [17]

```
with(controller) { this: LazyTableController<T>
    val recomposeTrigger = recomposeStateChanger

    if (isLoading) {
        Box(Modifier.fillMaxWidth().padding(16.dp)
            CircularProgressIndicator(modifier =
        )
    }
}
```

Abbildung 7: Beispiel von `with` im *Compose BigLazyTable* Code.

2.1.3.3 *Null Safety*

Kotlin sticht durch seine Null Safety hervor. Werte können nur null sein, wenn sie als solche angegeben werden. Dabei wird hinter dem Typ ein Fragezeichen gesetzt. Wenn auf einem nullable Typ eine Funktion aufgerufen wird, wird dies mit `nullable?.callFunction()` gemacht. Wenn entsprechend reagiert werden möchte falls die Variable null ist, wird dies mit dem Elvis Operator `?:` gemacht. Ein Aufruf würde dann so aussehen: `nullable?.callFunction() ?: println(«nullable was null»)`. Ein Weg in Kotlin Abfragen nach null zu ersetzen, stellt folgende Synthax dar: `nullable?.let { it.callFunction() }`. Der Bereich innerhalb der `let` Funktion wird nur ausgeführt, wenn nullable nicht null ist. Ansonsten wird nichts ausgeführt.

2.1.3.4 *Object / Singleton*

In Kotlin ist das Singleton Pattern direkt in die Sprache integriert. Wenn man eine Klasse als `object` markiert, ist diese eine Singleton Klasse und kann von überall direkt aufgerufen werden. Dies wurde beim [DBService](#) und bei der Exposed Tabelle `DatabasePlaylist` verwendet.

2.1.3.5 *Immutability*

In Kotlin sind Klassen, Listen und Variablen mit dem `val` Keyword standardmässig immutable=unveränderbar. Deswegen sieht man oft Ausdrücke wie `mutableListOf` oder `mutableStateOf`, welche eine veränderbare Liste beziehungsweise Zustand erzeugen. Letzterer wird im Kapitel [Compose Eigenheiten](#) genauer beschrieben.

2.2 Compose

In diesem Unterkapitel wird das Compose Framework genauer beschrieben. Es wird auf die Eigenheiten von Compose eingegangen und *Compose Multiplatform* genauer beschrieben.

2.2.1 Compose Eigenheiten

2.2.1.1 Zustandsänderungen

Das deklarative UI Toolkit Compose setzt auf sogenannte States, welche darzustellende Daten fürs UI enthalten. Wenn sich die Daten eines States ändern, wird dies durch das Framework festgestellt und die «betroffenen» Teile des UI werden neu gerendert. Im Compose Jargon spricht man hierbei von einem *Recompose*. Mit betroffen sind alle Teile des UI, welche auf den geänderten State zugreifen. Diese darunterliegende reaktive Art und Weise von Compose, ist Plattformübergreifend gleich, ob auf Android, Desktop oder Web.

```
var attributeFilter = mutableStateMapOf<Attribute<*, *, *>, Filter?>()

var displayedFilterStrings = mutableStateMapOf<Attribute<*, *, *>, String>()

var attributeCaseSensitive = mutableStateMapOf<Attribute<*, *, *>, Boolean>()

var isFiltering: Boolean by mutableStateOf( value: false)
```

Abbildung 8: Die Definierung von States anhand der mutableState(Map)Of(initialValue) Erzeugerfunktionen.

2.2.1.2 Zustand definieren

Zustände oder States kann man auf zwei verschiedene Arten definieren:

Variante	Vorteile	Nachteile
<pre>var state: Boolean by mutableStatOf(false) state = true if (state) ...</pre>	<ul style="list-style-type: none"> - Anhand eines Delegates wird das Abholen und Setzen eines Werts auf dem State Objekt delegiert und es kann direkt auf dem state gelesen und bearbeitet werden. State ist immer vom entsprechenden Typ. - Es muss weniger Code geschrieben werden, für den gleichen Endeffekt. 	<ul style="list-style-type: none"> - getValue und setValue müssen importiert werden. - Wenn innerhalb einer Funktion der Wert des States bearbeitet werden möchte, ist dies nicht möglich da kein State sondern der direkte Wert zurückgeliefert wird.
<pre>val state: State<Boolean> = mutableStateOf(false) state.value = true if (state.value) ...</pre>	<ul style="list-style-type: none"> - Der Wert ist innerhalb des State Objekts enkapselt und man kann ein state Objekt in einer Funktion als Argument mitgeben und anschliessend den State bearbeiten. 	<ul style="list-style-type: none"> - Es muss mehr Code geschrieben werden. - Mehr Code steigert die Komplexität und bietet mehr Möglichkeiten für Fehler.

Tabelle 1: Varianten, um States zu definieren.

Es wurde entschieden, im Normalfall die erste Variante zu verwenden, da dadurch weniger Code geschrieben werden musste und das Weglassen von `.value` auch den Code übersichtlicher macht. Die zweite Variante konnte dank des genannten Vorteils, zweckbringend beim Ändern des `TriStateCheckbox` Werts eingesetzt werden. Wichtig ist, dass es kein richtig oder falsch gibt. Persönliche Präferenz und manchmal gerade der passende Ort für eine Variante entscheiden am Ende welche Variante eingesetzt wird.

2.2.1.3 Zustand merken

Anhand des `Compose` Keyword `remember` können States einen Wert speichern, ohne dass dieser bei einem *Recompose* wieder neu evaluiert und das UI neu gezeichnet wird.

2.2.1.4 Komponenten

Im Verlauf des `Compose BigLazyTable` Projekt wurde in *Compose For Web* eine ernüchternde Entscheidung seitens JetBrains getroffen. Anstelle der aus *Jetpack Compose* und auch in *Compose For Desktop* bekannten deklarativen Standardkomponenten wie `Column`, `Row`, `LazyColumn`, `LazyRow`, `Box` und Weiteren, wurden diese im *Compose Web* Umfeld als `Deprecated` markiert. Als Alternative wurde eine eigene *Compose DOM DSL* umgesetzt. Dies hat zur Folge, dass das zu Beginn von *Compose for Web* versprochene UI Codesharing zwischen allen drei Plattformen nicht mehr möglich ist. Aus diesem Grund wurde für dieses Projekt entschieden, auf *Compose For Web* zu verzichten, da das Web UI komplett von Grund auf hätte neu gebaut werden müssen.

Im Gegensatz dazu sind bei *Jetpack Compose* und *Compose for Desktop* ziemlich alle Standardkomponenten gleich. Während diese bei *Jetpack Compose* als darunterliegende Elemente Android native Komponenten besitzen, sind es bei *Compose For Desktop* AWT/Swing Elemente, mit welchen es auch interoperabel ist.

Um die tatsächliche Code Wiederverwendung zwischen Desktop und Jetpack zu testen, wurde ein kleines Beispiel Android Projekt aufgesetzt und alles nicht UI spezifische auskommentiert. Dabei konnten alle Komponenten bis auf die Horizontal/Vertical-Scrollbar ohne Änderung wiederverwendet werden.

2.2.2 Compose Multiplatform

Compose Multiplatform ist ein Kotlin-Framework für die Entwicklung von Desktop- und Web-UIs, basierend auf dem modernen Toolkit *Jetpack Compose* von Google und bereitgestellt von JetBrains. *Compose Multiplatform* verspricht ein schnelles, reaktives Framework zu sein, das die UI-Entwicklung für Desktop- und Webanwendungen vereinfachen und beschleunigen soll. Es ermöglicht eine weitgehend gemeinsame Nutzung von UI-Code zwischen den Plattformen Android, Desktop und Web. Kotlin wurde ursprünglich als JVM-Sprache gestartet. Später kam Kotlin/JS dazu, um Webanwendungen mit JavaScript zu erstellen. 2017 wurde dann noch Kotlin/Native hinzugefügt, um plattformspezifische Binaries zu erstellen. Kotlin Multiplatform vereint diese drei Zielplattformen und bietet mit der Kombination von *Jetpack Compose* das Framework *Compose Multiplatform* an. [9]

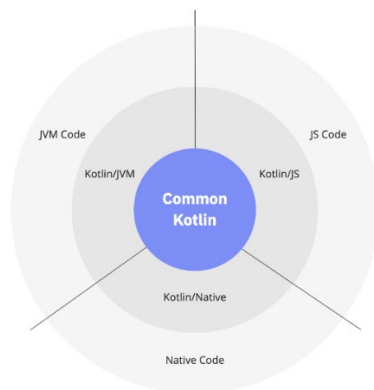


Abbildung 9: Kotlin Multiplatform kombiniert plattformspezifische Libraries und Werkzeuge. [<https://www.heise.de/news/Cross-Plattform-JetBrains-bringt-Compose-auf-den-Desktop-und-ins-Web-6155707.html>]

Compose for Desktop bietet einen deklarativen und reaktiven Ansatz für die Entwicklung von Desktop-Bedienoberflächen mit Kotlin. Das UI kann sehr einfach aus sogenannten Composable-Funktionen zusammengebaut werden. Die IDE und das Build-System bieten eine vollständige Unterstützung an und es werden keine speziellen XML- oder Template-Sprachen benötigt. Unter der Haube läuft die JVM (Java Virtual Machine) und ermöglicht durch die Nutzung der nativen Hochleistungs-Grafikbibliothek *Skia* ein leistungsstarkes, hardwarebeschleunigtes UI-Rendering auf macOS, Windows und Linux. Mit einem Klick können einfach native Distributionen von einer Applikation erstellt werden. *Compose for Desktop* wurde Ende 2020 als alpha-Release veröffentlicht. Anfang Dezember 2021 wurde die erste stable Version veröffentlicht. Die aktuelle Version ist die Version 1.1.1, welche Mitte März 2022 veröffentlicht wurde. *Compose BigLazyTable* verwendet aktuell die Version 1.1.0. [4, 6]

Compose for Web bietet die Möglichkeit, reaktive Web-Bedienoberflächen in Kotlin zu entwickeln. Dabei werden der Zustand, das Verhalten und die Logik der Applikation mithilfe der Konzepte und APIs von *Jetpack Compose* definiert. Zudem bietet es zahlreiche Möglichkeiten, Bedienoberflächen in Kotlin zu deklarieren, sodass mit einer deklarativen DOM-API die vollständige Kontrolle über das Website-Layout gegeben ist. *Compose for Web* ist im GitHub Projekt von *Compose Multiplatform*, by JetBrains (<https://github.com/JetBrains/compose-jb>) enthalten und somit auch in der Version 1.1.1 verfügbar. [5]

2.3 Paging Mechanismus

Jetpack Compose bietet neben vielen anderen Libraries, eine eigene Paging-Library an, welche aktuell unter Jetpack Paging 3.0 veröffentlicht ist. Google ist im Bereich Libraries sehr aktiv und pusht gewisse Libraries als «Jetpack» Libraries, welche viele zusätzliche Funktionalitäten für den Entwickler bringen. *Compose for Desktop* hat bisher nur sehr wenige Libraries, da es im Vergleich zu *Jetpack Compose* auch noch sehr neu ist. Da die *Jetpack Compose* Paging Library noch nicht zu *Compose for Desktop* migriert wurde, musste hier ein eigener Mechanismus gebaut werden.

In einem ersten Schritt wurden Informationen über die Möglichkeiten bei der Implementierung von einem *Paging-Mechanismus* gesammelt. Hilfreich dabei war auch der Code der *Jetpack Compose* Variante; für die Anwendung bei Compose BigLazyTable war er jedoch zu komplex. Am Anfang der Entwicklung musste entschieden werden, welche Variante des Pagings gewählt wird. Dabei wurde entschieden eine eigene Variante umzusetzen, welche im Kapitel [LazyTableController](#) genauer beschrieben wird.

3 Methodik

In diesem Kapitel wird aufgezeigt, wie gearbeitet wurde. Als Arbeitsmethode ist [Kanban](#) gewählt worden, was im ersten Unterkapitel genauer erläutert wird. Für das Code-Management wurde GitHub eingesetzt. Das Projekt ist dort standardmässig aufgesetzt, mit einem master-Branch, einem develop-Branch und den entsprechenden feature-Branches.

Für das automatisierte Testen wurden [SonarQube](#) und lokal [SonarLint](#) eingesetzt. Wie diese Tools eingesetzt worden sind, wird im Nachfolgenden noch genauer erklärt.

Zum Schluss wird noch erklärt, mit welchen Daten getestet wurde und wie diese genau aufgearbeitet wurden.

3.1 Kanban

Für die Entwicklung von Compose BigLazyTable wurde als Arbeitsmethode Kanban gewählt. Als Tool wurde GitHub gewählt, welches mit der Projects Funktionalität sehr gute Unterstützung bietet. Mehrheitlich wöchentliche Meetings mit dem Auftraggeber Dr. Dieter Holz stellten sicher, dass sich das Produkt in die richtige Richtung entwickelte und die Priorisierungen entsprechend gesetzt wurden. Die Erkenntnisse und neu entstandene Issues wurden dann in GitHub auf dem Kanban Board erfasst.

3.1.1 Grundlagen

Das Wort Kanban stammt aus dem Japanischen und bedeutet übersetzt „Schild“ oder „visuelles Signal“. Es kommt ursprünglich von Toyota. Als diese 1940 die „Just in Time“-Produktion einführten wurde von einem Push-System zu einem Pull-System gewechselt. Die Produktion wurde dabei an der Kundennachfrage ausgerichtet und nicht auf eine bestimmte Menge festgesetzt. Dieses neue Produktionssystem legte den Grundstein für eine sogenannte Lean-Produktion. Ihr Hauptzweck ist die Minimierung von Aktivitäten, die zu Verlusten führen, ohne die Produktivität zu beeinflussen. Hauptziel ist es, ohne Zusatzkosten einen Mehrwert für die Kunden zu schaffen. David J. Anderson sorgte dafür, dass die Kanban-Methode auch den Weg in die Wissensarbeit fand. [1]

Es gibt viele verschiedene Umsetzungen von Kanban, jedoch fokussieren alle auf folgende Grundpraktiken:

- **Workflow visualisieren**

Alle Schritte müssen visualisiert sein. Es wird ein Board mit Karten und Spalten benötigt. Jede Karte repräsentiert ein Arbeitselement und jede Spalte ein Schritt im gesamten Workflow. Auf diese Weise kann problemlos der Fortschritt verfolgt werden und Engpässe werden aufgedeckt.

- **Laufende Arbeit begrenzen**
Die laufende Arbeit muss begrenzt sein. Sogenannte Work-in-Progress-Limits (WIP) stellen dies sicher. Eine Karte kann nur eine Spalte weiter, wenn dort genügend Kapazität zur Verfügung steht.
- **Workflow Management**
Der Flow muss gemessen und angepasst werden. Ideal wäre ein schneller und gleichmässiger Fluss, um das Risiko zu minimieren und Verzögerungen zu vermeiden.
- **Prozessrichtlinien ausformulieren**
Der Prozess muss klar definiert und gemeinsam besprochen werden. Dies hilft bei der Verbesserung und einer höheren Beteiligung, da es für alle nachvollziehbar ist.
- **Feedbackschleifen**
Mithilfe von kurzen Meetings mit dem Kanban-Board wird der Wissenstransfer ermöglicht und Probleme im Team können gelöst werden.
- **Zusammenarbeit verbessern**
Durch gemeinsame Theorien zu Arbeit, Workflow, Prozessen und Risiken wird ein gemeinsames Verständnis von Problemen gefördert.

Kanban unterscheidet sich zu Scrum, da bei Kanban die Durchlaufzeit (Cycle Time) gemessen wird und bei Scrum die Team-Geschwindigkeit (Velocity). Neue Anforderungen können bei Kanban jederzeit eingebaut werden, solange es Kapazitäten gibt. Folgende Metriken werden bei Kanban gemessen:

- **Cumulative Flow Diagram (CFD)**
Visualisiert die Zustände an allen Stationen des Kanban-Systems. Es zeigt so schnell, wo sich Bottlenecks befinden.
- **Work in Progress (WIP)**
Zeigt welche Anzahl Issues sich gleichzeitig in einem Status befinden können.
- **Durchsatz**
Durchschnittliche Anzahl Issues, die ein Team in einem bestimmten Zeitraum erledigen kann.
- **Fehlerrate**
Anzahl der Bugs, die sich über die Zeit entwickeln. Kurze Durchlaufzeiten sind nur möglich durch hohe Qualität.

3.1.2 Umsetzung

Compose BigLazyTable eignete sich sehr gut für den Einsatz von Kanban. Mit der Kombination von GitHub Projects konnte ein einfaches Kanban Board erstellt werden. Für das Board wurden folgende Spalten verwendet:

Spalte	Beschreibung
To do	Sammlung aller Issues (wie Backlog in Scrum)
To do for active Milestone	Hier befinden sich alle Issues, welche im aktuellen Meilenstein umgesetzt werden möchten. Die Issues sind alle mit einer Beschreibung ausgestattet und wurden mit dem Auftraggeber besprochen.
In progress (WIP = 3)	Wenn ein Issue entwickelt wurde, hat man es im ersten Schritt nach In progress verschoben. Hier dürfen maximal drei gleichzeitig sein.
Review in progress (WIP = 2)	Wenn die Entwicklung eines Issues abgeschlossen war, wurde ein Pull Request eröffnet. Dieser hat das Issue automatisch in die Spalte Review in progress verschoben.
Reviewer approved	Wenn das Review akzeptiert wurde, wurde das Issue in diese Spalte verschoben.
Done	Wenn ein Pull Request gemerged wurde, wurde das verknüpfte Issue automatisch nach Done verschoben.

Tabelle 2: Beschreibung der verwendeten Kanban-Board Spalten.

Alle erstellten Issues wurden immer mit entsprechenden Labels versehen. Beispiel-Labels sind «enhancement», «bug» oder «documentation».

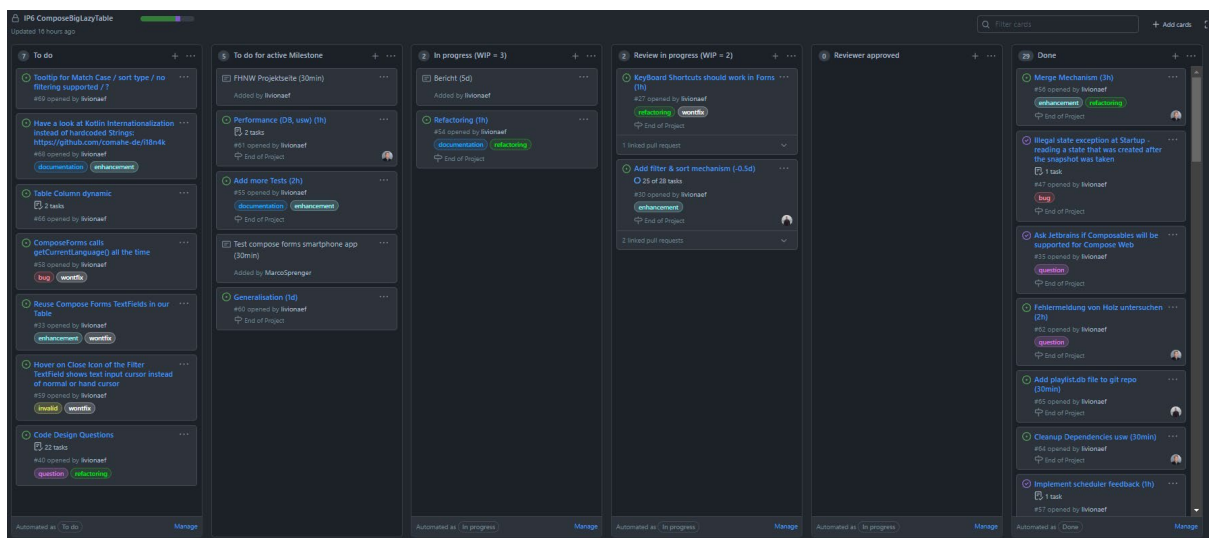


Abbildung 10: Kanban-Board auf GitHub Project von Compose BigLazyTable

3.1.3 Fazit

Kanban ist eine sehr unkomplizierte Methode und einfach anzuwenden. Das Kanban-Board visualisiert jederzeit den aktuellen Stand des Projektes und hilft bei Meetings direkt einen Überblick zu haben. Durch die „Work-in-Progress“-Limiten wird sichergestellt, dass nicht zu viele Issues auf einmal erledigt werden und daher nicht richtig fertiggestellt sind. So fällt die Konzentration auf einen Issue und dessen Abschliessung leichter. Da im Projekt auch viel Pair-Programming zum Einsatz gekommen ist, konnten einige Tasks die Review Spalten direkt überspringen. Dies stellte aber kein Problem dar und der Prozess konnte trotzdem mit einem Pull Request eingehalten werden.

3.2 Codeverwaltung

Die Code-Verwaltung wurde mit GitHub umgesetzt. Es wurden die Grundfunktionalitäten mit Branches und Pull Requests genutzt, sowie das Kanban-Board Feature, wie im Kapitel [Kanban](#) beschrieben. Zudem wurde das Feature der GitHub Actions eingesetzt, wie im nachfolgenden Kapitel [SonarQube](#) beschrieben wird.

Es wurden zwei Haupt-Banches bewirtschaftet. Einmal der master-Branch und der develop-Branch. Auf dem main-Branch befand sich jederzeit eine laufende und getestete Version, welche mit einer Versionsnummer veröffentlicht werden konnte. Auf dem develop-Branch befand sich jederzeit eine laufende Version mit allen neusten Entwicklungen, die getätigt wurden.

Für das Programmieren von neuen Funktionalitäten wurde im ersten Schritt ein neues GitHub Issue erstellt. Anschliessend wurde dieses Issue eine Person zugewiesen und auf dem Kanban-Board nach «in Progress» verschoben. Für das Hinzufügen des neuen Codes wurde ein neuer Branch erstellt. Dieser wurde jeweils mit einem kurzen Stichwort und anschliessend der GitHub Issue Nummer benannt. Ein Issue galt als abgeschlossen, wenn die Funktionalität umgesetzt, die Tests dazu geschrieben und allfällige Probleme im Code behoben wurden.

War ein Issue fertig, so wurde der Code auf GitHub gepusht und dort ein neuer Pull Request auf den develop-Branch eröffnet. Durch das Eröffnen eines neuen Pull Requests, wurde das Issue automatisch von GitHub auf dem Kanban-Board in die Review Spalte verschoben. Als Reviewer musste immer eine Person eingetragen werden. Somit konnte sichergestellt werden, dass die Qualität des Codes hoch ist und beide Projektmitglieder den ganzen Code kennen. Im Code Review konnten Codeabschnitte kommentiert und wenn nötig Änderungen vorgeschlagen werden. Wenn der Reviewer den Pull Request akzeptierte, konnte dieser gemerged werden und der Branch wurde anschliessend gelöscht. Das Issue wurde durch diesen Vorgang automatisch auf Done gesetzt. Somit lief auf dem develop Branch automatisch eine Pipeline durch, die SonarQube beinhaltet. Diese wird von GitHub automatisch eingerichtet, wird im Repository ein Ordner .github/workflows bereitgestellt, der die Konfiguration der Pipeline als yaml-File enthält. Mehr dazu ist im folgenden Kapitel zu finden.

Sobald auf dem develop ein gewisser Stand erreicht wurde und eine ausreichend grosse Anzahl neuer Funktionalitäten dazu gekommen war, konnte ein Merge auf den master-Branch durchgeführt werden.

3.3 SonarQube

Für die Sicherstellung einer hohen Softwarequalität wurde SonarQube eingesetzt. SonarQube ist ein Code-Analyse Tool, welches Entwicklern dabei hilft, Fehler, Sicherheitslücken, Code-Duplikationen und Verletzungen von Coding-Standards zu identifizieren. Das Tool kann einfach für jederzeit aktuelle Analysen von neu geschriebenem Code in eine CI/CD-Pipeline eingebaut werden. SonarQube setzt ein Standard-Set für Kotlin ein, welches verschiedene statische Regeln definiert, die bei der Code-Analyse eingesetzt werden. [2]

Mit der Hauptmetrik „Quality Gate“ wird angegeben, ob der Code für den gewählten Analyse-Typ ausreichend ist. Mit weiteren Metriken wie der Test-Coverage oder einer Zeitschätzung für die Behebung eines gefundenen Fehlers, bekommt der Entwickler weitere wichtige Resultate.

Für Compose BigLazyTable wird die SonarQube Analyse durch ein push auf den develop-Branch ausgelöst. Mithilfe von GitHub Workflows kann diese Pipeline genauer konfiguriert werden.

```
name: Sonarqube
on:
  pull_request:
  push:
    branches:
      - develop

jobs:
  build:
    name: Sonarqube
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0 # Shallow clones should be disabled for a better relevancy of analysis
      - name: Set up JDK 11
        uses: actions/setup-java@v1
        with:
          java-version: 11
      - name: Cache SonarQube packages
        uses: actions/cache@v1
        with:
          path: ~/.sonar/cache
          key: ${ runner.os }-sonar
          restore-keys: ${ runner.os }-sonar
      - name: Cache Gradle packages
        uses: actions/cache@v1
        with:
          path: ~/.gradle/caches
          key: ${ runner.os }-gradle-${ hashFiles('**/*.gradle') }
          restore-keys: ${ runner.os }-gradle
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN } # Needed to get PR information, if any
          SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
          SONAR_HOST_URL: ${ secrets.SONAR_HOST_URL }
        run: ./gradlew build sonarqube --info
```

Abbildung 11: yaml-Skript für die GitHub Pipeline.

```
plugins { this: PluginDependenciesSpecScope
  id("org.sonarqube") version "3.3"
}

// sonarqube configuration
sonarqube { this: SonarQubeExtension
  properties { this: SonarQubeProperties
    property("sonar.projectKey", "common")
  }
}
```

Abbildung 12: Ausschnitt aus dem build.gradle File für die Konfiguration von SonarQube.

Bei den Ergebnissen der Analyse wurde vor allem auf Fehler und Code Smells geachtet. Diese wurden regelmässig angeschaut und im Code angepasst, was zu einer besseren Lesbarkeit und Wartbarkeit des Codes führte. Ausserdem ist SonarQube sehr stark in der Erkennung von zu komplex aufgebauten Stellen. Somit konnten bestimmte Funktionen und Code-Abschnitte vereinfacht werden.

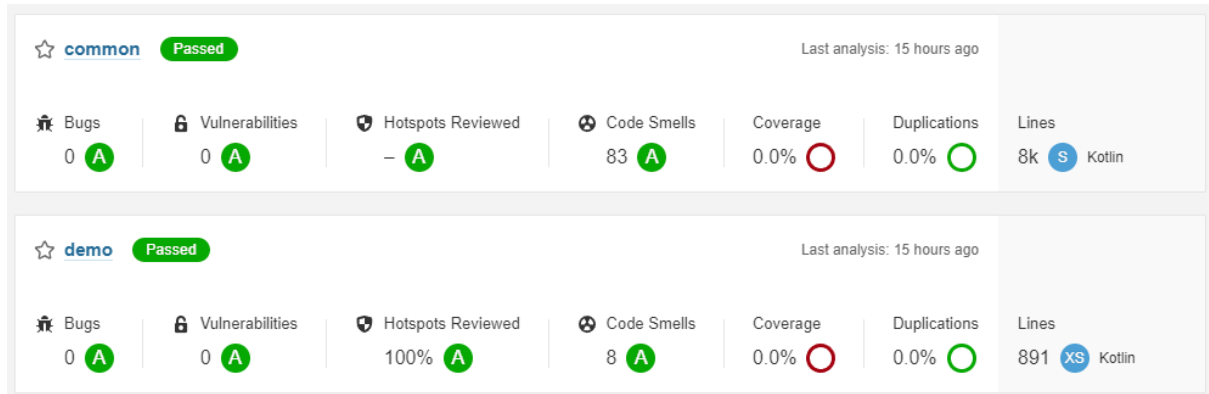


Abbildung 13: Ausschnitt Compose BigLazyTable Projekt auf SonarQube (Stand: 23.03.2022)

Rückblickend auf dieses Projekt stellt SonarQube besonders bei der Verbesserung der Code-Qualität ein hilfreiches Tool dar. Die Auswertungen sind übersichtlich dargestellt und für den Entwickler ausreichend beschrieben.

3.4 SonarLint

SonarLint ist eine IDE-Erweiterung, welche wie SonarQube Code-Analysen durchführen kann, dies jedoch bereits lokal während dem Programmieren umsetzt. SonarLint ist bei *IntelliJ* als Plugin verfügbar und kann mit einem Klick installiert werden. [3]

Der Code wird direkt in der IDE von SonarLint markiert, für den Fall, dass Code Smells oder sonstige Probleme auftreten. Somit wird der Entwickler direkt darauf aufmerksam und kann das Problem an der Stelle beheben, an der er gerade arbeitet.

Durch den Einsatz von SonarLint können schon viele Probleme im Voraus erkannt und behoben werden. Dadurch muss der Entwickler im Anschluss an den SonarQube Check weniger Code erneut anschauen und anpassen.

IntelliJ bietet bereits Hinweise zum Code, welche sehr hilfreich sind. Diese helfen vor allem auch Kotlin Neueinsteigern. In Kombination mit SonarLint bieten sie eine grosse Unterstützung beim Einhalten von Best-Practises.

3.5 Aufarbeitung der Daten

Um das richtige Testen der Library zu ermöglichen, wurde ein Datensatz mit realen Daten organisiert und in der Demo-Applikation eingebaut. Der Datensatz stammt von Spotify und enthält 1 Million Datensätze. Ein Element repräsentiert eine Spotify-Playlist einer US-Person. Bei einer Playlist stehen jeweils verschiedene Informationen zur Verfügung, wie zum Beispiel Name, Anzahl Likes oder Anzahl Songs. Zudem sind alle Songs der Playlist aufgelistet, welche ebenfalls Eigenschaften wie Name, Artist, Album oder Länge des Songs aufweisen.

Die Daten waren ursprünglich als JSON Files verfügbar. Insgesamt waren es 10'000 JSON-Files mit je 100 Playlists. In einem ersten Schritt wurden die JSON-Files zu einem grossen JSON-File umgewandelt, welches die gesamten Daten beinhaltet. Dies wurde mit dem „jq command-line JSON processor“ gemacht (<https://stedolan.github.io/jq/>).

Zu Beginn des Projektes wurde mit einem CSV-File gearbeitet. Die JSON-Daten wurden mit Hilfe des „jq“ Command Line Tools umgewandelt. Da die JSON Daten verschachtelt waren, verschoben sich die Eigenschaften von einem Song eine Ebene nach oben. Zudem wurden nur die ersten fünf Songs einer Playlist verwendet, da diese Datenmenge für die Demo-Applikation ausreichend war. Einige Eigenschaften der Playlist oder eines Songs konnten auch weggelassen werden, da nicht so viele Daten benötigt wurden.

```
{
  "name": "musical",
  "collaborative": "false",
  "pid": 5,
  "modified_at": 1493424000,
  "num_albums": 7,
  "num_tracks": 12,
  "num_followers": 1,
  "num_edits": 2,
  "duration_ms": 2657366,
  "num_artists": 6,
  "tracks": [
    {
      "pos": 0,
      "artist_name": "Degiheugi",
      "track_uri": "spotify:track:7vqa3sDmtEaVJ2gcvxtRID",
      "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
      "track_name": "Finalement",
      "album_uri": "spotify:album:2KRMJ9z7Xjoz1Az406UML",
      "duration_ms": 166264,
      "album_name": "Dancing Chords and Fireflies"
    },
    {
      "pos": 1,
      "artist_name": "Degiheugi",
      "track_uri": "spotify:track:23E0mJiv0Z88WJPUBIPjh6",
      "artist_uri": "spotify:artist:3V2paBXEoZIAhfZRJmo2jL",
      "track_name": "Betty",
      "album_uri": "spotify:album:3lUSlvjUoHNA8IkNTqURqd",
      "duration_ms": 235534,
      "album_name": "Endless Smile"
    }
    // 10 tracks more
  ]
}
```

Abbildung 14: Ausschnitt aus originalen JSON-Daten einer Playlist.

```
{
  "name": "musical",
  "collaborative": "false",
  "pid": 5,
  "modified_at": 1493424000,
  "num_albums": 7,
  "num_tracks": 12,
  "num_followers": 1,
  "num_edits": 2,
  "duration_ms": 2657366,
  "num_artists": 6,
  "track0_artist_name": "Degiheugi",
  "track0_track_name": "Finalement",
  "track0_duration_ms": 166264,
  "track0_album_name": "Dancing Chords and Fireflies",
  "track1_artist_name": "Degiheugi",
  "track1_track_name": "Betty",
  "track1_duration_ms": 235534,
  "track1_album_name": "Endless Smile"
  // 3 tracks more
}
```

Abbildung 15: Ausschnitt aus transformierten JSON-Daten einer Playlist.

Nach den ersten Versuchen mit dem CSV-File wurde festgestellt, dass sich für die Demo-Applikation besser eine Datenbank eignet, da bei Geschäfts-Applikationen fast immer eine Datenbank als Datenquelle gewählt wird.

In einem nächsten Schritt wurde umgestellt auf eine lokale Datenbank. Dafür wurde SQLite gewählt, welches zusammen mit der *Exposed* Library für Kotlin implementiert werden kann. Die Daten sind in einem SQLite-File gespeichert, welches in den Projekt Resources hinzugefügt werden kann. Durch den Einsatz von der Applikation „DB Browser for SQLite“, konnte das bereits erstellte CSV-File importiert und in eine SQLite Datenbank konvertiert werden. Mit dem Einsatz der Datenbank konnten auch die Datentypen der Spalten definiert werden, was später bei der Entwicklung des Filter-Mechanismus sehr entscheidend war.

Gegen Ende des Projekts hat sich herausgestellt, dass SQLite in Bezug auf das Sortieren und Filtern nicht ausreichend performant ist.

4 Lösungsaufbau

In den folgenden Unterkapiteln wird genauer auf den Aufbau der Lösung eingegangen. Dabei wird es darum gehen, welche Klassen wie verwendet wurden, wie diese zueinanderstehen und wie die Daten durch die Architektur fließen.

4.1 Software-Architektur

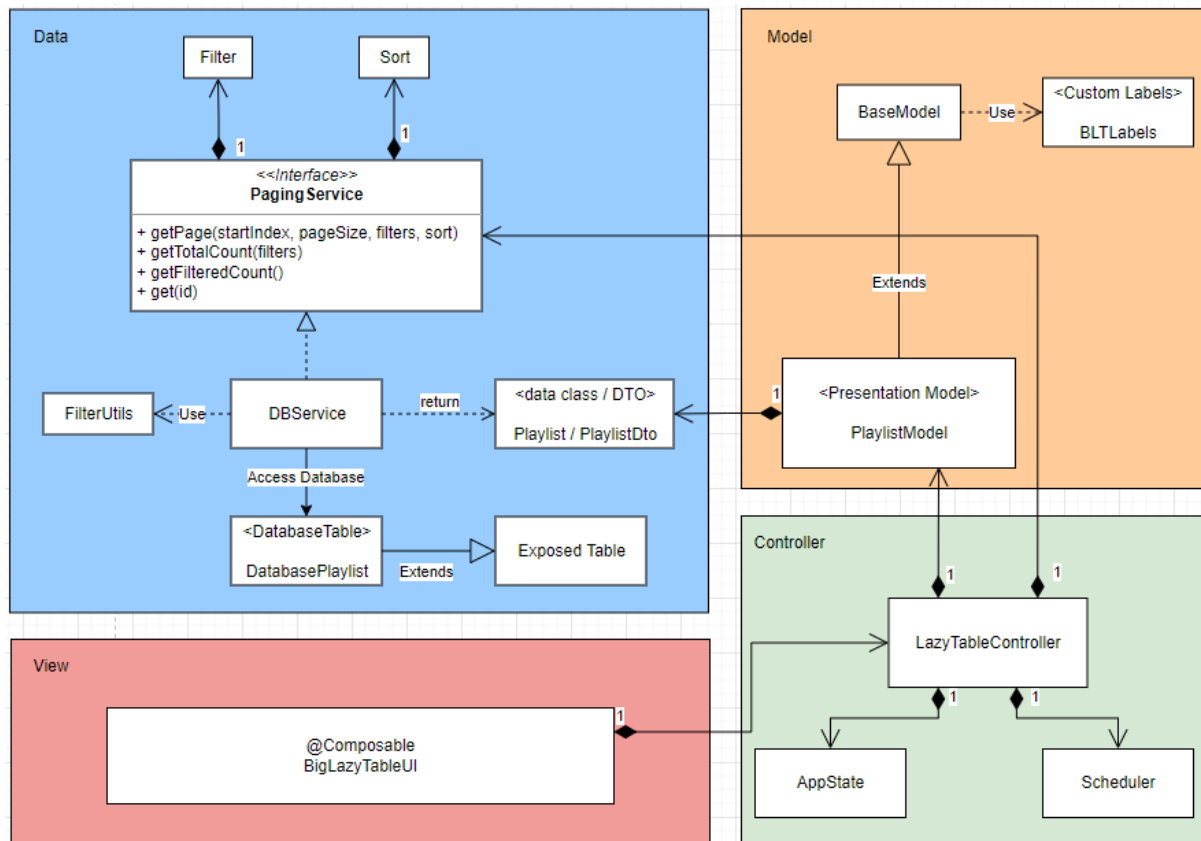


Abbildung 16: Eine Grobe Gesamtübersicht über Compose BigLazyTable.

Die Grundarchitektur besteht aus drei Layern: Data, Model und UI. Data verweist auf den Herkunftsort der Daten, die in der Tabelle dargestellt werden. Model verweist auf die zentralen Klassen, welche die Daten mit den entsprechenden Informationen darstellen. Im UI wird die Tabelle zusammengebaut und auf Interaktionen des End-Benutzer reagiert. Die Reaktion vollzieht sich dabei im Controller, der eng mit der Tabelle gekoppelt ist. Dieser Bereich wurde somit mit der MVC Architektur umgesetzt. Als Observables kommen die Compose States zum Einsatz; diese sind noch etwas mächtiger und Compose zeichnet bei Änderungen die entsprechend betroffenen UI Komponenten reaktiv neu. [10]

Zu Beginn war der Controller aus Gewohnheit von Android als ViewModel benannt; dies widersprach jedoch dem eigentlichen Verwendungszweck. Bei Android werden normalerweise bei Konfigurationsänderungen alle Daten anhand der `onCreate()` Funktion neu geladen. Das ViewModel hilft angezeigte Daten zu persistieren, so dass bei Konfigurationsänderungen immer noch die gleichen Daten dargestellt werden. Bei *Compose for Desktop* ist dies jedoch nicht nötig, da es keine `onCreate()` Funktion wie in Android und deshalb auch keinen Grund für ein ViewModel gibt.

4.1.1 Data

Im Data Layer sind verschiedene Klassen angesiedelt. Zuerst wird das PagingService Interface und die konkrete Implementierung des DBService genauer beschrieben. Dabei wird auch auf die DTOs genauer eingegangen, welche die Daten aus der Datenbank den Presentation Models zugänglich machen. Ebenfalls wird auf verwendete Hilfsfunktionen beim Filter-Mechanismus eingegangen und wie diese das Arbeiten mit dem DBService erleichtert haben und zukünftig erleichtern sollen. Für das Arbeiten mit SQLite wurde eine praktische Hilfswrapperklasse erstellt, welche das Initialisieren der Datenbankverbindung in sich verborgen hält.

4.1.1.1 PagingService

Das Interface wurde sehr früh definiert und diente als roter Faden für die Implementation eines konkreten Paging Service. Zu Beginn wies die getPage() und die getFilteredCount() Funktion einen String filter als Argument zum Angeben eines Filters auf. Dies wurde dann beim Hinzufügen der Filter zur Tabelle so erweitert, dass man eine Liste von Filtern angeben kann. Im Filter Objekt sind dann auch Einzelheiten, wie das zu filternde Datenbankfeld, der Filterwert entweder als String oder direkt als genauer Typ (Int, Float, usw.), from und to Werte bei den Nummern Filter und Case Sensitive bei den String Filtern vorhanden. Ebenfalls dazugekommen ist ein Sort Objekt, mit dem angegeben werden kann, welches Datenbankfeld mit welcher Sortierordnung sortiert werden soll. In [Abbildung 17](#) ist nun die aktuelle Version des Paging Service Interface sichtbar:

```
interface IPagingService<T> {  
  
    fun getPage(startIndex: Int, pageSize: Int, filters: List<Filter> = emptyList(), sort: Sort? = null): List<T>  
  
    fun getFilteredCount(filters: List<Filter>): Int  
  
    fun getTotalCount(): Int  
  
    fun get(id: Long): T  
  
    fun indexOf(id: Long, filter: String = ""): Int  
  
}
```

Abbildung 17: Das Paging Service Interface.

Der Typparameter T ist ein Platzhalter für das zu verwendende Data Transfer Object, in welches die Datenbankwerte hineingeschrieben werden. Mehr dazu folgt im Kapitel [Data Transfer Objects](#).

4.1.1.2 DBService

Nachdem ein genug grosser Datensatz gefunden wurde, war es an der Zeit diesen der Compose BigLazyTable Tabelle zur Verfügung zu stellen. Zu Beginn wurde anhand eines CSV Files der Daten ein CSVService geschrieben, welcher das CSV File einliest und die Werte in Data Transfer Objekte speichert. Problem dieses Ansatzes war, dass das Einlesen der Daten jedes Mal 20 Sekunden benötigte, bevor die Tabelle dann mit diesen Daten aufgestartet werden konnte. Ebenfalls erfolgte das Nachliefern von Daten sehr langsam und es war nicht möglich, innerhalb einer vertretbaren Zeit mit der Tabelle zu interagieren. Deshalb wurde entschieden,

für die Demo-Applikation mit einer relationalen Datenbank zu arbeiten, da so die Performance gesteigert werden und für das Filtern und Sortieren auf die bereits von SQL bestehenden Abfragemöglichkeiten zurückgegriffen werden kann. Auf einen Vergleich mit NoSQL Datenbanken wird hierbei verzichtet, da es für dieses Projekt und die Beantwortung der Fragestellung nicht von grösserem Wert ist. Theoretisch hätte auch eine NoSQL Datenbank verwendet werden können, aber basierend auf der Erfahrung der Teammitglieder wurde eine relationale Datenbank vorgezogen.

Um in Kotlin effektiv mit einer Datenbank zu arbeiten, wurde zuerst etwas Zeit in die Recherche darüber investiert, welche Methoden einen schnellen und einfachen Zugriff auf eine Datenbank ermöglichen.

Variante	Vorteile	Nachteile
Auf Datenbank anhand JDBC zugreifen. [11]	<ul style="list-style-type: none"> - Es ist genau bekannt, was nötig ist, um auf die Datenbank zuzugreifen. Viele Details sind somit bekannt, die damit verbunden sind. 	<ul style="list-style-type: none"> - Man muss viel Code selbst schreiben. - Es wäre für neue Datenbanken wieder der komplette Aufwand nötig, um auf diese zuzugreifen. Man wäre fest an eine gewisse Datenbank gebunden oder müsste nochmals Zeit investieren, um es für andere Datenbanken verfügbar zu machen.
Bewährte Java Lösung.	<ul style="list-style-type: none"> - Ist eine bewährte Lösung aus dem Java Umfeld. 	<ul style="list-style-type: none"> - Ist nicht in Kotlin geschrieben. - Für die Aussage, dass ein gewisser Approach besonders vertraut und daher schnell umsetzbar ist, fehlte die Erfahrung im Projektteam.
Exposed ORM Framework von JetBrains. [13]	<ul style="list-style-type: none"> - Ist in Kotlin geschrieben. - Mehrere Datenbanken sind unterstützt. - Die Konzentration auf die darunterliegenden JDBC Details ist nicht notwendig. - Wurde aufgrund der einfachen Anwendung von vielen Seiten und Personen empfohlen. - Es gab neben dem Exposed Wiki & Readme auch eine Hands-On Anleitung von der Website „Baeldung“. - Passte zum Gesamtprojekt, wo der Stand von Kotlin / Compose / usw. evaluiert werden soll. Zudem ist wie <i>Compose For Desktop</i> auch diese Library von JetBrains geschrieben. 	<ul style="list-style-type: none"> - Library ist eher neu und mit seiner aktueller Version 0.37 noch nicht bei einem stable Release angekommen.

	- Exposed kann verschiedene Datenbankengines nachahmen und hilft ohne feste Abhängigkeit auf eine spezifische Datenbankengine Datenbankapplikationen zu erstellen. Das Hin und Herwechseln zwischen verschiedenen Datenbankengines funktioniert mit minimen bis gar keinen Änderungen im Code.	
--	--	--

Tabelle 3: Varianten der Datenbankbindung in Kotlin.

In einem anderen Projekt hätte sich vermutlich eine bewährte, bereits aus dem Java Umfeld bekannte Lösung durchgesetzt. Aufgrund der Tatsache, dass das ganze Projekt auf Kotlin und Mechanismen beruht, die noch nicht stable sind und es zudem ein Ziel dieser Arbeit ist, den Reifegrad von Kotlin und Compose zu identifizieren, wurde entschieden auf die Exposed Library zu setzen. Obwohl diese von der Version nicht stable ist, wurde diese Entscheidung bis zum Ende des Projektes nicht bereut.

Um auf die Datenbank zuzugreifen, muss zuerst eine Klasse erstellt werden, welche die Daten der Datenbanktabelle erhält. Ebenfalls muss diese Klasse von der Exposed Table Klasse erben. In [Abbildung 18](#) ist ein Ausschnitt dieser Datenbanktabellenklasse zu sehen.

```
object DatabasePlaylists : Table() {
    val id = long( name: "id")
    val name = varchar( name: "name", length = 100)
    val modified_at = integer( name: "modified_at")
    val track0_artist_name = varchar( name: "track0_artist_name", length = 100)
    val collaborative = bool( name: "collaborative")
    val num_tracks = integer( name: "num_tracks")
    val num_albums = integer( name: "num_albums")
    val num_followers = integer( name: "num_followers")
}
```

Abbildung 18: Ausschnitt der verwendeten Exposed Tabelle.

Der Zugriff auf die Datenbank erfolgt dann entweder über eine typesafe SQL wrappende DSL oder anhand von leichtgewichtigen data access objects. Anhand der Hands-On Anleitung und des Readme war die typesafe SQL wrappende DSL sehr gut beschrieben, weshalb entschieden wurde diese zu verwenden. In [Abbildung 19](#) können einige Details dieser DSL herausgelesen werden. Der Zugriff auf die Datenbank erfolgt innerhalb eines transaction Blocks. Dort können dann auf der in [Abbildung 18](#) definierten Datenbankklasse anhand von Kotlin-Funktionen SQL Queries aufgerufen werden. Ein Beispiel wäre hier `selectAll()`, welches `SELECT *` darstellt. Das `count()` zählt schliesslich alle vorher selektierten Elemente. Das `toInt()` am Ende ist eine Funktion von der Kotlin Standardbibliothek, die ebenfalls auf die Exposed DSL Funktionen angewendet werden kann. [12]

```
override fun getTotalCount(): Int = transaction {
    DatabasePlaylists
        .selectAll() Query
        .count() Long
        .toInt()
}
```

Abbildung 19: Ausschnitt der „getTotalCount()“ Funktion in Exposed Kotlin SQL Syntax

Bei der `getPage()` Funktion konnte auf verschiedene praktische Exposed Funktionen zurückgegriffen werden. Die `selectWithAllFilters()` ist eine Hilfsfunktion, welche weiter unten im Kapitel [Filter Hilfsfunktionen](#) detaillierter beschrieben wird. Kurz zusammengefasst stellt sie ein Select Statement über alle Filterwerte dar. Die `orderBy()` Funktion ermöglicht das Sortieren anhand des mitgegebenen Sort Objekts. Limit ist sehr praktisch für das Paging, da die `pageSize` als Anzahl sowie der `startIndex` als Offset angegeben werden kann. Bei der `map()` Funktion zeigt sich ein weiterer Vorteil von Exposed. Da alle Datenbankabfragen mit Kotlin Code geschrieben werden können, ist es möglich mit Kotlin Funktionen wie `map()` auf die Werte zuzugreifen. Im Beispiel in [Abbildung 20](#) ist nach der Limit Funktion eine Liste von Exposed `ResultRow` verfügbar. Diese `ResultRow` wird dann dem `PlaylistDto` als `it` im Konstruktor mitgegeben. Das Mapping in die `Playlist` Klasse wird von `toPlaylist()` übernommen. Somit werden anstatt einer Liste von `ResultRow` eine Liste von `Playlist` Objekten zurückgegeben. Mehr dazu im Kapitel [Data Transfer Objects](#).

```
return transaction { this: Transaction
    DatabasePlaylists
        .selectWithAllFilters(filters)
        .orderBy( ...order: sort.dbField as Column<String> to sort.sortOrder)
        .limit(n = pageSize, offset = start)
        .map { PlaylistDto(it).toPlaylist() }
}
```

Abbildung 20: Ausschnitt aus der `getPage()` Funktion mit Filtern und Sortieren.

4.1.1.3 Filter Hilfsfunktionen

FilterUtil		
filterBetweenFromIncluded (Filter)		Op< Boolean >
caseSensitiveNotLike (Boolean, Column <String>, String)		Op< Boolean >
filterEquals (Filter)		Op< Boolean >
filterBetweenBothNotIncluded (Filter)		Op< Boolean >
caseSensitiveLike (Boolean, Column <String>, String)		Op< Boolean >
filterNotEquals (Filter)		Op< Boolean >
chooseCorrectFilterTypeMethod (MultipleOperationsFilter, FilterOperation)		Op< Boolean >
filterBetweenBothIncluded (Filter)		Op< Boolean >
filterBetweenToIncluded (Filter)		Op< Boolean >
filterLessEquals (Filter)		Op< Boolean >
filterGreater (Filter)		Op< Boolean >
filterLess (Filter)		Op< Boolean >
filterGreaterEquals (Filter)		Op< Boolean >
retrieveSql (Filter)		Op< Boolean >

Abbildung 21: Übersicht über alle Filter Hilfsfunktionen

Da das Ziel die Ermöglichung eines Filterns mit mehreren Filtern war, brauchte es einen Weg, um die von Exposed bereitgestellte Funktion `select { <query> }` für mehrere Filter gleichzeitig nutzen zu können. In anderen Sprachen wie zum Beispiel Java wäre dieses Problem nicht trivial, da `select` auf dem `DatabasePlaylist` Objekt aufgerufen wird. In Kotlin gibt es Extension Functions die sehr einfach erstellt werden und auf einer vorgegebenen Klasse aufgerufen werden können.

```
fun Table.selectWithAllFilters(filters: List<Filter>): Query {
    if (filters.isEmpty()) return Query( set: this, where: null)

    var sql = retrieveSql(filter = filters.first())
    for (i in 1 until filters.size) {
        val sql2 = retrieveSql(filter = filters[i])
        sql = sql and sql2
    }
    return this.select { sql }
}
```

Abbildung 22: Die `selectWithAllFilters()` Extension Funktion.

Anhand der [Abbildung 22](#) kann die Funktion auf allen Exposed Tables aufgerufen werden und stellt dadurch eine wichtige Schnittstelle dar, um von der Liste der Filter zu einem entsprechenden Select auf Datenbankebene zu gelangen. Für den Programmierer, der Compose BigLazyTable verwendet, kann somit die ganze Arbeit gespart werden, indem dieser einfach die bereitgestellte `selectWithAllFilters()` Funktion aufruft.

Die `retrieveSql()` Funktion greift beim angegebenen Filter auf die entsprechenden Felder zu, um daraus die Sql Abfrage abzuleiten.


```
fun retrieveSql(filter: Filter): Op<Boolean> = when (filter) {
    is BooleanFilter        -> filterEquals(filter)
    is MultipleOperationsFilter -> chooseCorrectFilterTypeMethod(filter = filter, filterType = filter.filterOperation)
}
```

Abbildung 23: Die retrieveSql() Funktion wählt je nach Filter Typ passende Funktion aus.

Die Fallunterscheidung anhand der verschiedenen zur Verfügung stehenden Filter Operationen wird in der untenstehenden Funktion durchgeführt. Je nach Filter Operation wird es zur dazugehörigen Filter-Funktion weitergeleitet.

```
private fun chooseCorrectFilterTypeMethod(filter: MultipleOperationsFilter, filterType: FilterOperation): Op<Boolean> {
    return when (filterType) {
        FilterOperation.EQUALS -> filterEquals(filter)
        FilterOperation.LESS   -> filterLess(filter)
        FilterOperation.GREATER -> filterGreater(filter)
        FilterOperation.NOT_EQUALS -> filterNotEquals(filter)
        FilterOperation.LESS_EQUALS -> filterLessEquals(filter)
        FilterOperation.GREATER_EQUALS -> filterGreaterEquals(filter)
        FilterOperation.BETWEEN_TO_INCLUDED -> filterBetweenToIncluded(filter)
        FilterOperation.BETWEEN_FROM_INCLUDED -> filterBetweenFromIncluded(filter)
        FilterOperation.BETWEEN_BOTH_INCLUDED -> filterBetweenBothIncluded(filter)
        FilterOperation.BETWEEN_BOTH_NOT_INCLUDED -> filterBetweenBothNotIncluded(filter)
    }
}
```

Abbildung 24: Die Unterscheidung zwischen den verschiedenen Filter Operatoren.

[Abbildung 25](#) zeigt ein Beispiel einer spezifischen Filter Funktion. Dabei wird zwischen den verschiedenen Filter Typen unterschieden und durch Kotlin Smart Casts (grün eingefärbt) wird der Filter als konkreter Filter eines gewissen Typs verwandelt, auf dem dann auch das dbField und der Filter den korrekten Typ aufweisen.

```
private fun filterEquals(filter: Filter): Op<Boolean> = when (filter) {
    is LongFilter    -> filter.dbField eq filter.filter
    is IntFilter     -> filter.dbField eq filter.filter
    is DoubleFilter  -> filter.dbField eq filter.filter
    is FloatFilter   -> filter.dbField eq filter.filter
    is ShortFilter   -> filter.dbField eq filter.filter
    is BooleanFilter -> filter.dbField as Column<String> like filter.filter.toString()
    is StringFilter  -> caseSensitiveLike(filter.caseSensitive, filter.dbField, filter.filter)
}
```

Abbildung 25: Die jeweiligen Funktionen zu den Filter Operatoren sind immer im gleichen Stil definiert

Die in [Abbildung 25](#) dargestellte Fallunterscheidung war nötig, da zuvor das dbField als Column<*> durchgereicht wurde und auf dieser untersten Ebene als Column<String> gecastet werden musste. Anhand dieses Casts war es daraufhin aber nicht mehr möglich, bei den Nummern Filtern nach grösser, kleiner usw. zu filtern. Nachdem die Datenbank schliesslich mit den korrekten Typen ausgestattet worden war (siehe Kapitel [Aufarbeitung der Daten](#)), funktionierte das Filtern auch mit Nummern-Filtern. Die korrekten Typen beschleunigten ebenfalls die Performance.

4.1.1.4 Data Transfer Objects

Wie der Name schon sagt, sind DTOs für das Transferieren der Daten zuständig. Auch hier konnte eine Kotlin Spracheigenheit zu Nutze gezogen werden. Anhand einer data class werden

nämlich getter, setter, hashCode und equals für den Programmierer definiert. Die nötigen Werte werdem dem Konstruktor der Data Class mitgegeben und schon ist der Entwickler fertig. In [Abbildung 26](#) ist ein Ausschnitt aus der in der Demo verwendeten Playlist Data Class sichtbar, welche zusätzlich für alle Konstruktor Argumente mit einem default Wert ausgestattet wurde. Dies brachte den Vorteil, dass der Klasse nicht alle Werte mitgegeben werden musste, sondern einfach ein leerer Konstruktor angegeben und die default Werte verwendet werden konnten. Falls Werte dem Konstruktor übergeben werden, werden die default Werte durch die mitgegebenen überschrieben. Die default Werte sind ebenfalls Placeholder fürs UI; wenn neue Daten geladen werden, werden diese Placeholder angezeigt. Das erstellte Playlist Objekt wird schliesslich dem Presentation Model weitergegeben, um die in der Tabelle und im Forms angezeigten Attribute mit den korrekten Werten zu befüllen.

```
const val loadingPlaceholderString = "..."  
const val loadingPlaceholderNumber = -999_999  
  
data class Playlist(  
    val id: Long = loadingPlaceholderNumber.toLong(),  
    val name: String = loadingPlaceholderString,  
    val collaborative: Boolean = false,  
    val modifiedAt: Int = loadingPlaceholderNumber,  
    val numTracks: Int = loadingPlaceholderNumber,
```

Abbildung 26: Ausschnitt aus der Playlist Klasse.

Wie in Kapitel [DBService](#) bereits erwähnt und in [Abbildung 20](#) angegeben, wurde bei den DB Service Funktionen am Ende das von Exposed zurückgegebene ResultRow dem PlaylistDto mitgegeben. Dieses ist in [Abbildung 27](#) dargestellt und verwendet genau ein Konstruktor Argument; ein ResultRow Objekt. Zusätzlich dazu bietet das PlaylistDto die Funktion toPlaylist() an, welches die ResultRow in ein Playlist Objekt abfüllt.

```
data class PlaylistDto(val resultRow: ResultRow) {  
  
    /**  
     * Helper function to map an Exposed [resultRow] into a Playlist  
     * @param resultRow the return type of a query from the Exposed framework  
     * @return a Playlist filled with all the needed attributes from the [resultRow]  
     */  
    fun toPlaylist(): Playlist = resultRow.let { it: ResultRow  
        Playlist(  
            it[DatabasePlaylists.id],  
            it[DatabasePlaylists.name],  
            it[DatabasePlaylists.collaborative],  
            it[DatabasePlaylists.modified_at],  
            it[DatabasePlaylists.num_tracks],  
            it[DatabasePlaylists.num_albums],  
            it[DatabasePlaylists.num_followers],
```

Abbildung 27: Ausschnitt aus der PlaylistDto Klasse.

4.1.1.5 SQLite

In diesem kurzen Kapitel wird die Hilfswrapperklasse SQLiteDatabase beschrieben. Wie in Kapitel [Filter Hilfsfunktionen](#) bereits besprochen, ist das LIKE von SQLite per default nicht case

sensitive. Die Lösung dieses Problems ist es, ein entsprechendes PRAGMA bei der Datenbank URL anzugeben. Grundsätzlich könnte das `case_sensitive_like=true` als hardcodierter Wert an die URL angehängt werden können, doch um ein einfaches Umschalten von case sensitivem LIKE zu unterstützen, Datenbank spezifische Details dem Programmierer abzunehmen und ebenfalls eine Unterstützung anzubieten um weitere PRAGMAS anzugeben, ist die Klasse in [Abbildung 28](#) entstanden. Das einzige Pflichtfeld ist der Pfad zur Datenbank. Case Sensitive ist per default eingeschaltet und die Liste der PRAGMAS ist standardmässig null. Die beiden handle Lambdas definieren anhand der mitgegebenen Werte, wie die URL zusammengebaut werden soll. Um die Datenbankverbindung zu initialisieren, wird auf der SQLiteDatabase Klasse die entsprechende Funktion aufgerufen.

```
class SQLiteDatabase(
    pathToDb: String,
    caseSensitiveFiltering: Boolean = true,
    listOfPragmas: List<String>? = null
) {
    private val makeSqliteCaseSensitive = "?case_sensitive_like=true"

    private val handleCaseSensitive = { caseSensitive: Boolean ->
        if (caseSensitive) makeSqliteCaseSensitive else ""
    }
    private val handlePragmas = { pragmas: List<String>? ->
        var params = ""
        pragmas?.onEach { param -> params += "&$param" }
        params ^lambda
    }

    private val url =
        "jdbc:sqlite:$pathToDb${handleCaseSensitive(caseSensitiveFiltering)}${handlePragmas(listOfPragmas)}"
    private val driver = "org.sqlite.JDBC"
    private val isolationLevel = Connection.TRANSACTION_SERIALIZABLE

    fun initializeConnection() {
        println(url)
        Database.connect(url = url, driver = driver)
        TransactionManager.manager.defaultIsolationLevel = isolationLevel
    }
}
```

Abbildung 28: Wrapper Klasse, um Details des Initialisierens der Datenbank vor dem Programmierer zu verbergen.

4.1.2 Model

In diesem Kapitel wird auf die einzelnen Klassen des Model Layers eingegangen. Zuerst wird beschrieben, wie ein Attribute Store gemieden werden konnte und anschliessend analysiert, wie das Presentation Model in die Tabelle integriert werden kann und welche Änderungen dafür nötig sind.

4.1.2.1 Kein Attribute Store

In der Grundidee wurde davon ausgegangen, dass für das Handling von verschiedenen kritischen Situationen, die aufgrund von der geteilten Verwendung desselben Models wie in *Compose Forms* und der dadurch gleichen darunterliegenden Attribute in der Tabelle auftreten können, einen Attribute Store benötigt werden würde. Ein solcher Attribute Store verwaltet zentral die Zustände der Attribute und sorgt dafür, dass diese Attribute immer synchronisiert sind. Probleme wie das Hin- und Wegscrollen, welches dazu führt dass das darunterliegende Attribut ausgetauscht wird und allfällige Änderungen somit verlorengehen, können mit dem Attribute Store verhindert werden. In vergangenen Projekten stellte eine solche Attribute Store Klasse stets die Lösung für Probleme dieser Art dar. In diesem Projekt wurde das Ziel ausgeschrieben, so lange ohne ein Attribute Store zu arbeiten, wie es möglich ist und ohne übertrieben komplizierten oder grossen Aufwand verwaltet werden kann. Dieses Ziel konnte erreicht werden. Grund dafür ist die von Compose gegebene Struktur mit den States. Stattdessen werden die bearbeiteten Attribute in einer Liste im AppState verwaltet und wenn neue Pages nachgeladen werden, kommt ein selbst entwickelter Merge Mechanismus (siehe Kapitel [LazyTableController](#)) zum Zug, welcher das neue Attribut mit dem bearbeiteten Attribut aus der Liste ersetzt. Somit werden keine Daten verworfen, sondern bewusst ausgetauscht.

4.1.2.2 Presentation Model

Das Grundprinzip, welches in Compose BigLazyTable verwendet wird, um auf zwei verschiedenen Ebenen das gleiche Presentation Model mit darunterliegenden Attributen zu teilen und diese miteinander synchronisiert zu halten ist definiert durch das Projektor Pattern. Weitere Informationen können hier dazu gefunden werden: <https://dierk.github.io/Home/projectorPattern/ProjectorPattern.html>.

In diesem Kapitel wird genauer auf das Presentation Model eingegangen. Es ist das Herzstück von *Compose Forms* und jedes Datensatzes in Compose BigLazyTable. Das Model wird somit zwischen den beiden Projekten geteilt. Änderungen im Forms an einem Datensatz sollen entsprechend in der Tabelle dargestellt werden. Die im Presentation Model enthaltenen Attribute speichern unter anderem, ob sie einen validen Wert enthalten. Diese Information soll ebenfalls in der Tabelle hervorgehoben werden. Eine weitere geteilte Information ist die angezeigte Sprache, welche sich auch auf die Tabelle weiterziehen soll. In [Abbildung 29](#) ist ein Ausschnitt des Presentation Models der Demo-Applikation zu sehen.

```
class PlaylistModel(playlist: Playlist) : BaseModel<BLTLabels>(title = BLTLabels.TITLE) {

    override val id = LongAttribute(
        model = this,
        label = BLTLabels.ID,
        value = playlist.id,
        readOnly = true,
        canBeFiltered = true,
        databaseField = DatabasePlaylists.id,
        tableColumnWidth = 100.dp
    )

    private val numEdits = IntegerAttribute(
        model = this,
        label = BLTLabels.NUM_EDITS,
        value = playlist.numEdits,
        canBeFiltered = true,
        databaseField = DatabasePlaylists.num_edits,
        tableColumnWidth = 200.dp
    )

    private val name = StringAttribute(
        model = this,
        label = BLTLabels.NAME,
        value = playlist.name,
        canBeFiltered = true,
        databaseField = DatabasePlaylists.name,
        tableColumnWidth = 200.dp
    )
}
```

Abbildung 29: Ausschnitt aus der PlaylistModel Klasse mit drei Attributen.

Dem Presentation Model wird ein Playlist DTO mitgegeben, welches vom PagingService weitergegeben wird. Dem geerbten BaseModel wird die Enum Klasse BLTLabels der definierten Labels mit den verschiedenen Sprachwerten mitgegeben und der Titel wird gesetzt. Die Attribute, welche im Forms angezeigt werden sollten, werden schliesslich wie in [Abbildung 29](#) angegeben. Das Label wird mit einem selbst definierten Wert aus den Labels gesetzt und der Value aus dem DTO herausgeholt. Als Erweiterungen für die Tabelle wurden die Attribute um die Werte canBeFiltered, databaseField und tableColumnWidth ergänzt. Damit kann angegeben werden, ob ein Feld gefiltert werden kann und falls ja, was das entsprechend zugehörige Datenbankfeld ist. Der letzte Wert dient der Anpassung der Tabellenspalte anhand des Inhalts. Wichtig zu wissen ist, dass, falls der angegebene Wert einen gesetzten Minimalwert unterschreitet, einfach der Minimalwert gesetzt wird. Grund dafür ist, dass falls die Tabellenspalte kleiner als dieser Minimalwert ist, die Filterfelder zerquetscht werden und es somit das Layout der Tabelle verwerfen würde.

In einer ersten Version musste dem Presentation Model ein AppState als Parameter mitgegeben werden, damit bei Änderungen auf einem Attribut dieses in die AppState Liste der geänderten Presentation Models hinzugefügt werden konnte, um den Merge Mechanismus zu ermöglichen. Diese eher unschöne Praktik konnte in einer neuen Version vermieden werden, indem der AppState in die BaseModel Klasse verschoben wurde und das Hinzufügen der geänderten Attributen direkt in der dortigen updateChanges Funktion passiert. Details der updateChanges Funktion sind in [Abbildung 30](#) dargestellt.

```
override fun updateChanges(){
    // Compose BigLazyTable
    if (appState?.changedTableModels?.contains(this) == false) {
        appState?.changedTableModels?.add(this)
    }
}
```

Abbildung 30: Die Update Changes Funktion im Base Model mit den Änderungen für den Merge Mechanismus.

Da normalerweise nicht alle Attribute des Formulars auch in der Tabelle dargestellt werden sollen, sondern nur die wichtigsten zur Übersicht, wurde dem BaseModel eine abstrakte Membervariable vom Typ Liste von Attributen hinzugefügt. Dort können alle definierten Attribute, die auch in der Tabelle angezeigt werden sollen, hinzugefügt werden.

```
override val displayedAttributesInTable = listOf(  
    id,  
    numEdits,  
    name,  
    modifiedAt,  
    track0ArtistName,  
    collaborative,  
    numTracks,  
    durationMs,  
    numFollowers,  
    numArtists  
)
```

Abbildung 31: Die Liste mit den Attributen, welche in der Tabelle angezeigt werden sollen.

4.1.3 Controller

In diesem Kapitel wird auf den Controller eingegangen, die zentrale Verwaltungsklasse für Compose BigLazyTable. Der Controller stellt das Herzstück der Tabellenimplementierung dar. Der Controller beinhaltet das Anfordern von neuen Pages, den Merge Mechanismus und das Verwalten der Filter und der Sortierung. Die vom Controller verwendeten Klassen wie der AppState oder der Scheduler sind ebenfalls in diesem Kapitel beschrieben.

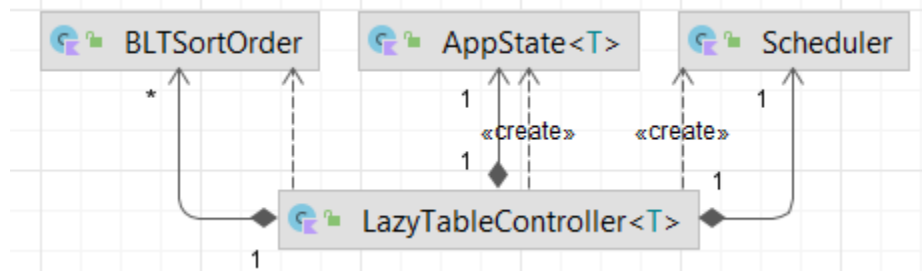


Abbildung 32: Übersicht über das Controller Package.

4.1.3.1 LazyTableController

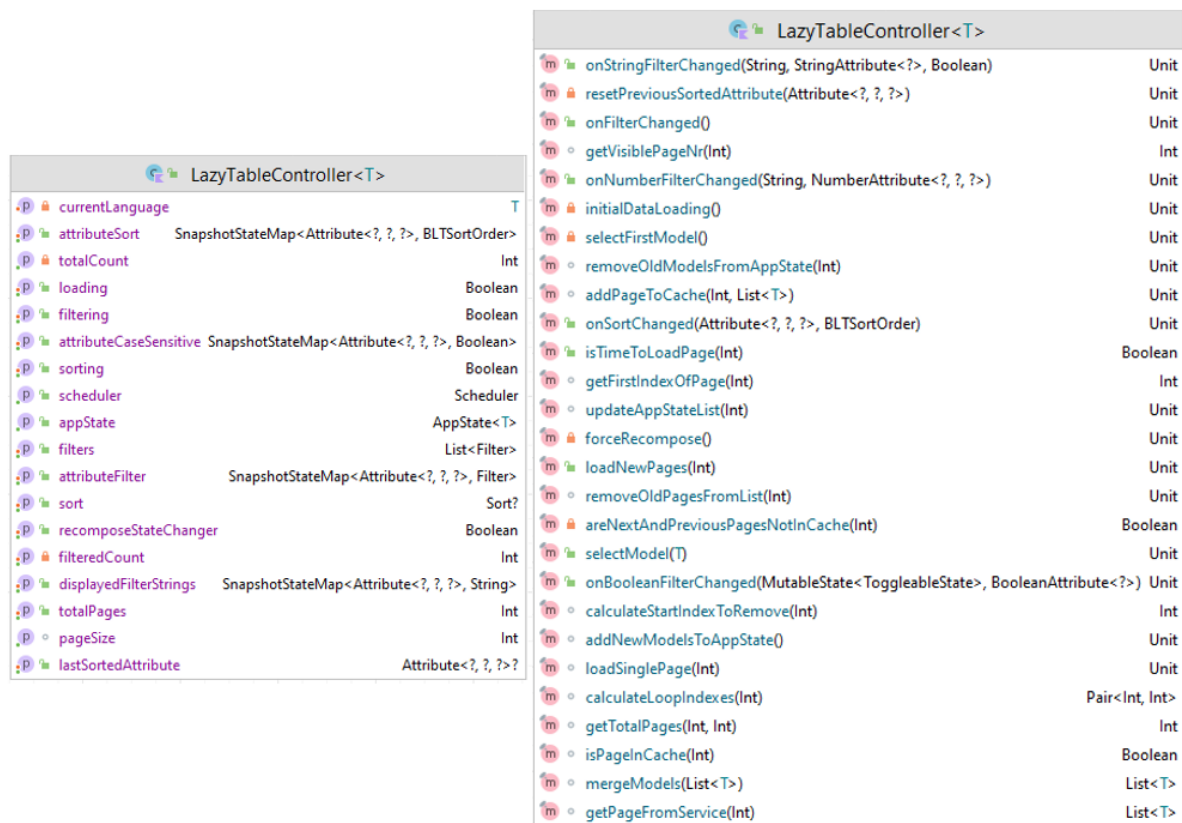


Abbildung 33: Der LazyTableController mit allen Membervariablen und Funktionen.

Aus Gewohnheit von Android war der Controller zuerst als ViewModel benannt. Wie in Kapitel [Software-Architektur](#) bereits beschrieben, ist dies bei *Compose for Desktop* aber gar nicht nötig. Zudem ist der Name irreführend, da sehr viel von dieser Klasse aus kontrolliert wird und es nicht nur darum geht die Daten, die angezeigt werden zur Verfügung zu stellen.

Schliesslich wurde mit einem Controller gearbeitet; genauer gesagt mit dem LazyTableController.

Um das Programm zu starten, wird dem Controller in der main-Funktion eine konkrete Implementierung eines Paging Service mitgegeben. Ebenfalls wird der Controller dem Eintrittspunkt der UI Implementierung mitgegeben, da fast das gesamte UI vom Controller abhängig ist und dementsprechend gezeichnet wird.

Im init-Block des Controllers werden die ersten Pages geladen, das erste Element ausgewählt und die benötigten Maps für das Filtern und Sortieren werden aufgesetzt. Der Controller besitzt intern einen LRU Cache (Last Recently Used) welcher eine Grösse von 5 hat und die geladenen Pages speichert. Beim initialen Datenladen wird dieser Cache mit den ersten 5 Pages gefüllt. Ebenfalls werden die neu geladenen Pages zum AppState hinzugefügt. Dazu mehr im Kapitel [AppState](#). Zu Beginn erfolgt das Laden der Daten innerhalb eines CoroutineScope gesetzt mit dem Main Dispatcher, das heisst, dass es auf dem Main Thread ausgeführt und normalerweise während einer laufenden Applikation dazu führen wird, dass die Anwendung nicht reagiert, bis die Daten geladen sind. Im Rest des Controllers werden neue Pages mithilfe des Scheduler angefordert. Der Scheduler ist dafür zuständig, dass immer der neuste Task im Hintergrund asynchron ausgeführt wird. Ein Task ist zum Beispiel das Laden von neuen Pages. Im Kapitel [Scheduler](#) wird genauer darauf eingegangen und weshalb es überhaupt einen braucht.

Doch woher die Gewissheit, wann neue Pages geladen werden sollen? Im Controller existiert genau dafür eine Funktion. Falls es an der Zeit ist, neue Pages zu laden wird der dafür verantwortlichen Funktion der firstVisibleItemIndex angegeben. Dieser Index ist anhand des ListStates, welcher von Compose zur Verfügung gestellt wird, erhältlich.

```
val firstVisibleItemIndex = verticallyLazyListState.firstVisibleItemIndex
if (isTimeToLoadPage(firstVisibleItemIndex)) {
    scheduler.scheduleTask { loadNewPages(firstVisibleItemIndex) }
}
```

Abbildung 34: Auslesen des firstVisibleItemIndex und Aufruf von loadNewPages auf Controller.

Der Cache verwaltet stets die aktuell sichtbare Page plus die vorherigen zwei und die nächsten zwei Pages. Falls an einen Ort gescrollt wird, wo noch keine Daten im Cache geladen sind, werden für den aktuellen Ort neue Pages geladen; die aktuelle und die letzten und nächsten zwei Pages. Ebenfalls werden im AppState die nicht mehr benötigten Pages entfernt und die neu im Cache gespeicherten Pages hinzugefügt.

Für die Problemfälle, die auftreten können, in denen Änderungen durch neue Pages überschrieben werden, wenn ein Datensatz bearbeitet, weggeschrollt und wieder zurückgeschrollt wird, wurde ein Merge Mechanismus geschrieben. Dieser prüft, ob Datensätzen Änderungen aufweisen. Diese Änderungen sind im AppState in der changedTableModels Liste zu finden. Die angeforderte Page wie auch die changedTableModels Liste besteht aus einer Liste von Presentation Models. Falls die changedTableModels Liste nicht leer ist, wird die angeforderte Page durchtraversiert und anhand der ID eines Presentation Model die Stelle der Änderung gefunden. An dieser Stelle wird das in der angeforderten Page neu geladene Presentation Model durch das bereits bearbeitete Model ersetzt.


```
internal fun mergeModels(pageOfModels: MutableList<T>, modelSaved: Boolean = false): MutableList<T> {
    for (i in firstPageNr ≤ until < pageSize) {
        // Check if a changed model is existing for the current iteration model
        val tableModel = appState.changedTableModels.find { tableModel ->
            tableModel.id.getValue() == pageOfModels[i].id.getValue()
        }

        // If tableModel is not null, the same model with changes exists
        if (tableModel != null) {
            // Replace the current model with the model containing changes
            pageOfModels[i] = tableModel
            // Remove the model with changes from the changedModels list
            if (modelSaved) {
                appState.changedTableModels.remove(tableModel)
            }
        }
    }

    return pageOfModels
}
```

Abbildung 35: Der Merge Mechanismus im Code.

Um die gleiche Sprache wie im Formular zu verwenden, existiert eine Funktion, die beim defaultModel die gesetzte Sprache abholt und anschliessend beim ausgewählten Model in der Tabelle diese als Sprache setzt. Somit müssen bei Sprachänderungen nicht alle Model überarbeitet werden, sondern jeweils nur das angezeigte. Der Rest ist nicht sichtbar und deshalb nicht von Bedeutung. In [Abbildung 36](#), [Abbildung 37](#) und [Abbildung 38](#) sind die Details im Code ersichtlich.

```
Row(
    modifier = Modifier.background(background-color).fillMaxWidth()
        .selectable(
            selected = isSelected,
            onClick = { controller.selectModel(playlistModel) }
        )
)
```

Abbildung 36: Ausschnitt einer Tabellen Zeile.

```
fun selectModel(tableModel: T) {
    setCurrentLanguage(tableModel = tableModel)
    appState.selectedTableModel = tableModel
}
```

Abbildung 37: jedes Mal wenn ein Model ausgewählt ist, wird die Sprache gesetzt.

```
private fun setCurrentLanguage(tableModel: T) {
    val currentLanguage = appState.defaultTableModel.getCurrentLanguage()
    tableModel.setCurrentLanguage(lang = currentLanguage)
}
```

Abbildung 38: Die Sprachänderung.

Ein notwendiges Übel, um bei der Tabelle einen *Recompose* zu erzwingen stellt die *forceRecompose* Funktion dar. Dabei wird einfach ein Boolean Wert negiert und in der Tabelle wird schliesslich auf diesen Boolean Wert zugegriffen. Da sich der Wert ändert, gibt es einen

Recompose vom Compose Framework und die neusten Werte werden dargestellt. Vorher musste immer noch ein kurzer Scroll durchgeführt werden, bevor die geladenen Daten auch angezeigt wurden. Ohne diese Scrollbewegung wurden nur die leeren Zeilen angezeigt und es wurde suggeriert, dass der Ladevorgang noch läuft, dabei waren die Daten schon längst geladen. Es fehlte jedoch eben dieser kleine *Recompose*, welcher zuvor mit der Scrollbewegung erzwungen wurde.

Im Controller sind ebenfalls die Funktionen enthalten, um auf geänderte Filter zu reagieren. Mehr dazu im Kapitel [Filtern](#).

Genau gleich wie bei den Filtern wird auch auf Änderungen beim Sortieren entsprechend reagiert. Mehr dazu im Kapitel [Sortieren](#).

4.1.3.2 AppState

Der AppState ist nur dazu da, den Tabellenzustand zu verwalten. [Abbildung 39](#) zeigt bereits die gesamte Klasse.

```
class AppState<T: BaseModel<*>>(pagingService: IPagingService<*>, defaultModel: T, pageSize: Int) {
    /** Default model used in form and table to store global data ...*/
    val defaultTableModel: T by mutableStateOf(defaultModel)

    /** Current selected model in table. ...*/
    var selectedTableModel: T by mutableStateOf(defaultTableModel)

    /** List of models. Size is the totalCount of the provided data. ...*/
    var tableModelList: MutableList<T?> = ArrayList(Collections.nCopies(pagingService.getTotalCount(), 0: null))

    /** List of filtered models. Size is the filteredCount of the provided data. ...*/
    var filteredTableModelList: MutableList<T?> = ArrayList(Collections.nCopies(pageSize, 0: null))

    /** List of all models containing changes. Used to prevent loosing changed data if new data is loaded from the service. ...*/
    val changedTableModels: MutableList<T> = mutableListOf()
}
```

Abbildung 39: Die Ganze AppState Klasse.

Das defaultTableModel wird benötigt, um Änderungen im *Compose Forms* der Tabelle zur Verfügung zu stellen. Dies ist zum Beispiel bei der Sprachänderung von Vorteil.

Die selectedTableModel Variable zeigt zu Beginn auf das defaultTableModel und nachher immer auf das aktuell ausgewählte Presentation Model.

Die tableModelList ist eine Liste, welche zu Beginn komplett mit null Werten befüllt ist. Die Grösse der Liste entspricht der Anzahl Datensätze, welche maximal vom Paging Service geliefert werden. Diese Liste wird im UI dynamisch gerendert und braucht die initiale Grösse, damit das UI den Scrollbalken entsprechend allen Daten darstellt. Die tableModelList wird zur Laufzeit mit den geladenen Pages befüllt und beim Weiterscrollen werden alte geladene Pages wieder auf null gesetzt.

Die filteredTableModelList hat den gleichen Nutzen wie die tableModelList mit dem Unterschied, dass sie sich auf die gefilterten Einträge bezieht. Durch die Trennung dieser beiden Listen kann schnell zwischen ihnen gewechselt werden was die User Experience verbessert.

Die `changedTableModels` verwaltet alle bearbeiteten Presentation Models und ermöglicht so den implementierten Merge Mechanismus.

4.1.3.3 Scheduler

Der Scheduler ist dafür zuständig, dass immer nur der neuste Task asynchron ausgeführt wird. Ein Task ist ein Aufruf, welcher zu längeren Wartezeiten führen kann. Der Scheduler ist wichtig, da beim schnellen Hin- und Herscrollen mit dem Scrollbalken sonst ständig neue Pages geladen werden. Anhand eines Delays kann dem Scheduler eine Zeitspanne angegeben werden, die gewartet wird, bevor ein Task in einem Coroutinen Block gestartet wird. In [Abbildung 40](#) sind die Details vom Code zu sehen.

```
class Scheduler(private val delayInMillis: Long = 50) {
    private var inProcess = false
    private var task: (() -> Unit)? = null
    private var taskToDo: (() -> Unit)? = null

    private fun process() {
        if (inProcess) return
        if (task == null) return
        inProcess = true
        taskToDo = task
        CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
            delay(delayInMillis)

            if (taskToDo == task) {
                taskToDo?.invoke()
            }
        }.invokeOnCompletion { it: Throwable?
            inProcess = false
            if (task == taskToDo) task = null
            process()
        }
    }

    fun scheduleTask(task: () -> Unit) {
        this.task = task
        process()
    }
}
```

Abbildung 40: Die Ganze Scheduler Klasse.

Obwohl es nur wenige Zeilen Code sind, steckt viel Arbeit dahinter und diese wenigen Zeilen sind von enormer Bedeutung für eine angenehm zu bedienende Tabelle. Das Scrollverhalten fühlt sich dank des Schedulers sehr viel näher an der eigentlichen Scrollbewegung an. Einen kleinen Verzug lässt sich bei genauem Hinsehen jedoch immernoch feststellen. Der Unterschied zu vorher, wo noch kein Scheduler existiert hat, ist aber gewaltig. Die Ladezeiten sind minimal geworden und bei schnellem Hin- und Herscrollen mit der Scrollbar werden keine Pages geladen. Erst wenn langsamer als der angegebene Delay gescrollt wird, werden Pages nachgeladen.

4.1.4 View

In diesem Kapitel wird auf die Implementierung der Benutzeroberfläche eingegangen und gewisse Eigenheiten von Compose hervorgehoben. Es folgt ein genauerer Blick hinter die Kulissen der Tabelle und wie diese konstruiert ist.

4.1.4.1 Aufbau

In einem Material Theme gehüllten Block wird als äusserste Schicht eine Row verwendet, welche den gesamten Platz einnehmen soll. Diese Row ist unterteilt in zwei ungleich grosse Hälften. Die kleinere linke Hälfte stellt die Tabelle dar und die grössere rechte Hälfte das Formular. Das Presentation Model wird dabei in der Tabelle als Zeile und im Formular dargestellt. Eingaben und Änderungen sind ohne Latenzen synchronisiert.

```
@Composable
fun BigLazyTableUI(controller: LazyTableController<*>) {
    BigLazyTableTheme {
        Row(modifier = Modifier.fillMaxSize()) { this: RowScope
            TableContainer(
                weight = 2f,
                controller = controller,
                appState = controller.appState
            )
            FormContainer(
                weight = 3f,
                model = controller.appState.selectedTableModel,
                appState = controller.appState
            )
        }
    }
}
```

Abbildung 41: Die Grundstruktur des BigLazyTable UI.

Im Table Container wird die Tabelle zuerst aus einer Box mit der Grösse der mitgegebenen Gewichtung (weight) aufgebaut. Die Gewichtung ist proportional zu anderen gewichteten Elementen der Row, also des Form Containers. Innerhalb der Box ist eine Column mit einem Header Element und der eigentlichen Tabelle. Das Header Element enthält innerhalb einer Row alle Filter Textfelder, die Überschriften der Tabelle und den jeweiligen Buttons zum Sortieren. Anhand der displayedAttributesInTable Liste aus dem BaseModel welches in der Demo vom PlaylistModel implementiert wird, werden die benötigten UI Elemente dynamisch erzeugt und dargestellt. Die zu Beginn genannte Box enthält ebenfalls eine horizontale Scrollbar, um die Gesamttabelle horizontal scrollbar zu machen, falls der Inhalt über den angezeigten Bereich hinausgeht.

Die eigentliche Tabelle greift auf ein Boolean Flag im Controller zu, das angibt ob neu geladen wird. Dies passiert jeweils beim initialen Datenladen im Controller. Wenn geladen wird, erscheint oben an den Tabellenzeilen ein Loading Spinner. Wenn Daten geladen sind, verschwindet der Loading Spinner wieder.

Die Tabellenzeilen sind wieder in einer Box verpackt. Diesmal ist die Box vertikal scrollbar. Innerhalb dieser Box werden anhand des ersten sichtbaren Index der vertikalen Liste neue Pages geladen. Anhand eines [Launched Effekt](#) Block wird auf Änderungen beim Filtern und Sortieren reagiert und als Reaktion darauf zum obersten Listenelement gescrollt.

Das Herzstück der Tabelle stellt die LazyColumn dar. Innerhalb der LazyColumn kann man anhand der items Funktion eine Liste mitgeben. Diese Liste ist entweder die gefilterte oder die normale Liste aus dem AppState. Da zu Beginn nicht alle Elemente der Listen geladen sind und der Rest mit null Werten befüllt ist findet eine Fallunterscheidung statt. Bei null Werten wird eine Platzhalter Tabellenzeile angezeigt und sonst die Tabellenzeile mit dem entsprechenden Presentation Model.

```
LazyColumn(
    modifier = Modifier.padding(end = ScrollbarThickness),
    verticalArrangement = Arrangement.spacedBy(4.dp),
    state = verticallyLazyListState
) { this: LazyListScope
    val lazyListItems = if (isFiltering) appState.filteredTableModellList else appState.tableModellList
    items(items = lazyListItems) { tableModel ->
        when (tableModel) {
            null -> TableRowPlaceholder(
                horizontalScrollState = horizontalScrollState,
                appState = appState
            )
            else -> TableRow(
                controller = controller,
                tableModel = tableModel as T,
                horizontalScrollState = horizontalScrollState,
                appState = appState
            )
        }
    }
}
```

Abbildung 42: Das Herzstück der Tabelle.

Unter der Haube ist wie der Name vermuten lässt eine Row zu finden. Zusätzlich kann diese angeklickt und dabei selektiert werden. Ein Klick auf eine solche Row ruft beim Controller die selectModel Funktion auf und gibt das darunterliegende tableModel mit. Die Attribute in der displayedAttributesInTable Liste werden anhand einer Tabellen Zelle dargestellt. Falls eine Platzhalter Zeile angezeigt wird, wird für alle anzuzeigenden Attribute ein ... dargestellt. Dies soll den Ladevorgang simulieren. Wenn es sich um keine Platzhalterzeile handelt, wird der Text, die Zellenbreite, die Textfarbe und Hintergrundfarbe, die Schriftgewichtung und allfällige Fehler anhand des Attributs ausgelesen und angezeigt.

Im Controller werden die geladenen Pages in einem Cache verwaltet. Dabei ist es notwendig, dass die im Cache enthaltenen Pages alle die gleiche Grösse haben. Wenn unterschiedliche Grössen enthalten wären, dann würde es zu Lücken in der Tabelle führen. Da sehr lange mit einem Defaultwert für die Page Grösse gearbeitet wurde und der Merge Mechanismus, das Abspeichern im Cache und das Hinzufügen zum AppState stark davon abhängig sind, war es ab einem bestimmten Punkt als zu spät angesehen worden auf eine dynamische Page Grösse zu wechseln. Zu viel bestehender und stabil laufender Code hätte überarbeitet werden müssen,

was schlicht nicht mehr möglich war. Im Kapitel [Mögliche Erweiterungen](#) wird erläutert, wie man von der festen Page Grösse wegkommt und was dabei zu beachten ist.

4.1.4.2 Filtern

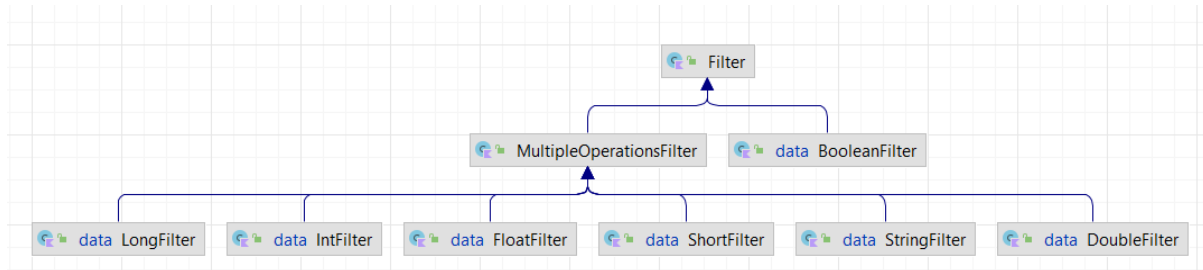


Abbildung 43: Der Grundaufbau der Filter Klassen

Das Implementieren der Filter baute wie schon das gesamte Projekt auf einem inkrementellen Ansatz auf. Anhand der noch verfügbaren Zeit wurde der Filter vertiefter angesehen und Schritt für Schritt weiter verfeinert bis zur aktuell finalen Version. Die verschiedenen Versionen vor der finalen Version sind im Anhang in Kapitel [Filter Versionen](#) genauer beschrieben. Die finale Version wird nachfolgend erläutert.

Zur Überprüfung wurde für das IntegerAttribute zuerst nur der IntFilter verwendet. Dieser Code allein ist schon ziemlich gross und kann noch weiter vereinfacht und etwas generalisiert werden. Schlussendlich muss nur überprüft werden, ob es sich um ein Nummer Attribut handelt und dann je nach Attribut der Filter mit dem passenden Typ erstellt werden. Das Parsen der NummerFilter in der Textfeld-onValueChanged-Funktion hat den UI Code in diesem File erheblich vergrössert und es gibt viele Duplikationen. Ebenfalls führt die onValueChanged Funktion eine Fallunterscheidung darüber durch, ob es sich um ein String- oder Nummern Attribute handelt. Ausserdem können Exceptions auftreten, wird ein ungültiger Wert in das Nummer Filter Textfeld eingegeben. Ein weiterer zu verbessernde Punkt ist, dass Nummern Textfelder und String Textfelder auf das gleiche Textfeld zugreifen. Die Nummern Textfelder verfügen über kein Leading Icon für Case Sensitive Filtern, weswegen dort auch kein Icon gesetzt wird. Dies führt jedoch dazu, dass der für ein solches Icon benötigte Platz dennoch vom Textfeld abgeschnitten wird und den Platz zur Eingabe erheblich verkleinert.

Um das letztgenannte Problem zu beseitigen und Separation of Concerns zu erreichen, wurden für Nummern und String Attribute eigene Textfelder verwendet: Das String Filter Textfeld mit einem Leading Icon fürs Case Sensitive Filtern und das Nummer Textfeld ohne ein solches. Somit wird auch keine Fallunterscheidung in den onValueChanged Funktionen mehr gemacht. Das Nummer Textfeld delegiert eine Wertänderung an den Controller anhand der onNumberFilterChanged() Funktion ([Abbildung 44](#)).

```
is NumberAttribute -> {
    TextField(
        modifier = Modifier.width(attribute.tableColumnWidth),
        value = controller.displayedFilterStrings[attribute].toString(),
        onChange = { newValue ->
            controller.onNumberFilterChanged(
                newValue = newValue,
                attribute = attribute
            )
        },
        textStyle = TextStyle(color = Color.White),
        label = { Text(text = FilterLabel, color = Color.White) },
        singleline = true,
        trailingIcon = { TrailingIcon(controller = controller, attribute = attribute) }
    )
}
```

Abbildung 44: Ausschnitt der Erkennung eines NumberAttribut und erstellen des entsprechenden Textfeldes. OnValueChanged wird an den Controller delegiert.

Die onNumberFilterChanged Funktion ruft eine entsprechende Funktion der Singleton Object Klasse NumberTextFieldUtil auf ([Abbildung 45](#)) und hat damit den Vorteil, dass der Code im Controller und im UI nicht zu gross wird. Die NumberTextFieldUtil Klasse könnte in zukünftigen Versionen um neue Implementierungen ergänzt werden. Im Controller kann man dann einfach zwischen verschiedenen Implementierungen wechseln.

```
fun onNumberFilterChanged(newValue: String, attribute: NumberAttribute<*, *, *>) {
    when (newValue) {
        "" -> {
            attributeFilter[attribute] = null
            displayedFilterStrings[attribute] = newValue
        }
        else -> NumberTextFieldUtil.createConcreteNumberFilter(
            newValue = newValue,
            controller = this,
            attribute = attribute
        )
    }
    onFilterChanged()
}
```

Abbildung 45: Im Controller werden die Änderungen an die NumberTextfieldUtil Klasse delegiert.

Beim Nummer Textfeld wurde der gesamte onChange Block mit try-catch umhüllt, um zu verhindern, dass die Tabelle wegen einer ungültigen Filtereingabe eine Exception wirft und das Programm beendet wird. Der Catch Block behandelt die entstandene Exception nicht, sondern ignoriert diese einfach. Für zukünftige Erweiterungen könnte man dem End-Benutzer der Tabelle anhand der Exception entsprechende Fehlermeldungen anzeigen. Mehr dazu im Kapitel [Mögliche Erweiterungen](#).

In diesem letzten Abschnitt wird abschliessend noch auf die definierte Filtersyntax eingegangen. Boolean Attribute werden anhand einer Checkbox mit drei Zuständen gefiltert. Bei String Attributen kann man wie man es sich von LIKE bei Datenbankabfragen gewohnt ist filtern. Zusätzlich ist auch eine NOT LIKE Filterung möglich, wenn man ein Ausrufezeichen (!) vor den zu filternden Wert setzt. Bei Nummern Attributen muss vor die Zahl die entsprechende Filter Operation (=, !=, <, <=, >, >=) gesetzt werden. Nur anhand einer eingegebenen Zahl passiert noch nichts, da zusätzlich noch mitgeteilt werden muss wie gefiltert werden soll. Um Nummern Attribute anhand eines Zahlenbereichs zu filtern, müssen der Start- und Endwert innerhalb von eckigen Klammern und mit Komma getrennt eingegeben werden. Bei einer Eingabe von [0,10] werden nur Zahlen von 0 bis 10 angezeigt. Zusätzlich können auch Zahlenbereiche angegeben werden, wo gesagt wird, dass ein Start- oder Endwert oder beide nicht zu den Resultaten dazugehören sollen. Bei einer Eingabe von]0,10[werden nur Zahlen von 1 bis 9 angezeigt, da 0 und 10 vom gesuchten Zahlenbereich ausgeschlossen sind.

4.1.4.3 Sortieren

Sortieren baut auf keiner spezifischen Liste vom AppState auf. Es kann auf der normalen und der gefilterten Liste sortiert werden.

Das Sortieren enthält drei Zustände und wurde in Kotlin mit einer Sealed Class (<https://kotlinlang.org/docs/sealed-classes.html>) umgesetzt. Alle Zustände, Texte, Icons und Funktionen der einzelnen Sortier Typen sind ausserdem dort festgehalten. Es ist immer nur möglich ein Attribut auf einmal zu sortieren. Falls mehr als die drei bereits definierten Zustände vorliegen sollen, können einfach neue hinzugefügt werden, ohne an der aktuellen Struktur etwas ändern zu müssen. Das Einzige, was getan werden muss ist die Reihenfolge der Bestehenden abzuändern. Die Sealed Klasse und ihre Unterklassen sind jedoch sehr übersichtlich und das Hinzufügen neuer Umstände sollt ohne grosse Mühe möglich sein.

Die Sealed Klasse ist so aufgebaut, dass man die Sortier Ordnung, ein Flag ob sortiert und das dazugehörige Icon bereits im Konstruktor angeben kann. Durch die Idee des Hin- und Herswitchen zwischen den drei Sortierzuständen wird stets ein Verweis vom aktuellen sortOrder Objekt auf das nächstfolgende und dessen Icon benötigt. Eine Idee bestand darin, den Verweis auf den nächsten BLTSortOrder auch im Konstruktor anzugeben, was jedoch zu einer NullPointerException führte. Das Problem dabei war, dass es eine Warnung gab, wenn man den Typ als Non nullable markiert hatte: „Constructor has non-null self reference parameter“. Die Vermutung liegt nahe, dass es nicht möglich ist, einen Parameter anzugeben, der dem gleichen Typ angehört wie die Klasse, in der der Parameter mitgegeben wird. In diesem Fall erreicht der Wert null und es wird eine NullPointerException geworfen. Um dieses Problem zu umgehen, wurden 2 Varianten geprüft: Zum einen der sealed Klasse eine abstrakte Membervariable mitzugeben oder ihr eine Funktion mitzugeben, die auf den nächsten SortState zeigt. Beide Varianten funktionieren. Gewählt wurde jedoch diejenige mit der Membervariable, da die Definition einer Variable für einen gleichbleibenden Wert sinnvoller schien als die einer Funktion. Beim Membervariablen Approach kann entweder der Wert anhand eines get() = Wert gesetzt werden oder direkt auf der Variable. Beide funktionieren und deshalb wurde entschieden, es so einfach wie nur möglich zu halten und die direkte Zuweisung der get Zuweisung vorzuziehen. Die Reihenfolge wurde so bestimmt,

dass auf keine Sortierung Ascending Sortierung und dann Descending Sortierung folgt. Danach beginnt der Zyklus wieder von vorne.

```
sealed class BLTSortOrder(
    var sortOrder: SortOrder?,
    var isSorting: Boolean,
    var icon: ImageVector
) {
    abstract val nextSortState: BLTSortOrder
    abstract val nextSortIcon: ImageVector
    abstract fun sortAttribute(attribute: Attribute<*, *, *>): Sort?

    object Asc : BLTSortOrder(sortOrder = SortOrder.ASC, isSorting = true, icon = Icons.Default.KeyboardArrowUp) {
        override val nextSortState: BLTSortOrder = Desc
        override val nextSortIcon: ImageVector = nextSortState.icon
        override fun sortAttribute(attribute: Attribute<*, *, *>): Sort = Sort(
            dbField = attribute.databaseField,
            sortOrder = sortOrder!!
        )
    }

    object Desc : BLTSortOrder(sortOrder = SortOrder.DESC, isSorting = true, icon = Icons.Default.KeyboardArrowDown) {
        override val nextSortState: BLTSortOrder = None
        override val nextSortIcon: ImageVector = nextSortState.icon
        override fun sortAttribute(attribute: Attribute<*, *, *>): Sort = Sort(
            dbField = attribute.databaseField,
            sortOrder = sortOrder!!
        )
    }

    object None : BLTSortOrder(sortOrder = null, isSorting = false, icon = Icons.Default.Close) {
        override val nextSortState: BLTSortOrder = Asc
        override val nextSortIcon: ImageVector = nextSortState.icon
        override fun sortAttribute(attribute: Attribute<*, *, *>): Sort? = null
    }
}
```

Abbildung 46: Die gesamte BLTSortOrder Klasse.

Die BLTSortOrder Sealed Klasse wird im Controller und ebenfalls im UI von der erstellten Composable Function Sort Button (IconButton) verwendet. Im UI wird der nextSortState und das Icon dazu benötigt, um das UI konsistent mit der Sortierung zu halten. Bei einem Klick auf den Sort Button wird im Controller onSortChanged mit nextSortState aufgerufen.

```
@Composable
fun SortButton(
    modifier: Modifier,
    attribute: Attribute<*, *, *>,
    controller: LazyTableController<*>
) {
    val sortOrder = controller.attributeSort[attribute]
    IconButton(
        modifier = modifier,
        onClick = {
            controller.onSortChanged(
                attribute = attribute,
                newSortOrder = sortOrder?.nextSortState ?: BLTSortOrder.None
            )
        }
    ) {
        Icon(
            imageVector = sortOrder?.nextSortIcon ?: BLTSortOrder.None.icon,
            contentDescription = ContentDescriptionSortIcon,
            tint = Color.White
        )
    }
}
```

Abbildung 47: Der Sortier Button ist ein Icon Button und wird wie gezeigt aufgebaut.

Im Controller werden zuerst bereits bestehende Sortierungen auf anderen Attributen auf den Standardzustand „unsortiert“ zurückgesetzt (weil wie schon erwähnt immer nur ein Attribut auf einmal sortiert wird). Danach wird eine Referenz auf das aktuelle Attribut als `lastSortedAttribute` gespeichert, um bei einem Sortieren auf einem anderen Attribut dieses zurückzusetzen. Der `attributeSort` Map wird das Attribut als Key und die neue `SortOrder` als Value mitgegeben. Diese Map verwaltet die Zustände (`BLTSortOrder`) der verschiedenen Attribute. Ausserdem befindet sich deshalb im UI eine Referenz auf die `BLTSortOrder` und das UI weiss welcher `SortState` der aktuelle und welcher der nächste ist. Diese Map würde zudem auch den Grundstein legen, um mehrere Attribute gleichzeitig zu sortieren, falls dies in Zukunft noch hinzugefügt werden möchte. Im Controller wird ebenfalls das Sort Objekt gesetzt, anhand dessen die Daten anders geladen werden.

```
fun onSortChanged(attribute: Attribute<*, *, *>, newSortOrder: BLTSortOrder) {
    resetPreviousSortedAttribute(newAttribute = attribute)

    lastSortedAttribute = attribute
    attributeSort[attribute] = newSortOrder
    isSorting = newSortOrder.isSorting
    sort = newSortOrder.sortAttribute(attribute)

    scheduler.scheduleTask { initialDataLoading() }
}
```

Abbildung 48: Ausschnitt aus der `onSortChanged` Funktion im Controller.

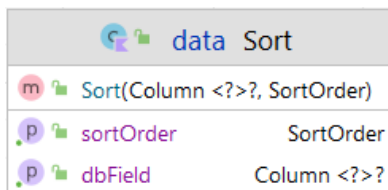


Abbildung 49: Ausschnitt des Sort Objekts.

Anschliessend werden anhand des gesetzten Sort Objekt die Daten vom Paging Service geladen. In der `getPage` Funktion existiert eine Unterscheidung, ob eine Sortierung der Daten vorliegen soll oder nicht. Falls ja wird anhand der `Exposed` Funktion `orderBy` das `dbField` des Sort Objekts und dessen `sortOrder` angegeben.

4.2 Laufzeit-Messungen

In diesem Kapitel geht es um die Laufzeit der Tabelle anhand verschiedener Szenarien. Die untersuchten Szenarien werden in den weiteren Abschnitten genauer erläutert. Um die Messungen schnell und unkompliziert durchzuführen, wurde entschieden an den zu messenden Stellen die Start- und Endzeit anhand `System.currentTimeMillis` auszulesen und die Differenz der beiden mit einem Konsolenlog auszugeben.

Das erste untersuchte Szenario, war das Aufstarten der Tabellenapplikation und das damit verbundene erste initiale Laden der Daten. Die gesamte Ladezeit liegt zwischen 350 bis 450 Millisekunden. Eine einzelne Page anzufordern dauert dabei zwischen 10 bis 50 Millisekunden.

Im zweiten Szenario wurde die Laufzeit beim Filtern genauer untersucht. Je nach gefilterter Spalte und der Anzahl der zu filternden Datensätze variiert die Gesamtladezeit zwischen 20 und 2'500 Millisekunden ziemlich stark. Eine einzelne Page anzufordern, kann dementsprechend ebenfalls zwischen 5 und 500 Millisekunden variieren. Das Hinzufügen und Entfernen von Filtern dauert ungefähr gleich lange.

Im dritten Szenario wurde das Sortieren untersucht. Wie schon beim vorherigen Szenario, können sich die Werte je nach Spalte und Anzahl der zu filternden Ergebnissen stark unterscheiden. Die Gesamtladezeiten liegen hier zwischen 20 und 3'500 Millisekunden. Das Laden einer einzelnen Page dauert zwischen 5 und 1'000 Millisekunden. Dabei ist zu beachten, dass die erste Spalte mit den ID Werten, welche in der Datenbank als Primary Key hinterlegt ist, im Gegensatz zu den anderen Spalten sehr schnell sortiert werden kann. Das Aufheben des Sortierens dauert in der Regel um die 50 Millisekunden.

Im letzten Szenario wurde das Zusammenspiel von mehreren Filtern und einer sortierten Spalte angesehen. Dabei ist anzumerken, dass sich die Laufzeit genau gleich verhält wie in den vorherigen einzelnen Filtern beziehungsweise Sortier Szenarien. Die Ladezeit wird immer für jeden neuen oder gelöschten Filter oder jede neue oder gelöschte sortierte Spalte ausgeführt. Es wurden keine grösseren Ladezeiten für mehrere Filter und gleichzeitiger Sortierung festgestellt.

Als Abschluss dieses Kapitels ist anzumerken, dass die entstandenen Ladezeiten beim Zugriff auf die Datenbank entstanden sind. Die Ladezeiten in Controller Funktionen, welche den Cache und den AppState verwalten, sind dabei von sehr kleinem Ausmass - sie bewegen sich bei diesen Funktionen stets im Bereich von 0 bis 5 Millisekunden. Ebenfalls ist anzumerken, dass das Mapping von Exposed ResultRows in Playlist Objekte fast immer bei 0 Millisekunden bleibt und sehr selten auf maximal 3 Millisekunden ansteigen kann.

4.3 Software-Testing

In diesem Kapitel wird beschrieben, wie beim Testen der Software vorgegangen wurde und welche Details an welchen Orten genauer betrachten wurden. Ebenfalls wird auf die jeweiligen Vorteile der verschiedenen Tests eingegangen. Zum Schluss dieses Kapitels wird eine Übersicht über die Test Code Coverage gegeben.

4.3.1 Logging

Da die Tests mit Gradle ausgeführt werden, wurde immer nur angezeigt, ob alle Tests durchgelaufen sind, ohne dabei anzuzeigen welche Tests genau darunter waren. Wenn ein Test fehlschlug, wurde nur dieser markiert und die durchgelaufenen Tests waren ebenfalls nicht sichtbar. Dadurch war es schwer nachzuvollziehen, was nun genau gemacht worden war. Die fehlende Sichtbarkeit förderte eine gewisse Unsicherheit, ob der Test überhaupt etwas machte. Tests sollten aber genau das Gegenteil bewirken. Sie sollten dem Entwickler die Sicherheit geben, dass die Funktionalität korrekt umgesetzt wurde und es so funktioniert, wie es sollte. Deswegen wurde entschieden in den Tests Logs auszugeben, um ein gewisses Mass an Informationen darüber bereitzustellen, was überhaupt passiert. So kann bei fehlschlagenden Tests schneller herausgefunden werden, weshalb der Tests nicht korrekt durchgelaufen ist. Um das Problem, dass ungewiss war, welcher Test ausgeführt wurde, zu lösen, gab es verschiedene Varianten:

Variante	Vorteile	Nachteile
Println des hardcodierten Testnamens am Anfang jedes Tests.	<ul style="list-style-type: none"> - Es wird im Log ausgegeben, welcher Test gerade ausgeführt wird. 	<ul style="list-style-type: none"> - Für jede Funktion muss von Hand ein println eingefügt werden. - Hardcodierte Strings: Jedesmal wenn sich ein Testname ändert, müsste auch der im println ausgegebene String geändert werden. - Sehr undynamische Lösung mit viel unnötiger, manueller Handarbeit.
Println mit <i>object</i> <code>{}.javaClass.enclosingMethod.name</code> als String. <pre>/** * Pass 'object {}.javaClass.enclosingMethod.name' * as [testMethodName] to get the test method name */ fun printTestMethodName(testMethodName: String) = println("----- ### \$testMethodName ### -----")</pre> <i>Abbildung 50: Ausschnitt einer möglichen Funktion</i>	<ul style="list-style-type: none"> - Es ist im Log sichtbar, welcher Test gerade ausgeführt wird. - Es wird kein hardcodierter String mitgegeben, sondern der Wert ändert sich dynamisch mit dem Testname. 	<ul style="list-style-type: none"> - Für jede Funktion muss von Hand ein println eingefügt werden. - Es muss bekannt sein, was dem println mitgegeben werden muss. Immer mühsames

		<p>Nachschauen, was nun wieder mitgegeben werden muss.</p> <ul style="list-style-type: none"> - Vergrössert Aufwand beim Schreiben eines Tests und führt dazu, dass weniger Tests geschrieben werden.
<p>Gradle Test Logging Events.</p> <pre>tasks.apply { this: TaskContainer withType<Test> { this: Test useJUnitPlatform() testLogging { this: TestLoggingContainer events("passed", "skipped", "failed") } } withType<KotlinCompile> { kotlinOptions.jvmTarget = "11" } }</pre> <p>Abbildung 51: Einschalten der Gradle Test Logging Events</p>	<ul style="list-style-type: none"> - Es ist im Log sichtbar, welche Tests ausgeführt worden sind und ob diese „Passed“, „Skipped“ oder „Failed“ sind. 	<ul style="list-style-type: none"> - Die Gradle Test Logging Events werden erst am Schluss aller Tests geloggt. Somit sind die spezifischen Log Statements innerhalb einer Testfunktion nicht zuordbar und bringen daher wenig.

Tabelle 4: Varianten von Test Logging.

Anhand der evaluierten Varianten wurde entschieden, eine Mischung aus der zweiten und letzten Variante zu verwenden. Werden in einer Testfunktion Logs ausgegeben, ist es praktisch zu wissen, zu welcher Testfunktion die Logs gehören. Dort wird dann zuoberst in der Funktion `printTestMethodName(object {}.javaClass.enclosingMethod.name)` angegeben. Die Gradle Test Logging Events sind immer eingeschaltet und runden die Test Logs mit einer Zusammenfassung ab.

Der letzte Nachteil der `printTestMethodName` Variante hört sich schrecklich an: Tests sind wichtig und es sollten so viele wie möglich geschrieben werden! Um diesen Nachteil zu umgehen, gibt es in IntelliJ ein effizientes Feature, welches sich Live Templates nennt. Dies ist zum Beispiel vom Erstellen von Composable Functions bekannt, wo nur „comp“ eingegeben werden muss und anschliessend der gesamte Boiler Plate Code erstellt wird. Gleiches kann bei den Tests zu Hilfe gezogen werden. Wird unter `File > Settings „Live template“` ins Textfeld eingegeben, gelangt man zur Live Templates Übersicht. Dort kann dann die gewollte Sprache (Kotlin) angewählt werden, in dem rechts das Plus Icon angeklickt und Live Template angewählt wird. Es öffnet sich ein Editor und folgendes wird eingegeben:

The screenshot shows the 'test (Create a JUnit Test Case)' live template configuration in IntelliJ. It includes fields for 'Abbreviation' (test) and 'Description' (Create a JUnit Test Case). The 'Template text' field contains the following Kotlin code:

```
@Test
fun $NAME$() {
    printTestMethodName(object {}.javaClass.enclosingMethod.name)
    $END$
}
```

On the right, the 'Options' section shows 'Expand with' set to 'Default (Tab)', and checkboxes for 'Reformat according to style', 'Use static import if possible', and 'Shorten EQ names' (which is checked).

Abbildung 52: Erstelltes Test Live Template.

Wichtig ist es, bei Edit variables NAME als methodName() Expression einzustellen. Ebenfalls muss unten links definiert werden, wo das Live Template zum Einsatz kommen kann. Nachdem ein Test eingegeben wurde, kann gespeichert werden und nach anschliessendem Enter wird der gesamte Test Boilerplate Code erstellt mit dem printTestMethodName bereits darin enthalten. Live Templates können für alles Mögliche erstellt werden und können viel Zeit sparen.

4.3.2 Tests

In diesem Unterkapitel wird auf die einzelnen Tests eingegangen sowie auf die Details, die es dabei zu beachten galt. Dabei werden die diversen Varianten der Ausführung sowie die daraus resultierenden Entscheidungen aufgeführt.

4.3.2.1 DB Service Tests

Da das Paging Service Interface ziemlich zu Beginn definiert wurde und wenig später mit dem DB Service eine konkrete Implementierung dazu kam, war dies die erste zu testende Klasse. Zu Beginn wurde die «produktive» Datenbank mit den ein Millionen Datensätzen zum Testen verwendet. Dies half dabei, ein besseres Verständnis über alle Daten zu erlangen. Als eine erste Version der Tests auf Git gepusht worden war, schlugen die Tests fehl, da das Datenbank File nicht im Git Repo enthalten war.

Es gab drei verschiedene Varianten, auf welchen Daten der DB Service Test beruhen sollte:

Variante	Vorteile	Nachteile
„Produktive“ Datenbank.	<ul style="list-style-type: none"> - Eine Möglichkeit, die produktive Datenbank besser kennenzulernen. - Es wird mit denselben Werten wie in der Demo gearbeitet und Probleme können besser reproduziert werden. - Lokal schlagen die Tests nicht fehl, wenn das Datenbank File lokal gespeichert und darauf in den Tests referenziert wurde. 	<ul style="list-style-type: none"> - Die Grösse übersteigt die maximal mögliche Grösse, um die Datenbank im Git Repository abzulegen. Bevor die Tests ausgeführt werden können, muss manuell die Datenbank hinzugefügt werden. - Da mit GitHub Actions gearbeitet wurde und jedes Mal bei einem Push direkt die Pipeline durchgelaufen ist, existierte die Datenbank nicht und die Tests schlugen fehl. - Wäre mit einem grossen Workaround vielleicht schon möglich gewesen, dem GitHub Runner das Datenbank File standardmässig anzugeben, hätte jedoch eine unsaubere Lösung zur Folge gehabt und wäre daher unnötig investierte Zeit gewesen.

Datenbank mocken.	<ul style="list-style-type: none"> - Die Werte, die verwendet werden sollen, könne selbst bestimmt werden. - Es benötigt kein Datenbank File, um Tests laufen zu lassen. 	<ul style="list-style-type: none"> - Es wird mit selbst mitgegebenen Werten gearbeitet und Edge Cases, die mit der echten Datenbank vorkommen könnten, können übersehen werden. - Hätte Einarbeitungszeit gebraucht, um sich mit einem Mocking-Framework auseinanderzusetzen (beide Teammitglieder hatten nur wenig Erfahrung damit) und es bestand keine Dringlichkeit, unbedingt die Datenbank zu mocken. Unter Betrachtung des Zeitaufwandes, den diese Lösung beansprucht hätte, wies dies keinen grossen Mehrwert auf.
„Produktive“ Datenbank auf Werte von 0 bis 1'000 verkleinern.	<ul style="list-style-type: none"> - Eine Möglichkeit, die produktiven Daten besser kennenzulernen. - Es wird mit denselben Werten wie in der Demo gearbeitet und Probleme können besser reproduziert werden. - Durch die Verkleinerung konnte das Datenbank File ins Git Repository eingchecked werden. - Lokal und auf GitHub laufen somit die Tests ohne Probleme. - Geringster Zeitaufwand mit dem grössten Mehrwert. 	<ul style="list-style-type: none"> - Nutzung von nur 1'001 Daten von insgesamt einer Million Datensätze. - Edge Cases könnten in den restlichen 999'000 Datensätzen verborgen sein.

Tabelle 5: Varianten der Testdatenbank.

Nachdem die verschiedenen Varianten geprüft wurden, wurde entschieden die „Produktive“ Datenbank auf Werte von 0 bis 1'000 zu verkleinern. Im DB Service Test wurde eine Konstante auf die Grösse der Testdatenbank gesetzt.

```
internal class DBServiceTest {

    private lateinit var dbService: DBService

    private val sizeTestDb = 1_001
    private val lastTestDbIndex = sizeTestDb - 1

    private var startIndex = 0
    private var startIndexRandom = (0 ≤ until (< lastTestDbIndex)).random()
    private var startIndexNegative = -3133

    private val pageSize = 20

    private lateinit var page: List<Playlist>
    private lateinit var randomPage: List<Playlist>
    private lateinit var negativePage: List<Playlist>
}
```

Abbildung 53: Der Grundaufbau des DB Service Tests.

Um verschiedene Szenarien zu prüfen und nicht immer nur die ersten paar Daten zu laden wurde entschieden, mit verschiedenen Start Indexen zu arbeiten. Dabei wurden der erste Index, ein zufälliger Index innerhalb der Testdatenbankgrösse und ein negativer Index gewählt, um zu gewährleisten, dass die DB Service Implementation stabil läuft. Anhand der Tests wurden dann in den DB Service Funktionen gewisse Überprüfungen eingebaut, die sicherstellten, dass bei Illegalen Argumenten (wie z.B. ungültiger Start Index) eine Exception geworfen und dass abgebrochen wird, bevor man mit den Illegalen Werten auf die Datenbank losgehen würde.

```
@Test
fun `getPage throws IllegalArgumentException on negative startIndex`() {
    printTestMethodName(object {}.javaClass.enclosingMethod.name)
    printFixedTestValues()

    runBlocking { this: CoroutineScope
        assertThrows<IllegalArgumentException> {
            Log.info { "Try to call getPage with negative startIndex $startIndexNegative" }
            dbService.getPage(startIndexNegative, pageSize)
            Log.info { "Has not thrown Exception!" }
        }
        Log.info { "Has thrown Exception!" }
    }
}
```

Abbildung 54: Beispiel eines Tests zur Prüfung des Start Index.

Es wurde angenommen, dass ohne zusätzliche Einstellungen Filter angegeben werden können und dass zwischen Gross und Kleinschreibung unterschieden wird. Durch exploratives Testen wurde herausgefunden, dass dies nicht der Fall ist. Eine Recherche ergab das Ergebnis, dass SQLite, die Datenbank, welche verwendet wurde, standardmässig beim LIKE Operator bei ASCII Zeichen (Zeichen von a-Z) nicht Case Sensitive ist und bei Unicode Zeichen, welche über die ASCII Range hinausgehen (ä, ö, ü, ...) schon. Das heisst, dass nicht zwischen einem «A» und einem «a» unterschieden wird. Dies zeigte sich vor allem an der Funktion getFilteredCount(), welche die gleiche Anzahl für beide Filter zurückgab. Zum Zeitpunkt der Tests war das Angeben von Filtern in der Tabelle noch ein weit entfernter Schritt. Durch das frühe Entdecken eines so zentralen Aspektes war eine Vorbereitung möglich und der getPage Funktion konnte ein caseSensitive Parameter mitgegeben werden. Um mit LIKE von SQLite Case Sensitive Filtern zu können, kann das PRAGMA Statement case_sensitive_like verwendet werden, welches genau für das Case Sensitive Filtern von ASCII Zeichen erstellt wurde. Einfachster Weg diese PRAGMA einzubauen war es, diese direkt an die JDBC URL als Query Parameter anzuhängen.

```
val makeSqliteCaseSensitive = "?case_sensitive_like=true"
Database.connect(
    url: "jdbc:sqlite:./src/test/resources/test_spotify_playlist_dataset.db$makeSqliteCaseSensitive",
    driver: "org.sqlite.JDBC"
)
```

Abbildung 55: PRAGMA an URL anhängen, um Case Sensitive Like zu erzwingen.

Somit wird nun standardmässig davon ausgegangen, dass eine Datenbank Case Sensitive filtert. Falls keine Case Sensitive Filterung erwünscht ist, wurde ausserdem eine Funktion erstellt, um dementsprechend zu reagieren und den ganzen Filter String in Lower Case zu verwandeln und so weiterzureichen.


```
/**
 * Is implemented with the thinking that the database will
 * use case sensitive like by default or with a pragma
 *
 * [dbField] must be lowerCase() in the else case
 */
private fun caseSensitiveLike(
    caseSensitive: Boolean,
    dbField: Column<String>,
    filter: String
): Op<Boolean> =
    if (caseSensitive) dbField like filter
    else dbField.lowerCase() like filter.lowercase(Locale.getDefault())
```

Abbildung 56: Unterscheidung zwischen Case Sensitiv oder Insensitiv Filtern.

Die DB Service Tests haben geholfen, ein besseres Verständnis für Exposed, SQLite und die in der Demo verwendete Datenbank zu bekommen.

4.3.2.2 Controller Tests

Der LazyTableController enthält alle Funktionen zur Verwaltung der Tabelle. Der Controller hat somit von allen Klassen die meiste Funktionalität. Dabei sind viele Funktionen Hilfsfunktionen, welche nicht von aussen aufgerufen werden sollten.

```
@Test
fun `selectModel sets the given playlistModel as selected`() {
    printTestMethodName(object {}.javaClass.enclosingMethod.name)

    // given
    val playlistModel = PlaylistModel(Playlist(name = "test"), appState)

    // when
    controller.selectModel(tableModel = playlistModel)

    // then
    assertEquals(playlistModel, appState.selectedTableModel)
}
```

Abbildung 57: Ein Beispiel eines Tests aus dem LazyTableControllerTest.

Daraus ergeben sich verschiedene Varianten dazu, mit welcher Sichtbarkeit diese Hilfsfunktionen markiert werden sollten:

Variante	Vorteile	Nachteile
Private.	- Funktionen nur innerhalb des Controllers aufrufbar.	- Nicht aufrufbar aus den Tests.
Protected.	- Funktionen innerhalb des Controllers oder einer Unterklasse des Controllers aufrufbar.	- Nicht aufrufbar aus den Tests.
Internal.	- Funktionen sind aus den Tests aufrufbar. - Funktionen sind trotzdem nicht öffentlich zugänglich.	- Innerhalb eines Moduls können andere Klassen auf die Funktionen zugreifen, obwohl sie eigentlich keinen Zugriff darauf haben sollten.
Public.	- Funktionen sind aus den Tests aufrufbar.	- Funktionen sind von überall aus zugreifbar.

Tabelle 6: Varianten der Hilfsfunktionen Sichtbarkeit.

Es wurde entschieden, die Hilfsfunktionen als internal zu markieren, um aus den Tests darauf zugreifen zu können. Grund dafür war, dass viel Funktionalität in diesen privaten Funktionen verborgen ist und es deshalb schwierig wäre, alles abzudecken, indem nur die öffentlichen Funktionen getestet werden. Die Funktionen bleiben aber trotzdem nicht öffentlich zugänglich.

Da dem Controller ein Paging Service mitgegeben werden muss, gab es verschiedene Möglichkeiten, auf einen solchen Paging Service zu gelangen:

Variante	Vorteile	Nachteile
Zugriff auf den DBService, welcher wiederum direkt auf die „Produktive“ Datenbank zugreift.	- Zugriff auf die produktiven Daten, mit welchen auch in der Tabelle in der Demo gearbeitet wird.	- Testumfang würde zu viele Ebenen einschliessen. Wenn auf Datenbankebene etwas schief geht laufen die Controller Tests nicht mehr. - Zu wenig unabhängig.
Erstellung eines FakePaging Service, um das Verhalten eines echten Paging Service zu emulieren.	- Emuliert mit wenig Code einen Paging Service. - Ohne zusätzliche Einarbeitung in ein Mocking Framework möglich.	- Muss selbst implementiert werden.
Erstellung eines Mocks von einem Paging Service.	- Wäre vermutlich einfach, bei einer Funktion ein gewisses Standardverhalten zu erstellen, welches diese dann immer gleich zurückgibt.	- Wenig Erfahrung mit Mocks. - Sollte einen echten Service emulieren und nicht einfach immer nur die gleichen Testwerte zurückliefern.

Tabelle 7: Möglichkeiten des Paging Service für Tests.

Es wurde entschieden, die Einarbeitungszeit für ein Mocking Framework (in Kotlin nochmals anders) zu sparen und die Controller Tests autonom zu behalten. Deswegen wurde schnell und einfach eine kleiner FakePagingService geschrieben.

```
class FakePagingService(val numberOfPlaylists: Int, val pageSize: Int) : IPagingService<Playlist> {

    private val allData = mutableMapOf<Int, List<Playlist>>()
    private val numberOfPages = PageUtils.getTotalPages(
        totalCount = numberOfPlaylists,
        pageSize = pageSize
    )

    init {
        for (pageNr in 0 until < numberOfPages) {
            val playlistsOfPage = mutableList<Playlist>()
            for (pageItemNr in 1 until < pageSize + 1) {
                val playlistId = (pageNr * pageSize) + pageItemNr.toLong()
                playlistsOfPage.add(Playlist(id = playlistId, name = "name $playlistId"))
            }
            allData[pageNr] = playlistsOfPage
        }
    }
}
```

Abbildung 58: Kurze Übersicht über den FakePagingService.

Dem FakePagingService wird die Anzahl Playlist Datensätze und die Page Grösse übergeben. Er verwaltet die «emulierte» Datenbank innerhalb einer Map allData, welche auf eine Page Nummer eine Liste von Playlist = Page zurückgibt. Die Daten reichen von Playlist ID 1 bis numberOfPlaylists.

Nach dem Erstellen des FakePagingService wurde anhand der Tests erkannt, dass eine ungünstige Abhängigkeit von der AppState Klasse zum DBService bestand. Die AppState Klasse wie auch die DBService Klasse waren mit object als Singletons markiert gewesen; so konnte von überall her auf sie zugegriffen werden. Dies hatte auch die AppState Klasse getan, indem sie auf dem DBService die getTotalCount() Funktion aufgerufen hatte, um die eine Liste zu initialisieren. Jedoch haben die Tests deshalb mit Exceptions abgebrochen. Um dieses Problem zu lösen, benötigte es eine Umstrukturierung des Codes.

Beim Starten wird zuerst ein pagingService erzeugt, welcher dann als Parameter dem LazyTableController übergeben wird. Der Controller erzeugt dann mit Hilfe des pagingService die AppState Klasse, welche auf dem pagingService getTotalCount für die Grösse der Liste aufruft.

```
val controller = LazyTableController(
    pagingService = DBService,
    defaultModel = PlaylistModel(Playlist()),
    mapToModels = mapToPlaylistModels
) // side effect: init loads first data to display
```

Abbildung 59: Aufruf beim App Start: Der Paging Service wird dem Controller mitgegeben.

```
class LazyTableController<T : BaseModel<*>> {
    private val pagingService: IPagingService<*>,
    defaultModel: T,
    private val mapToModels: (List<Any?>, AppState<BaseModel<*>>) -> List<T>,
    internal val pageSize: Int = 40,
} {
    val appState = AppState(pagingService, defaultModel, pageSize)
```

Abbildung 60: Der Controller erzeugt anhand des Paging Service die AppState Klasse.

```
private var pagingService: FakePagingService = FakePagingService(
    numberOfPlaylists = numberOfPlaylists,
    pageSize = pageSize
)

@BeforeEach
fun setUp() {
    controller = LazyTableController(
        pagingService = pagingService,
        pageSize = pageSize,
        defaultModel = PlaylistModel(Playlist()),
        mapToModels = {...}
    )
}
```

Abbildung 61: Aufruf im Controller Test.

Die Controller Tests konnten bei den zentralen Aspekten für das Verwalten der Tabelle Sicherheit geben. Ausserdem verhalfen die Tests zu einem Erkennen von schlechtem Code Design, da es zuerst schwierig war, Tests dafür zu schreiben. Die Überarbeitung der oben genannten Klassen verhalf zu einem besseren und stabileren Design. Grundsätzlich sollten die object Klassen auf ein Minimum reduziert werden, da diese wie Globale Variablen von überall aus zugreifbar sind, was in der Regel besser gelöst werden kann.

4.3.2.3 Scheduler Tests

Beim Scheduler wurde, nachdem dieser implementiert wurde, noch ein genaues Auge auf gewisse Details gelegt. Zum einen wurde sichergestellt, dass es zu keinem Overflow führt, wenn der Scheduler sehr oft (1 Million) aufgerufen wird. Dies wurde geprüft, da die process Funktion des Schedulers im invokeOnCompletion Block sich selbst aufruft, was sehr stark nach einem rekursiven Aufruf aussieht (in der IDE aber nicht als solcher gekennzeichnet wird).

```
private fun process() {
    if (inProcess) return
    if (task == null) return
    inProcess = true
    taskToDo = task
    CoroutineScope(Dispatchers.IO).launch {
        delay(delayInMillis)

        if (taskToDo == task) {
            taskToDo?.invoke()
        }
    }.invokeOnCompletion { it: Throwable?
        inProcess = false
        if (task == taskToDo) task = null
        process()
    }
}
```

Abbildung 62: Ein möglicher rekursiver Aufruf in der process Funktion.

Ein Test prüft den Task Overflow mit gegebenen Delays und weil der Scheduler dabei eine Million Mal aufgerufen wird, ist dieser Test normalerweise disabled mit einem Kommentar. Dieser Kommentar sagt aus, dass bei grösseren Änderungen am Scheduler dieser Test wieder eingeschaltet werden sollte, um zu prüfen, ob noch immer alles korrekt funktioniert.

Ebenfalls wurde getestet, ob immer nur der neueste Task verarbeitet wird; entweder mit einem angegebenen Delay oder ohne.

Ein letzter Test prüft, ob der Delay korrekt beachtet wird. Im mitgegebenen task wird ein Counter nach oben gezählt und nach drei Aufrufen, den Task zu schedulen und anschliessendem Warten soll am Ende der Counter auch den Wert drei haben.

```
@Test
fun delayIsConsidered() {
    var result = 0
    fun task() {
        result += 1
    }

    scheduler!!.scheduleTask { task() }
    Thread.sleep( millis: 155)
    scheduler!!.scheduleTask { task() }
    Thread.sleep( millis: 155)
    scheduler!!.scheduleTask { task() }

    Thread.sleep( millis: 1000)

    assertEquals( expected: 3, result)
}
```

Abbildung 63: Scheduler Test der prüft, ob der Delay korrekt beachtet wird.

4.3.2.4 UI Tests

Anhand der Android Entwickler Seite sahen UI Tests ziemlich straightforward aus und es kam die Hoffnung auf, dass dies genauso einfach und unkompliziert in Compose Desktop gehen würde. [15]

```
class MyComposeTest {

    @get:Rule
    val composeTestRule = createComposeRule()
    // use createAndroidComposeRule<YourActivity>() if you need access to
    // an activity

    @Test
    fun MyTest() {
        // Start the app
        composeTestRule.setContent {
            MyAppTheme {
                MainScreen(uiState = fakeUiState, /*...*/)
            }
        }

        composeTestRule.onNodeWithText("Continue").performClick()

        composeTestRule.onNodeWithText("Welcome").assertIsDisplayed()
    }
}
```

Abbildung 64: Beispiel von UI Tests im Android Umfeld.

```
@Suppress("WrapUnaryOperator")
@OptIn(ExperimentalTestApi::class)
class ScrollbarTest {
    @get:Rule
    val rule = createComposeRule()

    // don't inline, surface controls canvas life time
    private val surface = Surface.makeRasterN32Premul(100, 100)
    private val canvas = surface.canvas

    @Test
    fun `drag slider to the middle`() {
        runBlocking(Dispatchers.Main) {
            rule.setContent {
                TestBox(size = 100.dp, childSize = 20.dp, childCount = 10, scrollbarWidth = 10.dp)
            }
            rule.awaitIdle()

            rule.onNodeWithTag("scrollbar").performGesture {
                instantSwipe(start = Offset(0f, 25f), end = Offset(0f, 50f))
            }
            rule.awaitIdle()
            rule.onNodeWithTag("box0").assertTopPositionInRootIsEqualTo(-50.dp)
        }
    }
}
```

Abbildung 65: Beispiel eines Compose for Desktop UI Tests. ([Quelle Android Code Search](#))

Damit hätten unter anderem interessante End-to-End Tests durchgeführt werden:

Scheduler aus dem UI testen.	<ul style="list-style-type: none"> - Anhand eines schnellen Scrollens mit der Scrollbar sollten keine Daten geladen werden. - Langsameres Scrollen führt dazu, dass immer wieder nachgeladen wird.
Merge Mechanismus testen.	<ul style="list-style-type: none"> - Ein sichtbares Element wird bearbeitet, dann wird weggescrollt (damit neue Pages geladen werden) und wieder zum bearbeiteten Element zurückgescrollt. Dieses sollte immer noch die gleichen bearbeiteten Werte haben.
Tabellenverwaltung testen.	<ul style="list-style-type: none"> - Ausgewähltes Element muss beim Weg- und wieder Zurückscrollen wieder ausgewählt sein. - Synchronisierung zwischen der Tabelle und dem Formular funktioniert. - Beim Hin- und Wegscrollen kommen neue Instanzen der Elemente.
Filtern testen.	<ul style="list-style-type: none"> - Vom UI bis zum Service komplett testen, ob gesamter Vorgang funktioniert.
Sortieren testen.	<ul style="list-style-type: none"> - Vom UI bis zum Service komplett testen, ob gesamter Vorgang funktioniert.
Sonstiges testen.	<ul style="list-style-type: none"> - Alle anderen Szenarien, welche sonst immer manuell durchgespielt werden mussten, um zu prüfen, ob es Gesamtheitlich funktioniert.

Tabelle 8: End-to-End Tests, welche umgesetzt werden könnten.

Leider funktionierten die UI Tests im Projekt nicht. Es war zwar möglich eine Compose Test Rule zu erstellen, wenn dann aber auf dieser irgendeine Operationen aufgerufen werden wollten, erschien die Fehlermeldung „lateinit property scene has not been initialized“. Das Problem war, dass diese scene eine private Membervariable ist und nicht initialisiert werden kann. Ausserdem wurde der Test sonst im gleichen Stil aufgebaut, wie aus [Abbildung 64](#) und [Abbildung 65](#) hervorgeht. Zudem ist es wichtig darauf zu achten, dass die Tests mit JUnit4 geschrieben werden, da Compose eine starke Abhängigkeit von JUnit4 aufweist. Standardmässig werden jedoch JUnit5 Tests erstellt und alle anderen Tests sind JUnit5 Tests. Ebenfalls wichtig zu beachten ist, dass spezielle Dependencies sowie Einstellungen, die die gemeinsame Nutzung möglich machen benötigt werden, wenn JUnit4 und JUnit5 im selben Projekt zum Einsatz kommen.

Auch wenn es sicherlich spannend gewesen wäre, das Experiment noch lange weiterzuführen, musste es aus zeitlichen Gründen nach einer gewissen Zeit abgebrochen und der Fokus auf andere Dinge gelegt werden. Vermutlich wurden währenddessen auch kleine Details vergessen (zu welchen jedoch zu dieser frühen Zeit noch nicht wirklich Infos gefunden werden konnten) oder das Scheitern der Tests lag daran, dass *Compose for Desktop* noch immer ziemlich neu und frisch ist und vermutlich noch nicht alles so ausgereift ist. Nach dem Update auf Compose Version 1.1 wurde erneut probiert, die UI Tests zum Laufen zu bringen, jedoch ebenfalls ohne Erfolg.

4.3.2.5 Coverage

Bei der Coverage wurde kein spezifisches Ziel verfolgt. Trotzdem sollte so viel wie möglich getestet werden. Die Code Coverage der getesteten Klassen liegt bei allen um die 80% was ein solider Wert darstellt.

4.4 Generalisierung

In diesem Kapitel wird auf die Generalisierung von Compose BigLazyTable eingegangen. Diese ist von hoher Wichtigkeit – ohne sie würde eine Library wenig Sinn machen. Als Generalisierung wird der Vorgang bezeichnet, den Code so umzugestalten, dass es möglich ist mit beliebigen Daten die Tabelle dynamisch darzustellen. Ebenfalls ist damit in Compose Multiplattform die Verschiebung der generischen Klassen in das Common Modul und der konkreten für die Demo benötigten Klassen in das Demo Modul gemeint.

Später soll ein Entwickler nur gewisse Dinge selbst umsetzen müssen und den Rest der Compose BigLazyTable Library überlassen. Die einzigen Dinge, die der Entwickler zu erledigen hat, sind:

- Einen eigenen Service zu implementieren, welcher das PagingService Interface implementiert.
- Eigene Labels erstellen.
- Ein eigenes Presentation Model erstellen, welches die darzustellenden Daten, sogenannte DTOs, als Konstruktor Parameter mitbekommt.
- Die Verbindung zur Datenbank initialisieren.
- Alles entsprechend dem bereitgestellten Readme zusammenbauen. Mehr Details dazu sind im Projekt Readme zu finden (<https://github.com/FHNW-IP5-IP6/ComposeBigLazyTable#readme>).

Bis ein solches Szenario möglich war, mussten noch einige Probleme gelöst werden. Die Entwicklung wurde zuerst mit konkreten Implementierungen begonnen, um eine laufende Demo zu garantieren und dem Hintergedanken, später eine Generalisierung zu vollziehen. Grund dafür war auch, dass bei einem ersten Versuch zur Generalisierung (Verschieben des Codes in verschiedene Packages) viele unbekannte Gradle Fehlermeldungen erschienen sind, welche zu einem gewissen Respekt vor diesem Generalisierungsschritt führten.

Um eine Generalisierung des Codes zu erreichen, mussten alle hardcodierten Vorkommnisse des konkreten Presentation Models, in diesem Fall dem PlaylistModel, entfernt werden und dynamisch zugänglich gemacht werden.

Zu Beginn wurde der AppState dem Controller als Konstruktor Argument mitgegeben. Im Verlaufe des Projekts wurde dies umgeschrieben, so dass der Controller den AppState erzeugt und in sich verwaltet. Der AppState verwaltet ein default Presentation Model, welches hardcodiert war. Dieses konnte dynamisch ersetzt werden, indem es dem AppState als Konstruktor Parameter mitgegeben wurde. Da der AppState im Controller erzeugt wird, brauchte dementsprechend auch der Controller ein Konstruktor Parameter mit einem Presentation Model. Um keine konkrete Implementierung wie PlaylistModel als Typ zu verwenden, wurde anhand von Generics ein generischer Typ T gewählt, welcher von der BaseModel Klasse erbt. Alle Vorkommnisse von PlaylistModel konnten somit durch T ersetzt werden. Einziger Fall, an dem dieses Vorgehen nicht zur Lösung führte, trat beim Mappen der vom Service zurückgelieferten DTOs in ein Presentation Model auf. Grund war, dass das Presentation Model die Data Klasse als Konstruktor Parameter enthält. Anhand des generischen Typ T ist es aber unmöglich zu wissen, von welchem Typ die benötigte Data Klasse ist. Das Problem konnte mit Hilfe von Kotlin Lambdas gelöst werden. Anhand eines Lambdas wird beschrieben, wie die Liste von Data Klassen in eine Liste von Presentation

Models gemappt werden soll. Dieses Lambda wird ebenfalls dem Controller mitgegeben und ohne den genauen Typ der Data Klasse zu kennen, kann der Controller anschliessend dem Lambda, die vom Service erhaltene Page übergeben und die Presentation Models werden korrekt gemappt.

```
val page = pagingService.getPage(
    startIndex = startIndex,
    pageSize = pageSize,
    filters = filters,
    sort = sort
)

return mapToModels(page, appState as AppState<BaseModel<*>>)
```

Abbildung 66: Der Aufruf im Controller für das korrekte Mapping.

```
val mapToPlaylistModels: (List<Any?>, AppState<BaseModel<*>>) -> List<PlaylistModel> = { page, appState ->
    page.map { PlaylistModel(it as Playlist).apply { this.appState = appState } }
}
```

Abbildung 67: Das Lambda, das beschreibt, wie die Presentation Models gemapped werden sollen.

Das Lambda enthält aber nicht nur die Page, sondern auch den AppState. Dies muss zwingend so sein, da bei Änderungen auf einem Presentation Model die Liste der geänderten Models im AppState befüllt wird. Zu Beginn war AppState ebenfalls ein Konstruktor Parameter des Presentation Models. Dies hatte jedoch zur Folge, dass ein Entwickler für seine Presentation Model Klasse immer ein AppState Argument mitführen muss, obwohl der Entwickler diesen ausser beim Mapping nirgends benötigt. Eine weitere Konsequenz wäre, dass die updateChanged Funktion ebenfalls vom Entwickler hätte hinzugefügt und entsprechend implementiert werden. Um dies zu vermeiden wurde entschieden, den AppState in die BaseModel Klasse zu verschieben und die benötigten Änderungen direkt in der dortigen updateChanges Funktion hinzuzufügen. Der aktuelle Stand hat den Vorteil, dass der Entwickler den AppState nur bedingt kennen muss, für das Mapping Lambda aber sonst keinen AppState als Konstruktor Parameter angeben muss oder sogar noch selbst Logik zur updateChanges Funktion hinzufügen muss. Sicherlich würde es da auch andere Varianten für dieses Problem geben, aber diese wurden aus Zeitgründen nicht mehr genauer untersucht.

Zum BaseModel mussten ausserdem noch zwei weitere, in diesem Fall abstrakte, Membervariablen hinzugefügt werden. Zum einen die displayedAttributesInTable Liste und zum anderen die ID, welche dem Typ Attribut angehört. Diese beiden Ergänzungen wurden benötigt, da die View sowie auch Controller Überprüfungen anhand dieser beiden Felder durchführten. Da beide seit der Generalisierung kein konkretes PlaylistModel mehr verwendeten, sondern einen generischen Typ T: BaseModel war diese Änderung zwingend notwendig.

Nach den oben genannten Generalisierungsschritten wurde ein weiterer Versuch unternommen, die entsprechenden Klassen in das Common Modul zu verschieben. Eine Gradle Fehlermeldung war, dass eine zirkuläre Abhängigkeit zwischen dem Common und dem Demo Modul bestehen soll. Dies wurde verursacht, da im Common Modul eine Abhängigkeit auf das Demo Modul und umgekehrt vom Demo auf das Common Modul bestand. Beide Abhängigkeiten waren jedoch zwingend nötig, da ansonsten gewisse Klassen nicht mehr zugreifbar waren. Das darunterliegende Problem war aber eigentlich, dass die *Compose Forms*

Attribute im Demo Modul angesiedelt waren. Deswegen benötigte es vom Common Modul eine Abhängigkeit auf das Demo Modul. Es wurde entschieden, alle *Compose Forms* Attribute ebenfalls ins Common Modul zu übernehmen. Mit diesem Schritt konnte nicht nur die Abhängigkeit von Common auf Demo und die damit verbundene zirkuläre Abhängigkeit aufgelöst werden, sondern auch eine logische Zugehörigkeit zu den generischen Klassen erreicht werden. Im Demo Modul sollen nur Demo Applikationen und Projekte vorhanden sein, welche auf generische Klassen des Common Moduls zugreifen. Die praktischen Hilfsfunktionen von FilterUtils wurden ebenfalls in das Common Modul verschoben und können genau gleich auf einem Datenbank Service (welcher Exposed verwendet) aufgerufen werden.

4.5 Mögliche Erweiterungen

In diesem Kapitel werden mögliche Erweiterungen beschrieben und mit welchen Ansätzen diese implementiert werden könnten.

Aktuell wird für das Nummern Filter Textfeld eine lange Funktion verwendet, welche die Eingaben anhand der von Compose BigLazyTable definierten Filtersyntax einliest und verarbeitet. Das Einlesen des Nummern Filter ist umfasst mit einem grossen try-catch Block. Im Fehlerfall wird aktuell nichts gemacht. Die Validierung, die bei dem Nummern Filter Textfeld zum Einsatz kommt, erinnert stark an jene der von *Compose Forms* definierten Attribute. Eine mögliche Erweiterung wäre für das Nummern Filter Textfeld ein Attribut zu erstellen, welches im gleichen Stil wie die bisherigen Attribute validiert werden kann und bei einem invaliden Zustand entsprechend reagiert. Wahrscheinlich braucht ein solches Filter Attribut noch zusätzliche Eigenschaften im Vergleich zu den bisherigen Attributen.

In der aktuellen Version von Compose BigLazyTable wird mit einer festen Page Grösse gearbeitet. Eine mögliche Erweiterung wäre diese Page Grösse dynamisch der Anzahl angezeigten Tabellenzeilen anzupassen. Bei einem Wechsel von Fullscreen zu einem kleineren Fenster, soll sich die Page Grösse ebenfalls anpassen. Dabei könnten die angezeigten Membervariablen aus [Abbildung 68](#) von Vorteil sein.

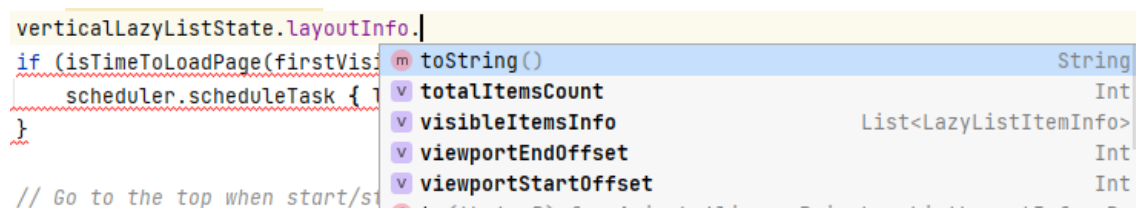


Abbildung 68: Informationen die über ein ScrollState verfügbar sind.

Anhand des aktuell verwendeten Cache und der Überschreibung der vom AppState angezeigten Liste, ist eine gleichbleibende Page Grösse zwingend. Hier könnte überlegt werden, ob diese Strukturen beibehalten werden möchte und anhand einer Funktion sichergestellt wird, dass die Daten so abgefüllt werden, dass keine Probleme wegen der wechselnden Page Grösse auftauchen. Falls sich dieser Ansatz als zu kompliziert erweist, könnte auch überlegt werden vom Cache und der AppState Liste Abstand zu nehmen, was das Verwalten sicherlich vereinfachen würde. Dann muss aber sichergestellt werden, dass die Tabelle weiterhin gleich flüssig läuft, wie dies aktuell der Fall ist.

Aktuell existieren für String, Boolean und Nummern Attribute angepasste UI Elemente, welche das Filtern mit dem entsprechenden Attribut erleichtern. Hier wäre eine mögliche Erweiterung für ein Selection Attribut und ein Decision Attribut ebenfalls passende UI Elemente zu verwenden. Bei einem Selection Attribut bietet sich ein Dropdown an, um zwischen den definierten Einträgen zu wählen. Beim Decision Attribut bietet sich entweder ein Dropdown oder etwas in einem ähnlichen Stile wie beim Boolean Attribut an. Also entweder im Dropdown eine Decision auszuwählen oder sich durch Klicks durch die Decisions zu togglen.

Eine weitere interessante Erweiterung wäre die Textfelder vom Formular direkt in die Tabelle zu übertragen. Dies wurde bereits in diesem Projekt untersucht aber anhand der Abhängigkeiten des *Compose Forms* internen Codes schliesslich nicht weiterverfolgt. Ziel

sollte sein, dass die validierbaren Textfelder einfach und ohne Probleme wiederverwendet werden können. Dies würde auch die erste mögliche Erweiterung eines Nummer Filter Attribut und Feld begünstigen.

Momentan besteht beim Datenbank Feld innerhalb der Filter und Sort Objekte eine feste Abhängigkeit zu Exposed. Eine Erweiterung wäre von dieser wegzukommen und beim Datenbank Feld mehr Freiheiten erlauben, ohne dass unbedingt mit Exposed gearbeitet werden muss. Dabei wäre es praktisch wenn die bisherigen Hilfsfunktionen von FilterUtils, welche ebenfalls stark von Exposed abhängen für weitere Implementierungen ausgebaut werden und dem Anwendungsentwickler mehr Freiraum bieten. Ebenfalls könnte untersucht werden, ob die aktuellen Filter anhand einer Kotlin inline reified Funktion generischer gemacht werden können und der aktuell existierenden Boilerplatecode für die Bestimmung des Typs weggelassen werden kann. [18]

Eine weitere spannende Erweiterung wäre herauszufinden, was Kotlin (auf der JVM) und *Compose for Desktop* für Möglichkeiten bieten der Internationalisierung. Es gilt zu evaluieren, ob eine Standardlösung vorhanden ist oder ob sogar eine fundierte Eigenentwicklung einer Internationalisierungs-Library Sinn machen würde.

Etwas das aktuell im UI der Tabelle noch etwas unschön wirkt, ist der Aufruf des `recomposeChanger`. Dabei sollte eine bessere Lösung gefunden werden.

Der Aspekt von geteiltem Code für *Compose Web* und Desktop, welcher völlig verlorengegangen ist durch den Wechsel von *Compose for Web* auf die *DOM DSL*, könnte weiter untersucht werden. Somit könnte *Compose BigLazyTable* anhand der *DOM DSL* ins Web übertragen werden.

Obwohl es bei Geschäftsapplikationen meistens unüblich ist eine perfekte Usability und User Experience aufzuweisen, könnte dieser Aspekt in einer möglichen Erweiterung vertiefter angesehen werden und die Tabelle so noch mehr von anderen abzuheben. Mögliche Punkte wären passende Farben für die Tabelle in einem Dunklen oder Hellen Theme, einen Tooltip zu haben zur Unterstützung fürs Filtern anhand der Filtersyntax, den passenden Cursor zu setzen oder die Tabellenbreite zur Laufzeit dynamisch anpassbar zu machen.

5 Schlussfolgerung

Compose for Desktop hat sich gut dafür geeignet, eine Tabelle mit den im Kapitel [Ziele des Projekts](#) beschriebenen Funktionalitäten zu implementieren. Wenn ein Entwickler bereits Erfahrung mit *Jetpack Compose* hat, ist es keine grosse Umstellung mit *Compose for Desktop* zu entwickeln. Durch den deklarativen Ansatz von *Compose for Desktop* ist das Programmieren deutlich angenehmer als bei *JavaFX*. Kotlin bietet viele Features, welche die Entwicklung einer Applikation intuitiv machen. Die Sprache hat sich als Alternative zu Java sehr bewährt.

Die Demo-Applikation hat gezeigt, dass es gut möglich ist, sehr viele Daten in der Applikation zu verwalten. Mithilfe der eingebauten Filter- und Sortiermöglichkeiten, kann ein End-Benutzer problemlos die Daten anschauen. In Kombination mit dem *Compose Forms* ist die Bearbeitung der Daten auch kein Problem mehr. Die im Hintergrund von *Compose BigLazyTable* implementierten Teile, wie zum Beispiel der Scheduler, haben dazu geführt, dass in der Tabelle mit vernachlässigbarer Verzögerung gescrollt werden kann. Das UI reagiert direkt auf User-Input und das Verwenden der Tabelle fühlt sich für den End-Benutzer sehr natürlich an.

Der eingebaute *Paging-Mechanismus* sorgt dafür, dass die Applikation keine Memory Probleme hat und die Daten schnell geladen werden. Obwohl der *Paging-Mechanismus* im Hintergrund läuft und der End-Benutzer nicht viel davon mitbekommt, ist dieser ein essenzieller Baustein der *Compose BigLazyTable Library*.

Jetpack Compose wird aktuell von Google stark gepusht und wird immer mehr bei Firmen im Android Umfeld als Alternative zu XML eingesetzt. *Compose for Desktop* ist hier noch sehr neu und deshalb noch nicht sehr verbreitet. Dadurch ist die Community noch nicht so gross und es kann schwieriger sein an Informationen zu gelangen. Hilfreich ist sicherlich das Beitreten zur Kotlin Slack Community. Dort gibt es spezifische Channels für die *Compose* Ausprägungen und Fragen werden immer in kürzester Zeit beantwortet.

Compose for Web ist noch stark in der Entwicklung. Das *DOM DSL* sieht aber bereits vielversprechend aus und Kotlin könnte in der nahen Zukunft auch im Web anzutreffen sein. Schade ist mit dieser Lösung nur, dass das UI Code Sharing zu den anderen *Compose* Varianten wegfällt.

Abschliessend ist zu sagen, dass Kotlin in Kombination mit *Compose for Desktop* so weit ausgereift ist, dass nichts im Wege steht, dies in der Ausbildung an der FHNW einzusetzen. Der Einstieg könnte sogar vielen Studierenden leichter fallen als bei Java mit *JavaFX*.

6 Reflexion

Das Projekt war eine grossartige Herausforderung für uns. Wir haben das erste Mal mit *Compose for Desktop* gearbeitet, was vor allem anspruchsvoll war, da es eine so neue Technologie ist. Wir hatten beide schon Erfahrung mit *Jetpack Compose*, jedoch ist in *Compose for Desktop* noch nicht alles genau gleich vorhanden und es gibt noch keine richtige Dokumentation dazu.

Das Setup des gesamten Projektes war durch das bereits gemachte IP5 Projekt nichts Neues mehr. Im Vergleich zu früheren Projekten im Studium war es sehr motivierend, dass man früh im Projekt schon voll programmieren konnte und dies auch im Fokus des Projektes stand. Man konnte sich voll auf die Entwicklung konzentrieren und am Schluss im Bericht genau über dies schreiben.

Wir haben es geschätzt, dass wir sehr frei in der Wahl des Schwerpunktes waren. Wir konnten unsere eigenen Ideen und Ziele einbringen und konkret an diesen Arbeiten. Durch die Inputs von Dr. Dieter Holz und Prof. Dierk König war es motivierend neue Funktionalitäten einzubauen und bestehende Ideen auch nochmals genauer zu überdenken.

Wir haben in diesem Projekt auch sehr viel im Pair-Programming gemacht. Damit haben wir sehr gute Erfahrungen gemacht und würden dies jedem weiterempfehlen. Man hat anschliessend ein viel besseres Verständnis vom Code und lernt von der anderen Person immer wieder Neues dazu. Die Qualität des Codes ist auch dadurch bei uns noch besser geworden.

Für ein zukünftiges Projekt würden wir noch mehr Coding-Sessions mit unserem Betreuer machen, da wir dieses Angebot jederzeit hatten und eher weniger genutzt haben. Dadurch hätten wir noch mehr Hinweise und Ideen zu unserem Code erhalten und hätten das ein oder andere anders gemacht.

Wir sind sehr zufrieden mit dem Stand, den wir erreichen konnten. Die *Compose BigLazyTable Library* funktioniert gut mit grösseren Datenmengen und hat viel Potenzial. Mit einem letzten grossen Schritt der gemacht wurde, der Generalisierung des Codes, kann die Library gut in Geschäftsapplikationen eingebaut werden und kann eine *JavaFX* Tabelle aus unserer Sicht sehr gut ablösen.

7 Literaturverzeichnis

- [1] Kanbanize. „Kanban, eine Anleitung für Einsteiger.“ Internet: <https://kanbanize.com/de/kanban-ressourcen/kanban-erste-schritte/was-ist-kanban>, [Mar. 16, 2022]
- [2] SonarSource S.A. „Code Quality and Code Security.“ Internet: <https://www.sonarqube.org/>, [Mar. 11, 2022]
- [3] SonarSource S.A. „Clean Code starts in your IDE.“ Internet: <https://www.sonarlint.org/>, [Mar. 11, 2022]
- [4] Skia. „About Skia.“ Internet: <https://skia.org/about/>, [Mar. 18, 2022]
- [5] JetBrains s.r.o. „Compose Multiplatform.“ Internet: <https://www.jetbrains.com/lp/compose-mpp/>, [Mar. 18, 2022]
- [6] JetBrains s.r.o. „Compose for Desktop.“ Internet: <https://www.jetbrains.com/lp/compose-desktop/>, [Mar. 18, 2022]
- [7] Jobtensor. „Moderne App-Entwicklung mit Kotlin – Übersicht und Vergleich der neuen Android-Entwicklungssprache mit Java.“ Internet: <https://jobtensor.com/Skill/Kotlin-vs-Java-Tutorial>, [Mar. 22, 2022]
- [8] JetBrains s.r.o. „Coroutines.“ Internet: <https://kotlinlang.org/docs/coroutines-overview.html>, Jul. 09, 2021 [Mar. 23, 2022]
- [9] Rainald Menge-Sonnentag. „Cross-Plattform: JetBrains bringt Compose auf den Desktop und ins Web.“ Internet: <https://www.heise.de/news/Cross-Plattform-JetBrains-bringt-Compose-auf-den-Desktop-und-ins-Web-6155707.html>, Aug. 05, 2021 [Mar. 23, 2022]
- [10] Brajesh Kumar. „Is MVC different from a 3 layered architecture ?.“ Internet: <https://www.quora.com/Is-MVC-different-from-a-3-layered-architecture>, 2020 [Mar. 20, 2022]
- [11] Nilanjan. „How to Access Database with Kotlin using JDBC: The Fundamentals.“ Internet: <https://levelup.gitconnected.com/how-to-access-database-with-kotlin-6b86f6680cd7>, Nov. 09, 2021 [Mar. 19, 2022]
- [12] Alesion Stalla. „Guide to the Kotlin Exposed Framework.“ Internet: <https://www.baeldung.com/kotlin/exposed-persistence>, Feb. 11, 2022 [Mar. 21, 2022]
- [13] JetBrains s.r.o. „Exposed Repository.“ Internet: <https://github.com/JetBrains/Exposed>, [Mar. 18, 2022]
- [14] JetBrains s.r.o. „compose-jb Repository.“ Internet: <https://github.com/JetBrains/compose-jb>, [Mar. 18, 2022]
- [15] Google Android. „Testing your Compose layout.“ Internet: <https://developer.android.com/jetpack/compose/testing>, [Mar. 18, 2022]
- [16] Steve Vogel, Louisa Reiniger. „Compose Forms Repository.“ Internet: <https://github.com/FHNW-IP5-IP6/ComposeForms>, [Mar. 18, 2022]
- [17] JetBrains s.r.o. „Scope functions.“ Internet: <https://kotlinlang.org/docs/scope-functions.html>, Feb. 09, 2022 [Mar. 24, 2022]
- [18] Mona Mohamadinia. „Reified Functions in Kotlin.“ Internet: <https://www.baeldung.com/kotlin/reified-functions>, Sep. 03, 2021 [Mar. 24, 2022]
- [19] Google Android. „Build better apps faster with Jetpack Compose.“ Internet: <https://developer.android.com/jetpack/compose>, [Mar. 24, 2022]

8 Abbildungsverzeichnis

Abbildung 1: JavaFX basierte Tabelle mit Sortier- und Filtermöglichkeiten.....	6
Abbildung 2: Compose BigLazyTable kombiniert mit einem Compose Forms	8
Abbildung 3: Beispiel von sequenzieller Programmierung mit Coroutines.	11
Abbildung 4: Coroutine Builder launch{} für den Aufruf einer suspending function.	12
Abbildung 5: Coroutine Builder "runBlocking{}" und "join()" für das Warten auf die suspending function.	12
Abbildung 6: "main"-Funktion als blocking definiert.	12
Abbildung 7: Beispiel von with im Compose BigLazyTable Code.	13
Abbildung 8: Die Definierung von States anhand der mutableState(Map)Of(initialValue) Erzeugerfunktionen.	15
Abbildung 9: Kotlin Multiplatform kombiniert plattformspezifische Libraries und Werkzeuge. [https://www.heise.de/news/Cross-Plattform-JetBrains-bringt-Compose-auf-den-Desktop-und-ins-Web-6155707.html]	17
Abbildung 10: Kanban-Board auf GitHub Project von Compose BigLazyTable	21
Abbildung 11: yaml-Skript für die GitHub Pipeline.....	23
Abbildung 12: Ausschnitt aus dem build.gradle File für die Konfiguration von SonarQube.	23
Abbildung 13: Ausschnitt Compose BigLazyTable Projekt auf SonarQube (Stand: 23.03.2022)	24
Abbildung 14: Ausschnitt aus originalen JSON-Daten einer Playlist.	25
Abbildung 15: Ausschnitt aus transformierten JSON-Daten einer Playlist.....	25
Abbildung 16: Eine Grobe Gesamtübersicht über Compose BigLazyTable.	27
Abbildung 17: Das Paging Service Interface.....	28
Abbildung 18: Ausschnitt der verwendeten Exposed Tabelle.....	30
Abbildung 19: Ausschnitt der „getTotalCount()“ Funktion in Exposed Kotlin SQL Syntax..	31
Abbildung 20: Ausschnitt aus der getPage() Funktion mit Filtern und Sortieren.	31
Abbildung 21: Übersicht über alle Filter Hilfsfunktionen.....	32
Abbildung 22: Die selectWithAllFilters() Extension Funktion.	32
Abbildung 23: Die retrieveSql() Funktion wählt je nach Filter Typ passende Funktion aus.	33
Abbildung 24: Die Unterscheidung zwischen den verschiedenen Filter Operatoren.	33
Abbildung 25: Die jeweiligen Funktionen zu den Filter Operatoren sind immer im gleichen Stil definiert	33
Abbildung 26: Ausschnitt aus der Playlist Klasse.	34
Abbildung 27: Ausschnitt aus der PlaylistDto Klasse.	34
Abbildung 28: Wrapper Klasse, um Details des Initialisierens der Datenbank vor dem Programmierer zu verbergen.....	35
Abbildung 29: Ausschnitt aus der PlaylistModel Klasse mit drei Attributen.....	37
Abbildung 30: Die Update Changes Funktion im Base Model mit den Änderungen für den Merge Mechanismus.	37
Abbildung 31: Die Liste mit den Attributen, welche in der Tabelle angezeigt werden sollen.	38
Abbildung 32: Übersicht über das Controller Package.....	39
Abbildung 33: Der LazyTableController mit allen Membervariablen und Funktionen.	39
Abbildung 34: Auslesen des firstVisibleItemIndex und Aufruf von loadNewPages auf Controller.	40
Abbildung 35: Der Merge Mechanismus im Code.	41

Abbildung 36: Ausschnitt einer Tabellen Zeile.....	41
Abbildung 37: jedes Mal wenn ein Model ausgewählt ist. wird die Sprache gesetzt.....	41
Abbildung 38: Die Sprachänderung.....	41
Abbildung 39: Die Ganze AppState Klasse.....	42
Abbildung 40: Die Ganze Scheduler Klasse.....	43
Abbildung 41: Die Grundstruktur des BigLazyTable UI.	44
Abbildung 42: Das Herzstück der Tabelle.....	45
Abbildung 43: Der Grundaufbau der Filter Klassen.....	46
Abbildung 44: Ausschnitt der Erkennung eines NumberAttribut und erstellen des entsprechenden Textfeldes. OnValueChanged wird an den Controller delegiert.....	47
Abbildung 45: Im Controller werden die Änderungen an die NumberTextfieldUtil Klasse gelegt.....	47
Abbildung 46: Die gesamte BLTSortOrder Klasse.	49
Abbildung 47: Der Sortier Button ist ein Icon Button und wird wie gezeigt aufgebaut.	49
Abbildung 48: Ausschnitt aus der onSortChanged Funktion im Controller.....	50
Abbildung 49: Ausschnitt des Sort Objekts.....	50
Abbildung 50: Ausschnitt einer möglichen Funktion.....	52
Abbildung 51: Einschalten der Gradle Test Logging Events	53
Abbildung 52: Erstelltes Test Live Template.	53
Abbildung 53: Der Grundaufbau des DB Service Tests.....	55
Abbildung 54: Beispiel eines Tests zur Prüfung des Start Index.	56
Abbildung 55: PRAGMA an URL anhängen, um Case Sensitive Like zu erzwingen.	56
Abbildung 56: Unterscheidung zwischen Case Sensitiv oder Insensitiv Filtern.	57
Abbildung 57: Ein Beispiel eines Tests aus dem LazyTableControllerTest.	57
Abbildung 58: Kurze Übersicht über den FakePagingService.	58
Abbildung 59: Aufruf beim App Start: Der Paging Service wird dem Controller mitgegeben.	59
Abbildung 60: Der Controller erzeugt anhand des Paging Service die AppState Klasse.....	59
Abbildung 61: Aufruf im Controller Test.....	59
Abbildung 62: Ein möglicher rekursiver Aufruf in der process Funktion.....	60
Abbildung 63: Scheduler Test der prüft, ob der Delay korrekt beachtet wird.....	60
Abbildung 64: Beispiel von UI Tests im Android Umfeld.....	61
Abbildung 65: Beispiel eines Compose for Desktop UI Tests. (Quelle Android Code Search)	61
Abbildung 66: Der Aufruf im Controller für das korrekte Mapping.	64
Abbildung 67: Das Lambda, das beschreibt, wie die Presentation Models gemapped werden sollen.	64
Abbildung 68: Informationen die über ein ScrollState verfügbar sind.....	66
Abbildung 69: Eine frühe Version des PagingService Interface mit String Filter.	75
Abbildung 70: Ausschnitt aus der Filter Klasse.	76
Abbildung 71: Ausschnitt anhand der StringAttribute Klasse, welches wie alle anderen Attribute mit den gezeigten Feldern erweitert wurden.	76
Abbildung 72: Ausschnitt, der zeigt wie ein Boolean Filter aufgesetzt wird.	77
Abbildung 73: Ausschnitt eines konkreten Nummer Filter anhand des IntFilter.	78
Abbildung 74: Das Between Objekt nimmt je einen Filter für den Start- und Endwert.....	78
Abbildung 75: Grundaufbau Compose BigLazyTable (Stand: 11.11.2021).	79

9 Tabellenverzeichnis

Tabelle 1: Varianten, um States zu definieren.	15
Tabelle 2: Beschreibung der verwendeten Kanban-Board Spalten.	21
Tabelle 3: Varianten der Datenbankbindung in Kotlin.	30
Tabelle 4: Varianten von Test Logging.	53
Tabelle 5: Varianten der Testdatenbank.	55
Tabelle 6: Varianten der Hilfsfunktionen Sichtbarkeit.	57
Tabelle 7: Möglichkeiten des Paging Service für Tests.....	58
Tabelle 8: End-to-End Tests, welche umgesetzt werden könnten.	62
Tabelle 9: Glossar	74

10 Anhang

10.1 Glossar

Begriff	Bedeutung
Compose for Desktop	Kotlin-Framework von JetBrains für die UI-Entwicklung von Desktop-Applikationen. Enthalten im Compose Multiplatform Projekt.
Compose for Web	Kotlin-Framework von JetBrains für die Entwicklung von reaktiven Web-Bedienoberflächen. Enthalten im Compose Multiplatform Projekt.
Compose Forms	Compose Forms ist eine Library, mit welcher sich schnell und einfach schöne & benutzerfreundliche Formulare erstellen lassen.
Compose Multiplatform	Schnelles, reaktives Kotlin-Framework für Desktop- und Web-UIs, basierend auf dem modernen Toolkit Jetpack Compose von Google und bereitgestellt von JetBrains. Mehr dazu im Kapitel Compose Multiplatform GitHub Projekt: https://github.com/JetBrains/compose-jb
DOM DSL	Domain spezifische Sprache für den Zugriff auf das Browser DOM (Document Object Model), welches in Compose for Web verwendet wird.
JavaFX	JavaFX ist ein Framework zur Erstellung plattformübergreifender Java-Applikationen.
Jetpack Compose	Jetpack Compose ist ein Toolkit von Android für die Entwicklung nativer Apps.
Lazy Loading	Daten werden erst geladen, wenn sie im UI gebraucht werden.
Paging-Mechanismus	Daten werden in kleinen Paketen geladen, welche als Page bezeichnet werden.
Recompose	Wenn sich die Daten eines States ändern, wird dies durch das Framework festgestellt und die «betroffenen» Teile des UI werden neu gerendert. Im Compose Jargon spricht man hierbei von einem Recompose.
Skia	Skia ist eine Open-Source-2D-Grafikbibliothek, die gemeinsame APIs bereitstellt, die auf einer Vielzahl von Hardware- und Softwareplattformen funktionieren. Sie dient als Grafik-Engine für Google Chrome und Chrome OS, Android, Flutter und viele andere Produkte.

Tabelle 9: Glossar

10.2 Filter Versionen

Version 0:

Zu Beginn war keine Filterung möglich. Die Grundvoraussetzungen waren aber anhand des Paging Service Interface gegeben. Ebenfalls war das Case Sensitive Filtern schon früh zu dieser Funktion hinzugekommen (anhand der Tests wurde festgestellt, dass die Filterung bei SQLite Datenbanken standardmässig nicht Case sensitive ist und deshalb gewisse Tests fehlschlagen).

```
fun getPage(startIndex: Int, pageSize: Int, filter: String = "", caseSensitive: Boolean = false): List<T>

fun getFilteredCount(filter: String, caseSensitive: Boolean = false): Int
```

Abbildung 69: Eine frühe Version des PagingService Interface mit String Filter.

Version 1:

Die AppState Klasse verwaltet bisher eine Liste aller Tabelleneinträge mit der Gesamtgrösse getTotalCount() vom Paging Service. Im Folgenden wird diese Liste als normale Liste bezeichnet. Um jedoch eine Filterung möglich zu machen, braucht es eine zusätzliche Liste, im folgenden filterListe genannt. Diese Liste hat anhand der Paging Service Funktion getFilteredCount(), bei der der entsprechende Filter als Parameter mitgegeben wird, eine dynamische Grösse.

Im UI wird dann anhand eines Boolean Flags geprüft, ob aktuell gefiltert wird. Dies ist der Fall, wenn mindestens ein Filter-Textfeld nicht leer ist. Beim Filtern wird auf die filterListe vom AppState zugegriffen; ist kein Filter gesetzt wird auf die normale Liste zugegriffen. Dadurch, dass es zwei unabhängige Listen sind funktioniert der Übergang vom Filtern zur normalen Liste sehr flüssig.

Die getPage Funktion arbeitet neben dem startIndex zusätzlich mit einem filter Parameter vom Typ String und einem boolean Parameter, der angibt, ob case sensitive gefiltert werden möchte.

Auf Datenbankebene ist Filtern nichts anderes als `SELECT dbField FROM dbTable WHERE dbField = filter`.

Um Filtern zu können wird also neben dem gesuchten filter Wert auch das entsprechende Datenbank Feld (dbField) benötigt. Da es aufgrund des Aufbaus der getPage Funktion nicht möglich war, Rückschlüsse darüber zu ziehen, auf welches dbField sich der String filter bezieht wurde das dbField «name» im DBService hardcodiert verwendet, um überhaupt das Filtern zu ermöglichen. Somit funktionierte nur die Filterung vom Namens-dbField dynamisch über die Eingabe in das entsprechende Namens-Textfeld im UI.

Version 2:

Ziel ist es, die vorhergehende Version so auszubauen, dass die Filterung für alle Tabellenspalten funktioniert. Um dies möglich zu machen, musste die getPage Funktion aufgerüstet werden, so dass neben dem Filter String auch noch das dazugehörige Datenbankfeld mitgegeben wird. Ebenfalls war es bei der vorherigen getPage Funktion nur möglich einen Filter String anzugeben. Das wurde verbessert, indem man der neuen getPage Funktion eine Liste von Filter Objekten mitgibt. Ein Filter Objekt enthält alle benötigten Informationen, um eine Spalte in der Tabelle zu filtern.

```
sealed class Filter

data class BooleanFilter(
    val filter: Boolean,
    val dbField: Column<Boolean>,
) : Filter()

sealed class MultipleOperationsFilter : Filter() {
    abstract val filterOperation: FilterOperation
}

data class StringFilter(
    val filter: String,
    val dbField: Column<String>,
    var caseSensitive: Boolean,
    override val filterOperation: FilterOperation = FilterOperation.EQUALS
) : MultipleOperationsFilter()
```

Abbildung 70: Ausschnitt aus der Filter Klasse.

Um dynamisch verwalten zu können, zu welchem Datenbankfeld nun ein zu filterndes Attribut gehören soll, wurden alle Attribut Klassen von *Compose Forms* um zwei Membervariablen erweitert. Hinzugefügt wurde, ob man dieses Attribut filtern kann und das zugehörige Datenbankfeld. Das Datenbankfeld ist standardmässig auf null gesetzt und muss zuerst beim Aufsetzen der verschiedenen Attribute mit diesen beiden Informationen ergänzt werden (siehe auch [Abbildung](#)). Standardmässig können alle Attribute gefiltert werden; wenn jedoch kein zugehöriges Datenbankfeld angegeben wird, hat der Filter keine Wirkung.

```
class StringAttribute<L>()
    canBeFiltered          : Boolean          = true,
    databaseField           : Column<String>? = null,
```

Abbildung 71: Ausschnitt anhand der StringAttribute Klasse, welches wie alle anderen Attribute mit den gezeigten Feldern erweitert wurden.

Zu allen Spalten der Tabelle existiert ein Textfeld. In dieses Textfeld wird der gesuchte Wert eingegeben. Dabei wird nicht zwischen Boolean-, Nummern- oder String-Attributen unterschieden. Bei der Filterung von Boolean-Attributen wird dann entweder «true» oder «false» in das Textfeld eingegeben. Bei den Nummern-Attributen wird die gesuchte Zahl eingegeben, es gäbe jedoch auch die Möglichkeit beliebige Zeichen einzugeben, welche zu keinem Suchresultat führen. Bei String-Attributen ist noch keine Case sensitive Filterung möglich und die Eingabe muss einfach enthalten sein (d.h. vor und nach dem Filter Wert werden Wildcards hinzugefügt). Zu diesem Zeitpunkt werden alle Werte als Strings gefiltert. Ein unerwarteter Nebeneffekt davon ist, dass eine Filterung eines Zahlenfelds mit dem Wert «10» nicht nur Einträge mit der Zahl 10 findet, sondern auch Einträge, bei denen die Zahl 10 vorkommt (z.B. 810 oder 101010). Grund dafür ist wie bereits oben erwähnt, dass alle String Werte vor und nach dem Filterwert Wildcards aufweisen und zusätzlich bei unserer Datenbank alle Werte als Strings abgespeichert waren.

Dieser Ansatz ist noch sehr unvollständig, macht aber immerhin schon einmal eine einfach gehaltene Filterung von allen Tabellenspalten möglich.

Version 3:

Ziel ist es, anhand des Typs der zu filternden Attribute passende Textfelder anzubieten. Damit sollen noch spezifischere Filter für die Attribute erstellt werden, welche das Filtern dem End-Benutzer erheblich erleichtern.

Nummer Attribute sollen als solche gefiltert werden und nicht anhand eines Strings mit Wildcards. Dies wurde erreicht, indem anstelle des bisher verwendeten „like“ ein „eq“ auf Datenbankebene verwendet wurde. Dies entspricht einem `==` einer Datenbankabfrage. Zuerst funktionierte das Abfragen anhand „eq“ nur für die erste Long Spalte ID, jedoch nicht für die anderen Nummern Spalten. Das Problem lag jedoch nicht beim Typ, sondern an unserer Datenbank, welche wie schon oben beschrieben nur aus String Werten bestand. Die einzige Spalte, welche aus Zahlen bestand war die ID Spalte, da diese der Primary Key ist. Nachdem die Datenbank überarbeitet wurde und die einzelnen Spalten auch den eigentlichen Wert aufwiesen, funktionierte bei allen Nummern Attributen die Filterung nach einer Zahl.

Boolean Attribute sollten anhand einer Checkbox gefiltert werden, damit nicht immer manuell true oder false eingegeben werden muss. Dies wurde mit einer von Compose bereits existierenden `TriStateCheckbox` gelöst. Wie der Name schon sagt, weist diese im Vergleich zur normalen Checkbox mit nur zwei Zuständen, drei Zustände auf: „Indeterminate“, „On“ und „Off“. Standardmässig ist die `TriStateCheckbox` auf „Indeterminate“ gestellt. Beim einem Klick auf die Check Box wird auf den Zustand „On“ gewechselt, welches die Tabellenspalte nach true Werten filtert. Bei einem erneuten Klick wird auf den Zustand „Off“ gewechselt, das nach false Werten filtert. Eine Erweiterung würde darin bestehen, dass statt denn Werten true und false auch die Werte eines Boolean Attributes in der Tabelle anhand einer Read-Only Checkbox angezeigt werden. Mehr dazu im Kapitel [Mögliche Erweiterungen](#).

```
is BooleanAttribute -> {  
    val toggleState = remember { mutableStateOf(ToggleableState.Indeterminate) }  
    TriStateCheckbox(  
        state = toggleState.value,  
        onClick = { controller.onBooleanFilterChanged(toggleState, attribute) }  
    )  
}
```

Abbildung 72: Ausschnitt, der zeigt wie ein Boolean Filter aufgesetzt wird.

String Attribute sollen durch den schon sehr früh existierenden Case sensitive Parameter gefiltert werden. Dies wurde erreicht, indem das bestehende Textfeld um ein zusätzliches führendes Icon erweitert wurde. Bei einem Klick darauf wurde somit dynamisch im UI ein Flag gesetzt, welches aussagt, dass dieses Attribut Case Sensitive gefiltert werden sollte. Hier wurde auch wieder eine Map verwendet, um über alle Attribute spezifische Änderungen machen zu können. Ohne die Map hätte ein Klick auf das Case Sensitive Icon bei einem Attribut auch dazu geführt, dass alle anderen Attribute Case Sensitive gefiltert worden wären.

Ein erhebliches Problem dieser Version bestand darin, dass immer noch alle Datenbankfelder in String Spalten gecastet wurden. Die Auswirkung war, dass so automatisch ein `Column<String>` erwartet wurde und nicht genauer spezifiziert werden musste, welchem Typ die Datenbankkolonne angehört. Eine mögliche Lösung wäre gewesen, das Datenbankfeld als `Column<T>` generisch anzugeben. Problem dieses Ansatzes war jedoch, dass man anhand von

T nicht wusste welcher genaue Typ nun dahinter steckt. Natürlich wäre es möglich gewesen, nur auf der untersten Ebene einen cast nach `Column<String>` zu machen, so oder so stellte es aber noch keine saubere Lösung dar.

Version 4:

Ziel ist es, Zahlen nicht nur mit genau dem eingegebenen Wert zu filtern, sondern noch weitere hilfreiche Zahlenfiltermöglichkeiten zu entwickeln. Als Idee wären `<`, `<=`, `>`, `>=`, `!=` und falls noch genug Zeit bleibt eine Art «between» Filterung hilfreich für den End-Benutzer. Um diese Art von Filterung möglich zu machen, musste von der Standard `Column<String>` Variante abgewichen werden, da Strings nicht auf grösser, kleiner oder between überprüft werden können. Um dies zu ermöglichen, wurde mit dem entsprechenden Typ als Datenbankfeld anhand einer sealed class für jeden Filter eine eigene Klasse entwickelt. Im UI wurde das ganze „parsen“ in einem grossen when-Block umgesetzt, wo der Textfeld-Inhalt auf die entsprechenden Filterzeichen überprüft und, falls diese vorkommen und danach eine Zahl auftritt, dann der entsprechende spezifische Filter gesetzt wird (z.B. `IntFilter`). Dieser wird dann via Controller an den Datenbank Service weitergereicht, wo nun eine Fallunterscheidung darüber vorgenommen werden muss, welchem Typ der Filter angehört. Danach sind die oben genannten Zahlenfiltermöglichkeiten nur noch Formsache. Durch diesen Approach musste leider etwas mehr Code geschrieben werden. An dieser Stelle kommt der Gedanke auf, ob dies nicht mit Generics hätte gelöst werden können – dies würde jedoch eine sehr komplexe Aufgabe darstellen, weswegen der Entscheid fiel, es so zu vereinfachen.

```
data class IntFilter(
    val filter: Int,
    val dbField: Column<Int>,
    override val filterOperation: FilterOperation,
    val between: Between<Int>? = null
) : MultipleOperationsFilter()
```

Abbildung 73: Ausschnitt eines konkreten Nummer Filter anhand des `IntFilter`.

```
data class Between<T>() {
    val fromFilter: Filter,
    val toFilter: Filter
}
```

Abbildung 74: Das `Between` Objekt nimmt je einen Filter für den Start- und Endwert.

Um den String Filter noch weiter zu verfeinern, wurde entschieden, es dem End-Benutzer zu überlassen, die Wildcards mit % im Textfeld zu setzen. Dadurch sind weitere Kombinationen wie `startsWith` / `endsWith` / `contains` / `exactly` oder auch Filterung nach einem bestimmten Muster von Strings möglich. Kurz gesagt funktioniert der String Filter nach dem gleichen Muster wie Datenbankabfragen mit `like`.

10.3 Skizze Grundaufbau Compose BigLazyTable

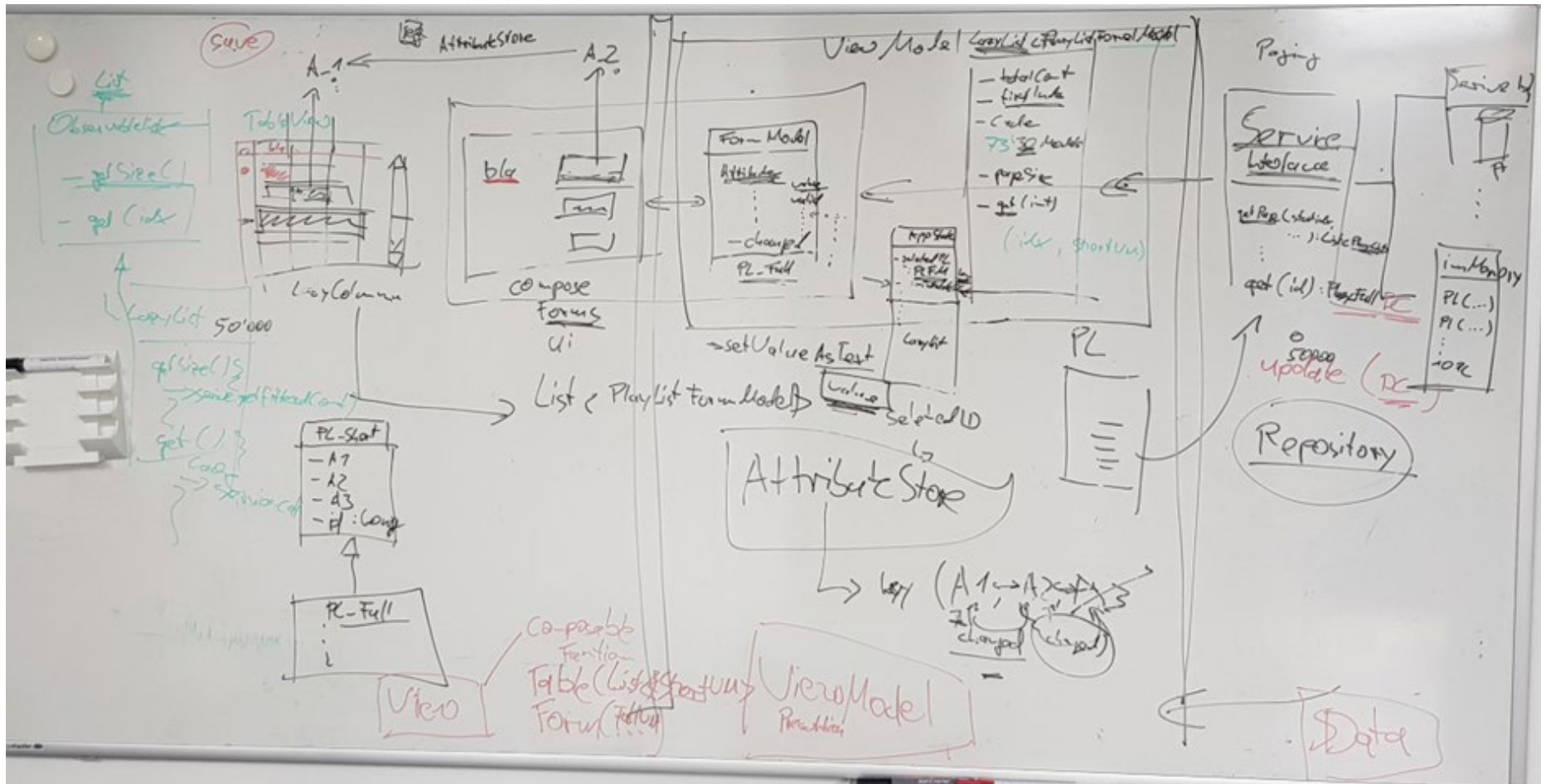


Abbildung 75: Grundaufbau Compose BigLazyTable (Stand: 11.11.2021).

11 Ehrlichkeitserklärung

Hiermit erklären wir, dass wir den vorliegenden Projektbericht des IP6 selbständig, ohne Hilfe von Dritten und nur unter Verwendung der angegebenen Quellen verfasst haben.

Windisch, 25.03.2022



Näf Livio

Windisch, 25.03.2022



Sprenger Marco