

UNIWERSYTET ŚLĄSKI  
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH

**Aleksander Jaworski**

313998

**KOTLIN AS A CROSS-PLATFORM  
PROGRAMMING LANGUAGE  
FOR ANDROID AND THE WEB**

MASTER'S THESIS

THESIS SUPERVISOR:  
Anna Gorczyca-Goraj, PhD Eng.

Katowice 2021

## **Abstract**

This thesis focuses on the capabilities of the Kotlin programming language, in particular on the ability to create a codebase which can be shared between different platforms. The thesis is backed by a mobile Android application, the web part consists of a React.JS web application and a Back-end server which is used by both applications. Every part of the project is created with Kotlin and the client applications share the same core logic.

## **Oświadczenie autora pracy**

Ja, niżej podpisany, autor pracy dyplomowej pt. Kotlin as a cross-platform programming language for Android and the Web, o numerze albumu: 313998, student Wydziału Nauk Ścisłych i Technicznych Uniwersytetu Śląskiego w Katowicach, kierunku studiów Informatyka oświadczam, że ww. praca dyplomowa:

- została przygotowana przeze mnie samodzielnie<sup>1</sup>,
- nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994r. o prawie autorskim i prawach pokrewnych (tekst jednolity Dz. U. z 2006r. Nr 90, poz. 631, z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- nie zawiera danych i informacji, które uzyskałem w sposób niedozwolony,
- nie była podstawą nadania dyplomu uczelni wyższej lub tytułu zawodowego ani mnie, ani innej osoby.

Oświadczam również, że treść pracy dyplomowej zamieszczonej przeze mnie w Archiwum Prac Dyplomowych jest identyczna z treścią zawartą w wydrukowanej wersji pracy.

**Jestem świadomy odpowiedzialności karnej za złożenie fałszywego oświadczenia.**

.....

(miejsce i data)

.....

(podpis autora pracy)

---

<sup>1</sup>uwzględniając merytoryczny wkład promotora (w ramach prowadzonego seminarium dyplomowego)

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation . . . . .	4
1.2	The project . . . . .	4
1.3	The Kotlin programming language . . . . .	4
1.3.1	Immutability . . . . .	5
1.3.2	Type inference and smart casting . . . . .	8
1.3.3	When statement . . . . .	9
1.3.4	Default and named arguments . . . . .	9
1.3.5	String templates . . . . .	10
1.3.6	Null safety . . . . .	10
1.3.7	Data class . . . . .	12
1.3.8	Objects . . . . .	13
1.3.9	Sealed class . . . . .	14
1.3.10	Extension functions . . . . .	14
1.3.11	Higher-order functions . . . . .	15
1.3.12	DSL . . . . .	16
1.3.13	Coroutines . . . . .	17
1.4	Kotlin Multiplatform . . . . .	21
1.5	Proof of concept . . . . .	22
1.6	Open source . . . . .	22
<b>2</b>	<b>The project</b>	<b>23</b>
2.1	Project structure . . . . .	23
2.1.1	Gradle [16] . . . . .	24
2.2	GitHub Actions . . . . .	24
2.2.1	Static code analysis . . . . .	25
2.3	Dependency Injection . . . . .	25
2.4	Ktor server . . . . .	27
2.4.1	PostgreSQL Database . . . . .	27
2.4.1.1	Note Schema . . . . .	27
2.4.2	Authentication . . . . .	27
2.4.3	Websocket . . . . .	28
2.4.3.1	Client . . . . .	28
2.4.3.2	Server . . . . .	28
2.4.4	Technologies used . . . . .	29
2.4.5	Hosting . . . . .	30
2.5	Shared module . . . . .	30

2.5.1	Structure . . . . .	30
2.5.2	How the code is shared . . . . .	31
2.5.2.1	The Expect and Actual declaration . . . . .	31
2.5.2.2	Dependency inversion . . . . .	33
2.5.3	Module contents . . . . .	36
2.5.3.1	Features . . . . .	36
2.5.3.2	Data structures . . . . .	36
2.5.4	Architecture . . . . .	37
2.5.4.1	View layer . . . . .	37
2.5.4.2	Presentation layer . . . . .	38
2.5.4.3	Domain layer . . . . .	38
2.5.4.4	Data layer . . . . .	39
2.5.5	Testing . . . . .	41
2.5.6	Technologies used . . . . .	41
2.6	Web app . . . . .	42
2.6.1	Kotlin/JS . . . . .	42
2.6.1.1	Calling JavaScript from Kotlin . . . . .	42
2.6.1.2	Using JavaScript libraries . . . . .	44
2.6.1.3	Kotlin React.JS . . . . .	45
2.6.2	Modules . . . . .	49
2.6.3	Architecture . . . . .	49
2.6.3.1	Redux . . . . .	49
2.6.3.2	View layer . . . . .	50
2.6.3.3	Presentation layer . . . . .	51
2.6.3.4	Summary . . . . .	52
2.6.4	Technologies used . . . . .	53
2.7	Android app . . . . .	53
2.7.1	Kotlin/JVM . . . . .	55
2.7.2	Modules . . . . .	55
2.7.3	Architecture . . . . .	56
2.7.3.1	Model-View-Viewmodel . . . . .	56
2.7.3.2	View layer . . . . .	56
2.7.3.3	Presentation layer . . . . .	56
2.7.3.4	Summary . . . . .	58
2.7.4	Testing . . . . .	58
2.7.5	Technologies used . . . . .	58
2.8	Code distribution . . . . .	59

<b>3 Hiccups/ problems encountered</b>	<b>63</b>
3.1 Kotlin/JS wrapper dependency . . . . .	63
3.2 Kotlin 1.4 update . . . . .	63
3.3 Platform dependent implementation problems . . . . .	63
3.4 Testing the shared module . . . . .	64
3.5 Broken navigation to declaration . . . . .	64
3.6 Kotlin/JS RAM usage . . . . .	65
<b>4 Summary</b>	<b>66</b>
4.1 Synopsis . . . . .	66
4.1.1 Kotlin/JVM . . . . .	66
4.1.2 Kotlin/JS . . . . .	66
4.1.3 Kotlin/Native and more . . . . .	67
4.1.4 Kotlin overview . . . . .	67
4.2 Future outlook . . . . .	68
4.2.1 Compose . . . . .	68
4.2.2 Kotlin Multiplatform Mobile . . . . .	69

# 1 Introduction

According to the StackOverflow 2020 survey [1], Kotlin is the 4<sup>th</sup> most *Loved* language. The survey is shown in Figure 51.

## 1.1 Motivation

I have started my professional career as a Web developer. After three years I have transitioned into Android App development. To develop Android apps I have used primarily the Kotlin programming language which can be seamlessly integrated with Java and the Java Virtual Machine (JVM). Additionally, Kotlin has also an option to be translated into JavaScript, which allows for creating Web Applications in Kotlin. Given my past experience I have wanted to make an App which is native to both Android and a browser. Leveraging the power of Kotlin Multiplatform I can share some parts of the codebase between these platforms, what will be described in details in the next sections.

## 1.2 The project

The project includes a native Android app, a React.JS [2] web app later called “Clients” and a Ktor server shown in Figure 1. The client applications allow the user to create, edit and delete notes. All of these operations are synchronized in real time, meaning that every change is visible on all other devices as long as the user is connected to the internet and authenticated using the same account. The server is responsible for providing an API to the clients, managing the database and notifying the clients of any database changes using web sockets.

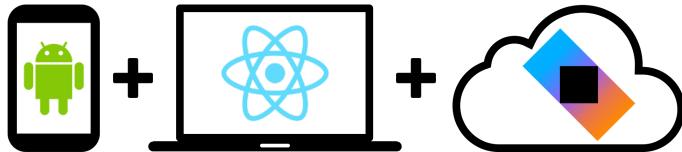


Figure 1: A visualization of the project credit, images were plotted based on:

[https://www.iconfinder.com/icons/2205195/mobile\\_phone\\_screen\\_smart\\_icon](https://www.iconfinder.com/icons/2205195/mobile_phone_screen_smart_icon)

[https://www.iconfinder.com/icons/171511/laptop\\_computer\\_icon](https://www.iconfinder.com/icons/171511/laptop_computer_icon)

[https://www.iconfinder.com/icons/370088/cloud\\_weather\\_cloudy\\_winter\\_icon](https://www.iconfinder.com/icons/370088/cloud_weather_cloudy_winter_icon)

## 1.3 The Kotlin programming language

Kotlin [3] is a modern open source statically typed programming language which can be used on multiple platforms. The language is currently under active development by JetBrains [4] and many open source contributors.



Figure 2: The Kotlin programming language logo <https://kotlinlang.org/>

The language prides itself on being concise, safe and interoperable. It can be used along with the Java programming language [5] with 100% interoperability [6]. Thanks to this feature Kotlin has become an official programming language for Android development [7].

The code written in Kotlin can use elements of object oriented programming, functional programming [8] and a mix of both.

Kotlin can be used for:

- Android app development;
- Server side development;
- Client side web development;
- Desktop app development;
- With the addition of Kotlin Multiplatform [9] and Kotlin Native [10] it can bridge the gap with iOS [11], and embedded systems development.

In the coming sections I will present Kotlin features.

### 1.3.1 Immutability

Immutability is not exactly a feature of the language, but Kotlin puts a big emphasis on it. Immutability ensures that a variable or an object cannot be changed once it is created. This makes the code more predictable and side-effect free.

For example setting a value depending on a condition could look like this:

```
1 var color: String
2 if (condition) {
3     color = "#000"
4 } else {
5     color = "#FFF"
6 }
```

Depending on the **condition** a different color is assigned to the color variable. The problem is that the color is declared as a **var** which means that it can be modified at a later point in the code. One solution for this problem would be declaring the color variable as a **val** which works because the compiler knows that the color variable never changes. Thanks to this the color variable is a

**val** meaning that it is immutable and cannot be changed thus making the code easier to reason about.

Additionally to make the code more readable the condition could be moved to a separate function or using the *if* condition as an expression:

```
1 val color: String = if (condition) "#000" else "#FFF"
```

Kotlin does not provide a ternary operator which is available in a lot of programming languages but allows for using conditions as expressions.

Mutable collections, lists, tables etc. often introduce a lot of unexpected behavior into a program. When a collection is referenced in a couple of places (later called clients) all of these clients suffer from an invisible interdependency which is hidden in the mutable collection. For example if a client changes a value in the collection, then this change propagates to all other clients that have a reference to that collection, even if they do not expect it.

Kotlin helps to solve this problem by defining a clear separation between mutable and immutable collections with the addition of leveraging the functional programming paradigm.

```
1 interface List<out E> : Collection<E>
2 interface MutableList<E> : List<E>, MutableCollection<E>
```

The **List** interface is used for immutable lists, and likewise the **MutableList** is used for mutable lists. The convention of using the **Mutable** prefix is prevalent in the Kotlin standard library and it is also often adopted in community libraries.

I will present multiple ways of implementing an example function which from an input of [1, 2, 3, null] returns [10, 30]. Let's start with the fully mutable way:

```
1 fun example(): MutableList<Int> {
2     val input: List<Int?> = listOf(1, 2, 3, null)
3     val result: MutableList<Int> = mutableListOf()
4     for (number in input) {
5         if (number != null && number % 2 == 1) {
6             result.add(number * 10)
7         }
8     }
9     return result
10 }
```

In this function the **result** is a mutable collection which is modified in the *for* loop. A worrying thing, however, is that it returns a mutable list, which can be subsequently modified and it may introduce errors later. To solve this problem the function could be slightly modified:

```
1 fun example(): List<Int> {
2     val input: List<Int?> = listOf(1, 2, 3, null)
```

```

3   val result: MutableList<Int> = mutableListOf()
4   input.forEach { number ->
5       if (number != null && number % 2 == 1) {
6           result.add(number * 10)
7       }
8   }
9   return result
10 }
```

In this function the *for* loop and the *return* type were changed. The *for* loop and the new *forEach* declaration are semantically equivalent and used just to show a different way of iterating over a collection. The return type of the function is a List, yet a MutableList is returned. This is possible because the MutableList interface implements the List interface meaning that a MutableList can be used in every place a List is required. Thanks to the fact that an immutable list is returned its value cannot ever be changed even though a mutable list was used to produce the result.

The functional solution leverages the built in Kotlin functional operators filter and map:

```

1 fun example(): List<Int> {
2     val input: List<Int?> = listOf(1, 2, 3, null)
3     return input
4         .filterNotNull()
5         .filter { number -> number % 2 == 1 }
6         .map { number -> number * 10 }
7 }
```

This solution is fully immutable and does not require declaring any mutable lists other than the ones hidden in the operator implementations. One disadvantage of using functional operators is the additional list creation underneath. In the example above three lists were created in order to produce the expected result. The number of lists is often equivalent to the number of operators, however this can be mitigated by creating an operator for that specific case (shown in section 1.3.11) or using other built-in operators like mapNotNull:

```

1 fun example(): List<Int> {
2     val input: List<Int?> = listOf(1, 2, 3, null)
3     return input
4         .filter { number -> number != null && number % 2 == 1 }
5         .mapNotNull { number ->
6             if (number != null) number * 10 else null
7         }
8 }
```

Summarizing, all of the example functions described in this section produce the same result, however the first solution suffers from mutability problems. The last two solutions are more elegant and concise but may be slower and use more memory because they create additional lists. In Kotlin using the functional operators is the preferred way of working with collections but when the number of elements is very high and there are a lot of operations involved then the solution presented in the second example would be a better fit. Modifying a mutable collection is faster and uses less memory than creating intermediary lists. The most important thing is keeping the mutation local and not public.

Besides the things described in this section, Kotlin does a lot more to help with enforcing immutability, some of which will be briefly described in the coming sections.

### 1.3.2 Type inference and smart casting

When declaring a variable, the type does not need to be specified directly thanks to type inference.

For example:

```
1 val colorWithType: String = "#000"
2 val colorWithoutType = "#000"
```

Both of the declarations above are semantically equivalent. The type inference is often used to shorten a line length and omit the obvious, which in the case above is that the second variable is a string.

Smart casting helps one avoid unnecessary casting (changing the type of the variable):

```
1 fun getStringLength(anyValue: Any): Int {
2     if (anyValue is String) {
3         return anyValue.length
4     } else {
5         return 0
6     }
7 }
```

When the condition (`anyValue` is `String`) is true, the compiler already knows that `anyValue` is a string and the variable can be used as such. In Kotlin this is often used when working with nullable values.

```
1 fun operateOnInt(number: Int?) {
2     if (number == null) return
3     val squared = number * number
4     // ...
5 }
```

In the case when the passed in **number** parameter is null the function returns early which means that the operation cannot be performed. However, if the condition is false the **number** parameter can be safely used as a non null value.

### 1.3.3 When statement

In Kotlin there is no *switch* statement but there is a *when* statement that can be used in a similar manner:

```
1 when (value) {  
2     0 -> println("0")  
3     1 -> println("1")  
4     in (2..5) -> println("Between 2 and 5")  
5     else -> println("No matches")  
6 }
```

There is an additional way in which the *when* statement can be used:

```
1 when {  
2     value == 0 -> println("0")  
3     condition == true -> println("The condition is true")  
4     else -> println("No matches")  
5 }
```

Without providing a variable to the *when* statement it can be used in a similar way to an *if* statement.

Because the *when* statement is a condition it can be also used as an expression:

```
1 val newState = when (action) {  
2     is SetNotesList -> /* ommited */  
3     is SetIsLoading -> /* ommited */  
4     else -> /* ommited */  
5 }
```

### 1.3.4 Default and named arguments

Thanks to default arguments Kotlin does not require declaring multiple functions with a different number of arguments (i.e. Java overloading). One function can be defined and in the case an argument is not provided, then a default value is used:

```
1 fun buildString(  
2     baseText: String,  
3     toUpperCase: Boolean = false,
```

```

4           times: Int = 1
5 ): String {
6     val text = if (toUpperCase) baseText.toUpperCase() else baseText
7     return text.repeat(times)
8 }
```

Thanks to this the **buildString** function can be called in the following manner:

```

1 buildString("Abc", true, 2) // ABCABC
2 buildString("Abc", true) // ABC
3 buildString("Abc") // Abc
```

Named arguments allow for specifying a name of an argument when passing in the value to the function. Expanding upon the previous example by adding named arguments in the **buildString**:

```

1 buildString(
2     baseText = "Abc",
3     toUpperCase = true,
4     times = 2
5 ) // ABCABC
6 buildString("Abc", times = 2) // AbcAbc
```

Both the default and named arguments can also be used on class constructors which makes the Builder pattern [12] and the Telescoping constructor pattern [13] obsolete in most cases.

### 1.3.5 String templates

This feature allows for fast and easy string concatenation.

```

1 val value = -5
2 println("direct\u00a5value\u00a5$value") // -5
3 println("nested\u00a5value\u00a5${value.absoluteValue}") // 5
4 println("side\u00a5by\u00a5side\u00a5value\u00a5${value}\u00a5${value}") // -5-5
```

The dollar sign (\$) allows for referencing variables from the string. Using it without curly braces ({\$}) is sufficient for most cases, however there are situations where curly braces are required. For example, executing a function or referencing nested value where a dot (.) needs to be used in order to retrieve the value.

### 1.3.6 Null safety

As it was shown in the previous examples the Kotlin type system differentiates between null and non-null values:

```
1 var nullableValue: String? = null
2 var nonNullValue: String = "abc"
```

Every null value type ends with a question mark (?). Because null types are supported by the type system this means that possible errors occurring from a null value are caught by the compiler (most of them, other cases will be described later). Writing instructions which would result in a NullPointerException are prevented from compiling:

```
1 var nullableValue: String? = "123"
2 println(nullableValue.length)
```

The code above will not compile, because of the **nullableValue** variable. On the next line it is used as a non-null value, however its type specifies that it can be null. If somehow the nullableValue variable was null, the program would crash with a NullPointerException at run-time. To get around this compilation error Kotlin allows accessing nullable variables using a safe call (?.):

```
1 var nullableValue: String? = "123"
2 var nullableLength: Int? = nullableValue?.length
```

In the above example the variable **nullableLength** either contains the length of a string or null depending on the initial value. In cases when a default value can be used in place of a null, the so-called elvis operator (?:) can be used:

```
1 var nullableValue: String? = "123"
2 var nullableLength: Int = nullableValue?.length ?: 0
```

Thanks to this the variable **nullableLength** can be non-null because it either contains the actual length or 0. Sometimes the type system cannot infer that a nullable value is not a null, to force the compilation of the program, the double bang operator (!! ) can be used:

```
1 var nullableValue: String? = "123"
2 var nullableLength: Int = nullableValue?.length !!
```

However, this operator should be avoided in almost all circumstances because it switches off Kotlin null compile time safety, which can lead to NullPointerException crashes.

Kotlin type system tries to prevent from NullPointerExceptions but it is not perfect. Most unexpected NullPointerExceptions come from so-called "platform types" which are a result of Kotlin interoperability. Platform types are a result of Kotlin calling for classes which were created in a different language. For example: Java does not enforce nullability and non-nullability in the type system. As a consequence when Kotlin calls a Java function that returns a string, the result could be a string or null. There is no guarantee without looking at the implementation details of the Java function. However, there are ways of making Java null-safe, but I will omit them here on purpose. A null returned from the Java function could result in a NullPointerException if Kotlin

treated it as non-null. In such cases it is better to always treat platform types as nullable in order to prevent crashes.

There are other ways of getting a NullPointerException in Kotlin but they are edge cases which do not happen that often. Most of the time they're related to property initializations in Kotlin. Because of their rare occurrence I will not go into their details.

### 1.3.7 Data class

Data classes create a clear separation between classes that contain behavior and classes that only hold data (data structures):

```
1 data class Person(  
2     val firstName: String,  
3     val lastName: String,  
4     val age: Int  
5 )
```

The class above contains three properties which in combination represent a Person. Because the Person class is declared as a data class it indicates that its main purpose is being a data structure without behavior.

Data classes can also be used for encapsulating one property. This can become valuable in cases when the exact details cannot be inferred from the type:

```
1 val timestamp: Long = getTimestamp()
```

In the above example the type Long does not show if the timestamp is in the format of milliseconds, seconds or even something else. Creating a data class which holds this timestamp solves this problem:

```
1 data class TimestampInMilliseconds(val value: Long)  
2 // ...  
3 val timestamp: TimestampInMilliseconds = getTimestamp()
```

Now the exact format of the timestamp can be inferred just based on the type.

```
1 data class TimestampInMilliseconds(val value: Long)  
2 // ...  
3 val timestamp: TimestampInMilliseconds = getTimestamp()
```

Data classes come with an additional benefit of automatically generating such function as equal, hashCode and toString. In Java these functions have to be written by hand or generated by the code editor. These functions are used for comparison of different instances of the class or help with string representations (e.g. logging). Another powerful function which is generated for data classes is the copy function, which helps preserve immutability, it can be used like this:

```

1 data class Person(
2     val firstName: String,
3     val lastName: String,
4     val age: Int
5 )
6 // ...
7 val teenager = Person("John", "Doe", 17)
8 println(teenager) // Person(firstName=John, lastName=Doe, age=17)
9 val adult = teenager.copy(age = 18)
10 println(adult) // Person(firstName=John, lastName=Doe, age=18)

```

The copy function, as its name suggests, copies the properties of a data class and allows for their modification. The result of the copy function is a complete new data class instance. However the copy function only makes a shallow copy, meaning that if the person class contained a list, the result of the copy would point to the same list reference (of course if the list is not changed with the copy function).

Thanks to data classes there is a clear separation between normal classes and data structures making it harder to accidentally put complex logic into a data structure.

### 1.3.8 Objects

In Kotlin objects are treated as Singletons [14] meaning that only one instance of it exists through the lifetime of a program. Because only one instance exists, they are better suited for encapsulating utility functions. For example:

```

1 object MathUtility {
2
3     fun isPrimeNumber(number: Int) {
4         // ...
5     }
6 }

```

Using objects as a replacement for normal classes is not a good idea, because of the fact that the same instance is shared with every client making its behavior potentially unpredictable. Additionally objects cannot be replaced when performing testing. For example if a database connection would be declared as an object, the test could not replace it with a fast alternative. It is better to leverage dependency injections (discussed in section 2.3) in order to provide a database instance. The real program uses the actual database but the test can use a fast alternative suited for testing purposes.

### 1.3.9 Sealed class

Sealed classes build upon the concept of enum classes, meaning that they represent a limited set of possibilities. For example, representing a Level of severity with three values (low, medium, high) with sealed classes can look like this:

```
1 sealed class Level {  
2     object Low : Level()  
3     object Medium : Level()  
4     object High : Level()  
5 }
```

The Level is represented by one of three defined types. The declaration above is logically equivalent to an enum class because all three Level types do not hold any data. Sealed classes, however, have an upper hand because every item from the limited set can have its own data:

```
1 sealed class Result {  
2     data class Success(val person: Person) : Result()  
3     data class Failure(val error: Exception) : Result()  
4 }
```

The Result sealed class has two possible types which define a success and a failure. The interesting part is that both of these types contain different data: The **Success** type contains a person; The **Failure** type contains an error which occurred. This pattern of defining a sealed Result class is often used when fetching data from an outside service, for example an API.

Sealed classes can also be used in order to represent the state of the User Interface (UI):

```
1 internal sealed class State {  
2     object Loading : State()  
3     data class ShowingPerson(val person: Person) : State()  
4     object ShowingError : State()  
5 }
```

The UI can be in one of three states. At first it will be in the loading state. Depending on the outcome, the next state could be either showing a person's information or showing an error which indicates what went wrong. With an error the user will probably want to retry the request which would change the state to Loading again.

### 1.3.10 Extension functions

Extension functions provide a way for extending the functionality of a class without having to change its internals or inheriting from it. They are useful in situations where the class is defined in a place where it cannot be changed e.g. a library or a standard type:

```

1 fun Int.isOddNumber(): Boolean {
2     return this % 2 == 1
3 }
4
5 val isOdd = 5.isOddNumber()
6 print(isOdd) // true

```

In the above example **this** refers to the actual value of the Int, which in that case was 5. Extension functions are a great way of providing utility functions.

A lot of the Kotlin standard library functions are declared as extension functions. The operator functions shown in the immutability section 1.3.1 like forEach, filter and map are all defined as extension functions.

### 1.3.11 Higher-order functions

Kotlin allows for functions to be saved as variables, meaning that they can be passed as parameters or returned from functions, thus enabling so called higher-order functions. These types of functions are popular in the functional programming paradigm, some of which have already been shown in the immutability section 1.3.1 through the forEach, filter and map operator functions. As a reminder, the filter function creates a new list based on an existing list, but the new list only contains elements which satisfy a given condition. For example, filtering a list of numbers from 1 to 5 with a condition to only allow numbers that are greater than three looks in the following way:

```

1 val filteredList = listOf(1, 2, 3, 4, 5)
2         .filter { number -> number > 3 }
3 println(filteredList) // [4, 5]

```

Creating such a filter function could look like this:

```

1 fun List<Int>.numberFilter(predicate: (Int) -> Boolean) : List<Int> {
2     val newList = mutableListOf<Int>()
3     this.forEach { number ->
4         if (predicate(number)) {
5             newList.add(number)
6         }
7     }
8     return newList
9 }
10
11 val filteredList = listOf(1, 2, 3, 4, 5)
12     .numberFilter { number -> number > 3 }

```

```
13 println(filteredList) // [4, 5]
```

The **numberFilter** is a higher-order function because it takes in another function as a parameter.

Recalling the last example from the immutability section 1.3.1 it is possible to mitigate the number of lists created by reducing the number of operators in the following way:

```
1 fun List<Int?>.filterAndMapNonNullNumbers(  
2     predicate: (Int) -> Boolean,  
3     transform: (Int) -> Int  
4 ) : List<Int> {  
5     val newList = mutableListOf<Int>()  
6     this.forEach { number ->  
7         if (number != null && predicate(number)) {  
8             val numberAfterTransformation = transform(number)  
9             newList.add(numberAfterTransformation)  
10        }  
11    }  
12    return newList  
13 }  
14  
15 fun example() {  
16     val input: List<Int?> = listOf(1, 2, 3, null)  
17     val result = input.filterAndMapNonNullNumbers(  
18         predicate = { number -> number % 2 == 1 },  
19         transform = { number -> number * 10 }  
20     )  
21     print(result) // [10, 30]  
22 }
```

Thanks to the **filterAndMapNonNullNumbers** higher-order function, only one list is created in order to output the result.

Higher-order functions help with abstracting away unnecessary implementation details e.g. the map and filter functions. Additionally they help with the reduction of boiler plate code e.g. callbacks can be simple functions instead of an interface. The most interesting aspect however is that they allow for the creation of Domain specific languages (DSL in section 1.3.12) in Kotlin.

### 1.3.12 DSL

Domain specific languages are, in principle, programming languages written in another programming language. The combination of extension and higher-order functions allow for creating type

safe DSLs in Kotlin. This is a very broad and complex topic so there will not be any implementation details presented here, only examples of using a DSL. The most straightforward example would be building an HTML document in Kotlin:

```
1 html {
2     head {
3         title {
4             + "Page\u201dtitle"
5         }
6     }
7     body {
8         h1 {
9             + "Building\u201dHTML\u201dtags\u201dwith\u201dKotlin"
10        }
11        p {
12            b(text = "Bold\u201dparagraph")
13        }
14    }
15 }
```

Like in the example above DSLs can be used for representing complex and nested data structures. Some libraries provide a DSL in order to use them. For instance, the Ktor server library discussed in section 2.4 provides a DSL for declaring server routes:

```
1 routing {
2     get("/note") {
3         // ...
4     }
5     post("/note") {
6         // ...
7     }
8 }
```

### 1.3.13 Coroutines

Coroutines are an officially supported way of handling synchronous programming in Kotlin. *Asynchronous* means that this action should be done on a background thread in order to not disturb the user by freezing or staggering the UI. Reading from a database or fetching something from an API are considered asynchronous actions.

I will not discuss in detail how the Kotlin coroutines work but I will touch the main aspects. In my opinion the most important concept of coroutines are suspending functions:

```

1 suspend fun fetchData(): List<Int> {
2     delay(3000)
3     return listOf(1, 2, 3)
4 }
```

The above suspending function returns the list after 3 seconds pass. Because the **delay** function is also a suspending function it results in the fact that the **fetchData** function will only resume after the **delay** completes which happens after 3 seconds. Suspending functions allow for programming asynchronous flows in a manner which is almost the same as synchronous flows:

```

1 suspend fun getA(): Int {
2     delay(1000) // Simulated network call
3     return 3
4 }
5 suspend fun getB(): Int {
6     delay(1000) // Simulated network call
7     return 2
8 }
9 suspend fun getResult(): Int {
10    val a = getA()
11    val b = getB()
12    return a + b
13 }
```

The **getResult** function returns the number 5 after 2 seconds. It is easy to imagine situations in which using suspending functions supports the clarity of the code.

Suspending functions can only be executed inside a coroutine. To keep this section brief I will not go into the inner workings of coroutines and just show an example with a simple explanation:

```

1 GlobalScope.launch {
2     val result = getResult()
3     println(result) // prints 5 after 2 seconds
4 }
```

The above code creates (launches) a coroutine in the **GlobalScope**, meaning that the coroutine can potentially be active through the whole lifetime of a program. **GlobalScope** should be avoided in most cases, but it was used for the sake of this example.

Another big feature of Kotlin coroutines are flows which can provide a way for creating finite or infinite streams of data:

```

1 val intFlow = flow {
2     (1..3).forEach { number ->
```

```

3         delay(1000)
4         emit(number)
5     }
6 }
```

In the above example the numbers 1 through 3 are emitted from the flow with a 1 second delay between the emissions. In order to get values from a flow they need to be collected inside a coroutine:

```

1 GlobalScope.launch {
2     intFlow.collect {
3         //After 1 seconds prints "1"
4         //After 2 seconds prints "2"
5         //After 3 seconds prints "3"
6         println(it)
7     }
8 }
```

Besides emitting data, flows can also be used as a wrapper for callbacks. For example, converting a button click callback to a flow could be implemented like this:

```

1 val buttonClickFlow = callbackFlow {
2     button.onClick {
3         offer(Unit)
4     }
5 }
```

The **callbackFlow** is a special type of a flow which should be used for callbacks. Every time a user clicks the button, its callback emits a **Unit** (empty) value to the flow. Such a flow could be combined with an operator that prevents the button from being clicked multiple times in a given span of time and consequently preventing from accidental double clicks.

Both of the above flows are so called **cold** flows meaning that they will only start emitting items when someone starts collecting them. There is another kind of a flow which emissions are independent of the fact if they are being collected. Such flows are called **hot** flows. One example of them is the **StateFlow**:

```

1 fun createStateFlow(): StateFlow<Int> {
2     val mutableStateFlow = MutableStateFlow(0)
3     GlobalScope.launch {
4         (1..3).forEach { number ->
5             delay(1000)
6             mutableStateFlow.value = number
7         }
8     }
9 }
```

```

7         }
8     }
9     return mutableStateFlow
10 }
11
12 val stateFlow = createStateFlow()
13 GlobalScope.launch {
14     stateFlow.collect { number ->
15         //Instantly prints "0"
16         //After 1 seconds prints "1"
17         //After 2 seconds prints "2"
18         //After 3 seconds prints "3"
19         println(number)
20     }
21 }
```

The convention of using the **Mutable** prefix is also preserved here. This mutable version of the state flow allows for changing its value. The return type of the **createStateFlow** is the immutable StateFlow meaning that its value cannot be changed outside the **createStateFlow** function.

When the StateFlow is collected, it emits its current value. At start it is 0 but changes every second for 3 seconds ending with the value 3. Because StateFlow is a **hot** flow its values are emitted even when no one is collecting them:

```

1 val stateFlow = createStateFlow()
2 GlobalScope.launch {
3     delay(5000)
4     stateFlow.collect { number ->
5         //Instantly prints "3"
6         println(number)
7     }
8 }
```

The above example only prints the number 3, because the flow is collected after 5 seconds meaning that the emissions of the value changes happened before the collecting.

StateFlow are useful in situations where the state is modified in an asynchronous manner and there is a need to retrieve its last value. One such situation could be showing data from a database in the UI. Even if retrieving the data completes faster than showing the UI, StateFlow guarantees that the UI will get its latest value.

## 1.4 Kotlin Multiplatform

Kotlin multiplatform is an alpha feature of the Kotlin ecosystem. *Alpha* means that this product will be developed further but breaking changes can be introduced with updates. It allows for sharing code between different platforms. The code is written in Kotlin, then it is compiled or transpiled to the desired platform (explained later in this section).

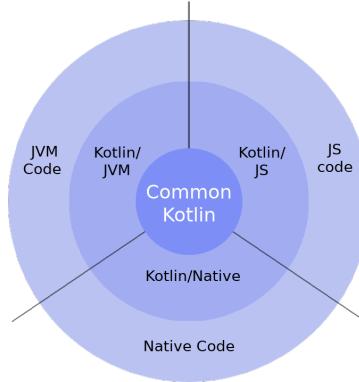


Figure 3: Kotlin Multiplatform library and compilation target division, source:  
<https://kotlinlang.org/docs/reference/multiplatform.html>

The inner circle in Figure 3 is the Common Kotlin Module. It contains the language core along with the core libraries. This core does not depend of any specific platform and works everywhere.

The middle circle consists of platform-specific Kotlin versions which are built on top of the Common module. They extend the core libraries and provide additional help for that specific platform. For example, the Kotlin/JVM helps Kotlin interoperate with Java, the Kotlin/JS enables Kotlin to call JavaScript specific functions.

The outer circle is a platform specific code that is written in the platform native code. This layer usually consists of the platform core and libraries which are written for that specific platform.

Kotlin/JVM is compiled to byte code which is the same thing that Java does, however Kotlin/JS gives Kotlin the ability to be transpiled into JavaScript. Transpilation is the act of compiling source code from one programming language to another. In this case Kotlin code will become JavaScript code. This transpiled Kotlin code will act and behave just as any other JavaScript code. Kotlin/JS also provides a standard library, which gives Kotlin the ability to interact with the browser. The interaction between Kotlin and JavaScript will be explained in the coming sections.

In my project I am focusing only on the JVM and JS parts of Kotlin Multiplatform. However, the most popular combination is JVM and Native which allows for sharing code between the Android and iOS mobile platforms.

## **1.5 Proof of concept**

The project I am creating for the sake of my MSc thesis can be thought of as a proof of concept. It is worth noticing that it is not a full fledged application. It is a sample of what can be done with the Kotlin language. I have tried to create an application which is easy to be understood and has a shared functionality between the Android app and Web app. In my project this functionality mostly consists of making network requests, using a local database (e.g Sqlite database in Android) and operating on the app main data, which are notes.

## **1.6 Open source**

Since Kotlin and most of the libraries used in this project are open source I have wanted to give back to the community, hence this project is available on GitHub [15] at AKJAW/fuller-stack-kotlin-multiplatform and is accessible to everyone.

## 2 The project

The application allows for taking and saving notes. The notes are persisted locally and on the server, what allows for offline use. In order to use the app every user has to be authenticated. Thanks to this authentication the notes can be synchronized between multiple devices. When the user opens the app simultaneously on different devices and has internet access, the synchronization happens in real-time. This means that adding a new note on one device is visible on other devices after a short delay. There are also operations which do not interact with the API or the database such as filtering, searching the notes and changing their date format representation.

### 2.1 Project structure

The project is divided into 4 main modules shown in Figure 4.

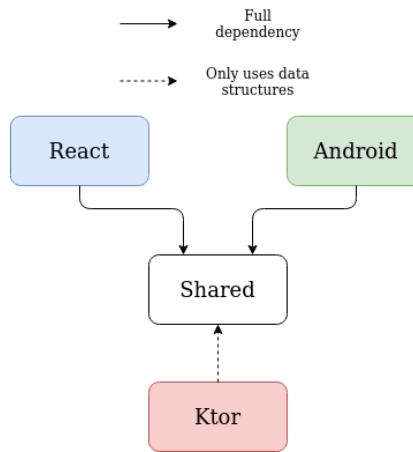


Figure 4: Module division with dependency arrows

The React and Android module contain code for the UI of their respective platforms (Web and Android) and all other platforms specific code. Both of these modules depend on the Shared module, which is a Kotlin Multiplatform module. The shared module contains all of the data structures used in this project along with logic for the shared app features i.e. user input validation, database synchronization, networking, note operations.

The Ktor module contains the back-end (interchangeably called the server) that both of the client apps (Android and React.JS) use. It has a dependency on the Shared module but uses only the data structures from that module, no other logic is used. Unfortunately, even though this module does not need anything besides the data structures, it still needs to compile the whole Shared module. The data structures could have been duplicated between the Shared and Ktor module, which could cause problems when either of them gets out of sync. The Shared dependency in the Ktor module guarantees that all of the modules have up to date data structures.

### 2.1.1 Gradle [16]

Gradle is a build automation tool mostly used for JVM projects, but has a support for other environments. Kotlin provides a special plugin for Gradle which enables support for Kotlin Multiplatform projects. From a higher perspective Gradle divides projects into modules and manages their dependencies. Either from other internal modules in the projects or external online sources.

Figure 4 shows that three modules: React; Android and Ktor all depend on the module Shared. The three modules could be thought of as sub-projects because they compile to different environments and do not depend on each other. Dependencies are composed in a tree like structure. Figure 5 shows the client platform modules and the modules they depend on.

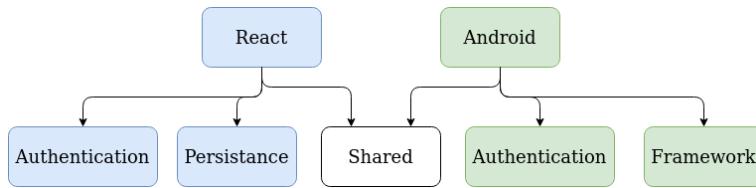


Figure 5: Gradle dependencies for the client platforms with dependency arrows

## 2.2 GitHub Actions

GitHub Actions are a CI/CD (Continuous Integration / Continuous Delivery) tool that is integrated into GitHub repositories. GitHub Actions use so called workflows which could be used as CI, CD or both.

Continuous Integration is a practice of making frequent and small code changes. The workflows provided by GitHub Actions enable a way for automating the process of checking if the code is correct. These checks usually consist of compiling the code, running tests and other actions. Thanks to this automation any errors or bugs can be caught early into the development cycle.

Continuous Delivery automates the deployment/release process. This project does not use CD so it will not be discussed in details.

This MSc projects GitHub Actions have the following workflows:

- Build all of the platforms in order to verify that there have been no breaking changes introduced;
- Running tests for the Android app and the Shared code;
- Check the coding standards with the help of static code analysis tools.

These workflows are run every time a code change occurs in the GitHub repository ensuring that if something breaks it is caught early.

### 2.2.1 Static code analysis

It is an analysis which is performed without executing the code. Usually it is performed by an automated tool. This MSc project uses two static analysis tools, i.e.:

- Ktlint - checks the formatting of Kotlin code and notifies of any discrepancies in the coding standard. Additionally it has a built in formatter which can be used to automatically format Kotlin files;
- Detekt - checks the complexity of the code, for example it detects long function names or large blocks of code and then suggests what can be done in order to reduce the complexity.

## 2.3 Dependency Injection

In Object Oriented Programming paradigm dependency injection is the act of passing dependencies (classes) to clients (other classes). Usually these dependencies contain behavior which is used to extend the capability of the client. This results in using the composition over inheritance pattern [17]. In brief, instead of inheriting behavior it should be contained in separate classes which are used by the client. Dependency injection also divides the responsibilities of classes, either a class creates other classes, or it contains a behavior logic and/or a state. Without this separation classes would be responsible for both of these things, which is contradictory to the single responsibility principle [18]. This principle says that a class should only have one responsibility or just one reason to change.

Dependency Injection can be done manually by creating classes whose behavior consist of creating other classes or it can be done by using frameworks/libraries. On Android the most popular framework for dependency injection is Dagger 2. It eliminates the need for manually creating classes, because all of the creation classes are automatically generated during compilation. Unfortunately, Dagger 2 is only available on the JVM, which disqualifies it from this project because it reduces the capabilities of code sharing.

The cross-platform solution used in this project is Kodein [19]. It is not a dependency injection framework but a service locator. It still creates the dependencies but in a different way. To put it simply, dependency injection frameworks give the dependencies and service locators provide dependencies when asked by a client. The difference can be seen in Figure 6

Here is the general idea of how dependency injection is done with Kodein:

```
1  class Database() {  
2      // ...  
3  }  
4  
5  class GetNotes(private val database: Database) {  
6      // ...
```

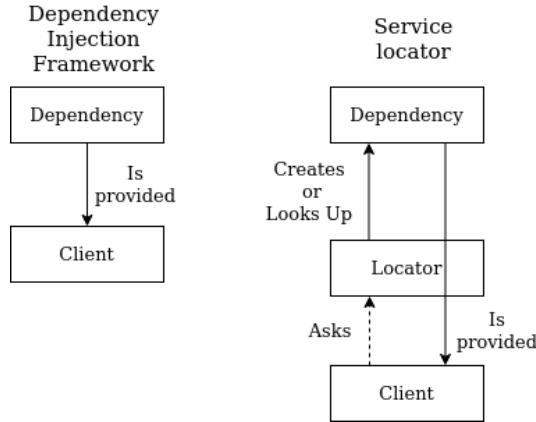


Figure 6: Simplified illustration of the difference between Dependency Injection Frameworks and Service Locators

```

7  }
8
9 class Client(private val getNotes: GetNotes) {
10    // ...
11 }
12
13 val module = DI.Module("Module") {
14     bind() from singleton { Database() }
15     bind() from singleton { GetNotes(instance()) }
16     bind() from singleton { Client(instance()) }
17 }
```

The module takes care of creating the classes and the **GetNotes** and **Client** classes just use what is injected in the constructor. Most of the dependencies can be injected through the constructor like in the example above, but sometimes the developer is not in charge of creating the classes e.g. an Android screen which is created by the system. In these cases the injection cannot happen in the constructor so it happens in the class properties by calling a Kodein function:

```

1 class Screen : DIAware {
2     private val getNotes: GetNotes by instance()
3 }
```

The **DIAware** interface enables the use of Kodein in the **Screen** class. Dependency injection happens through the **instance** function which searches through the dependency graph until it finds the correct class and provides it to the **Screen** class.

## 2.4 Ktor server

The server enables the apps to be fully cross-platform, as a user can use multiple devices without losing access to their data. The Ktor server was build for the JVM environment meaning that it leverages Kotlin/JVM (described in details in Android section 2.7)

### 2.4.1 PostgreSQL Database

This database, later called a remote database, contains the users notes. The notes kept in this database are a reflection of the notes stored in the applications local database. When a note is added locally it is also added remotely to this database etc. This allows the user to change devices and have up to date notes in their applications.

#### 2.4.1.1 Note Schema

This data structure is used for the remote database and also for the network calls in both of the apps. These are the most important fields:

- Title - the title of the note;
- Content - the body of the note;
- LastModificationTimestamp - the unix timestamp of last modification. This is used for comparing which version of the note should be kept during synchronization. The newer LastModificationTimestamp always overwrites the older one;
- CreationTimestamp - the unix timestamp of the creation time. This is used as the unique note identifier when synchronizing the notes locally. Normally this kind of an identifier is discouraged but for this project it will suffice;
- WasDeleted - a flag which indicates if the note was soft-deleted. The note still exists in the remote database but from the user's perspective it is deleted because they cannot see it.

Additionally, the remote database also has a field used for user identification.

### 2.4.2 Authentication

Authentication is handled by a third-party platform called Auth0 [20] (logo shown in Figure 7). The platform provides integration for multiple platforms including Android and React.JS. Delegating user authentication to a third-party means that the developer has fewer things to be concerned about like developing their own integration and security concerns.

Auth0 in the client apps allow the user to sign in using their email or Google [21] account. The authentication flow is as follows for both platforms:

1. User clicks the sign-in button in the app.
2. A new website is open (redirected in case of the web app). This website allows the user to input their credentials and sign into the app.



Figure 7: Auth0 is a secure and universal service for user Authentication  
<https://auth0.com/>

3. If the credentials are correct, the user is authenticated and redirected back to the app.

This whole flow is handled by Auth0, which means that the Ktor back-end is not involved. However, every call to the back-end after the authentication requires a subsequent authentication because every note is tied to a user and the server needs to differentiate between users notes.

When the apps are sending requests to the back-end they also send a JSON Web Token (JWT) [22] which is used by the server to obtain the user ID. This token is provided by the Auth0 libraries used on the client side. JWTs are widely used for authentication. Ktor provides an extra package for their support which means it does not need Auth0 or any other third-party dependencies for user authentication.

#### 2.4.3 Websocket

Websockets are a computer communications protocol allowing two-way communication between the client and the server. This protocol is used to give the user real-time updates about their notes which is illustrated in Figure 8. Ktor provides an additional package to create websocket sessions.

##### 2.4.3.1 Client

Every client app at a startup connects to the socket endpoint and sends the user's ID in order to identify and send updates to the correct user. In addition to the user ID, cookies are used to differentiate users sessions. As an example, if a user opens two browser windows, changes from one window are sent to the other window and the same occurs for other devices.

##### 2.4.3.2 Server

The back-end listens to socket connections and when a connection is established, the client sends the user identifier which is the only thing the client sends through the websocket. If the user is valid the connection is saved along with the session. After the client makes a change in the notes, the websocket notifies the clients but only the sessions which did not make the change. This prevents

unnecessary data transmission because the source of the changes does not need to be notified of them.

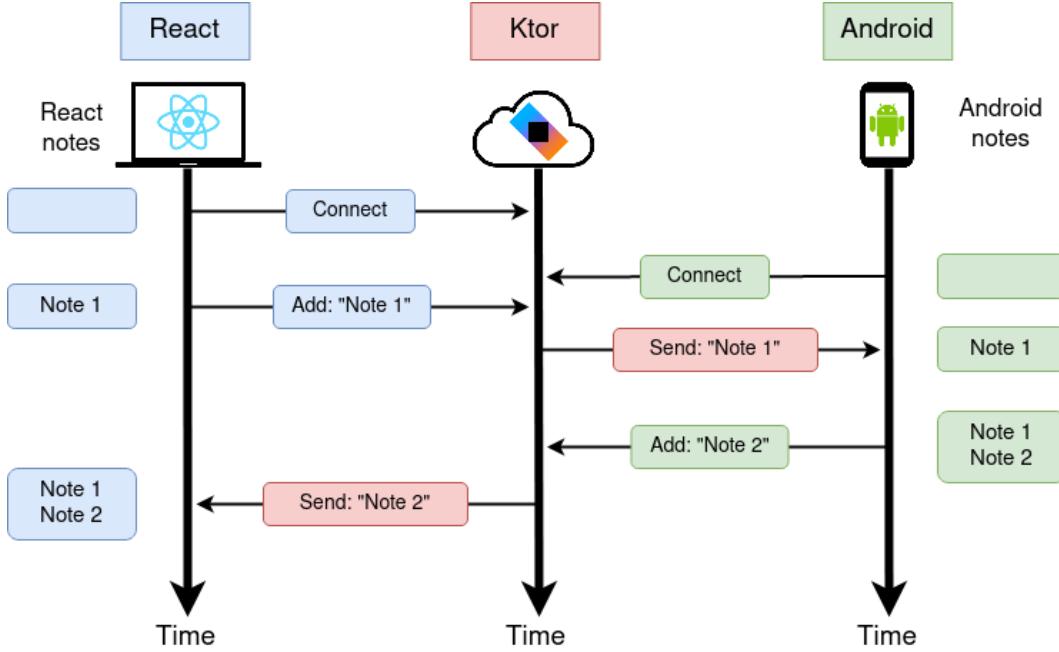


Figure 8: An example websocket exchange between an Android and React application.

[https://www.iconfinder.com/icons/2205195/mobile\\_phone\\_screen\\_smart\\_icon](https://www.iconfinder.com/icons/2205195/mobile_phone_screen_smart_icon)

[https://www.iconfinder.com/icons/171511/laptop\\_computer\\_icon](https://www.iconfinder.com/icons/171511/laptop_computer_icon)

[https://www.iconfinder.com/icons/370088/cloud\\_weather\\_cloudy\\_winter\\_icon](https://www.iconfinder.com/icons/370088/cloud_weather_cloudy_winter_icon)

#### 2.4.4 Technologies used

- Ktor:
  - Server core;
  - Server netty - allows for Ktor to use Netty as the server engine;
  - Auth JWT - adds support for JSON Web Token;
  - Websockets;
  - Serialization - allows for seamless JSON parsing for incoming and outgoing data.
- Kodein - cross-platform dependency injection;
- Klock - cross-platform Date and time library;
- Kotlin coroutines - enables coroutines support in Kotlin;
- Exposed - allows for the use of databases;

- PostgreSQL JDBC Driver - enables the server to connect to a PostgreSQL database;
- H2 Database - database engine used for running an in-memory database for development purposes.

#### 2.4.5 Hosting



Figure 9: Heroku is a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud

Back-end hosting is provided by Heroku [23] (logo shown in Figure 9) which is a platform for building and deploying applications. It supports multiple programming languages and frameworks, including Java and Kotlin applications such as Ktor. Besides the back-end Heroku also hosts the remote PostgreSQL database.

Heroku has a great documentation and a big community which makes finding help easier. It also offers a generous free plan which is perfect for side-projects or university projects like the one for this MSc thesis. The only requirement is an account which can be created without any payment.

## 2.5 Shared module

As stated in the Introduction, the shared module contains shared logic and data structures for the client platforms (Android and React). The platform modules depend on the shared module the most (please refer to Figure 4 and 5).

### 2.5.1 Structure

The shared module consists of other sub-modules, Figure 10 shows how it looks for this project.

Each sub-module is grouped by the platform:

- Common contains code that is written in common Kotlin (refer to Figure 3) ensuring that it can be used by all other sub-modules;
- React contains code for the Kotlin/JS environment;
- Android contains code for the Kotlin/JVM environment.

The platform groups are also split into two:

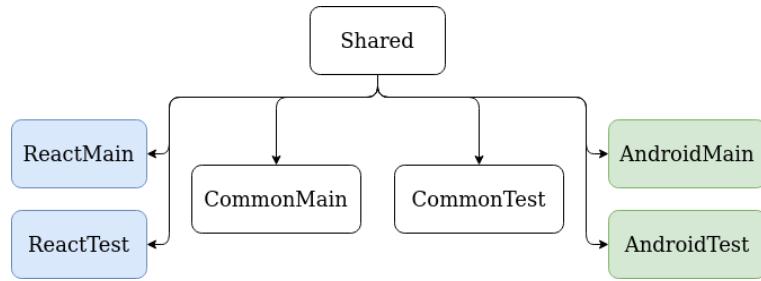


Figure 10: Sub-modules located in the shared module

- Main suffix - contains code that other modules use;
- Test suffix - contains code for testing the corresponding “Main” sub-module.

The client modules for the apps (Android, React) have only a dependency on the shared module. The sub-modules are irrelevant because depending on the platform, only the corresponding will be included.

### 2.5.2 How the code is shared

The following sections will describe two ways which can be used to share code: The “Expect and Actual declaration” and “Dependency inversion”.

#### 2.5.2.1 The Expect and Actual declaration

The declaration is a Kotlin Multiplatform specific feature. It works for most of Kotlin declarations such as:

- Functions;
- Classes;
- Interfaces;
- Properties;
- Annotation.

This mechanism works as follows: The Common sub-module “expects” that the declaration exists and the platform sub-modules provide the “actual” declaration. One can imagine a situation where the shared common sub-module needs a function to log a message but the platforms have different ways of doing it. Using the expect and actual declaration, the implementation could look like this:

```

1 //CommonMain
2 expect fun logMessage( message: String)

```

```

1 //ReactMain
2 actual fun logMessage( message: String) {
3     console.log(message)
4 }

1 //AndroidMain
2 actual fun logMessage( message: String) {
3     println(message)
4 }

```

Thanks to this declaration the use of the logMessage function is the same for every module but their internal implementation is different depending on the platform. Figure 11 shows a simplified diagram of the declaration.

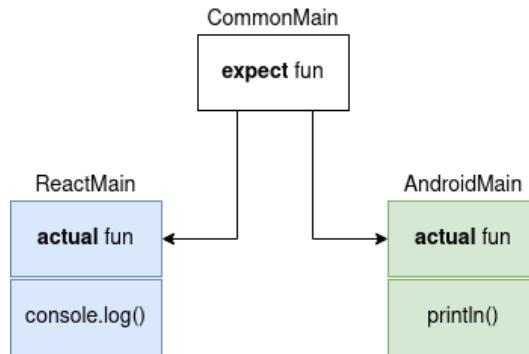


Figure 11: Expect and actual diagram for the logMessage function

The expect and actual declaration can also be used for data structures. Imagine a situation where the common code uses a database but the platform specific database implementation requires a different convention for their entities:

```

1 //CommonMain
2 expect class ItemEntity {
3     val id: Int
4     val name: String
5 }

1 //ReactMain
2 actual data class ItemEntity(
3     actual val id: Int,
4     actual val name: String
5 )

```

```

1 // AndroidMain
2 import androidx.room.Entity
3 import androidx.room.PrimaryKey
4
5 @Entity(tableName = "items")
6 actual data class ItemEntity(
7     @PrimaryKey(autoGenerate = true) actual val id: Int,
8     actual val name: String
9 )

```

The Android module uses the Room persistence library [24] which requires additional annotations such as the database table name and the primary key, on the other hand the React does not need them. Thanks to the expect and actual declarations, this data structure can be used as one and the same independently of platform specific implementations. This exact situation happened when working on this MSc thesis (Data structures 2.5.3.2).

### 2.5.2.2 Dependency inversion

Dependency inversion [25] is an object oriented design which allows for better decoupling between modules. It also promotes better design by forcing classes to depend on abstraction instead of implementation. With dependency inversion the shared module is able to use outside dependencies which are hidden behind abstraction, while the expect and actual declaration does not (without using dependency inversion).

The logMessage example from the expect and actual declaration section can be visualised like in Figure 12. The Client class uses the logMessage function directly but internally it maps to the platform specific implementation.

When a module needs something from another module it depends on that module. In Figure 12 the React and Android modules depend on the Shared module in order to use the function. If the Shared module needs something from the React or Android module it cannot simply depend on them because that would lead to circular dependency shown in Figure 13. As the red crosses suggest this does not work and the code will not compile.

To allow the shared module to use something from an external module, dependency inversion shown in Figure 14 can be applied. The platform specific implementation does not reside in the Shared module anymore which allows the ConsoleLogger and PrintLogger classes to use other classes from their respective modules without creating a circular dependency. All clients use the Logger interface but behind their abstraction there is a platform specific implementation.

```

1 // CommonMain
2 interface Logger {
3

```

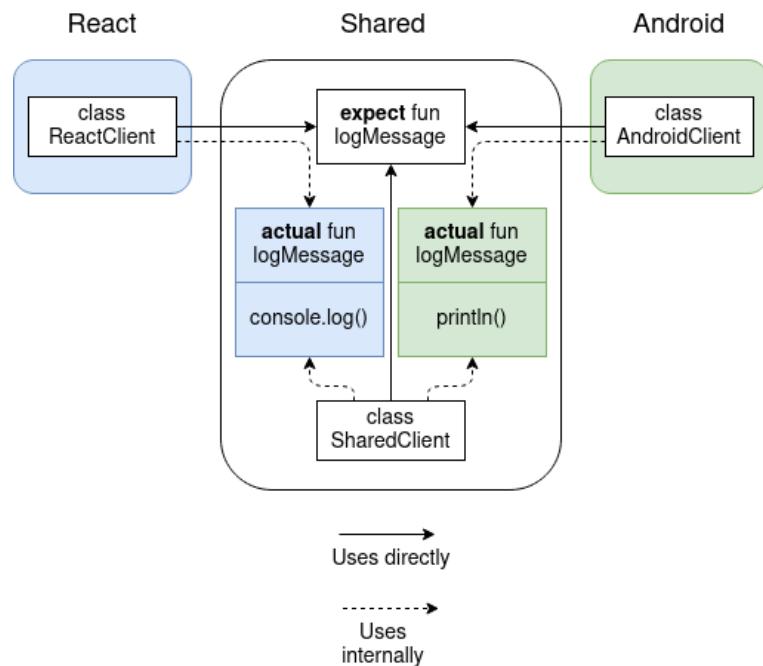


Figure 12: A diagram of expect actual declaration from the perspective of modules

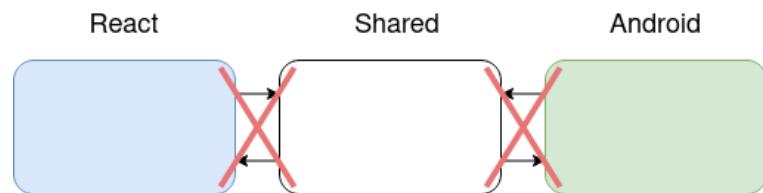


Figure 13: A diagram of circular dependencies between of modules

```

4     fun logMessage(message: String)
5 }

1 //React
2 interface ConsoleLogger : Logger {
3
4     override fun logMessage(message: String) {
5         console.log(message)
6     }
7 }

1 //Android
2 interface PrintLogger : Logger {
3

```

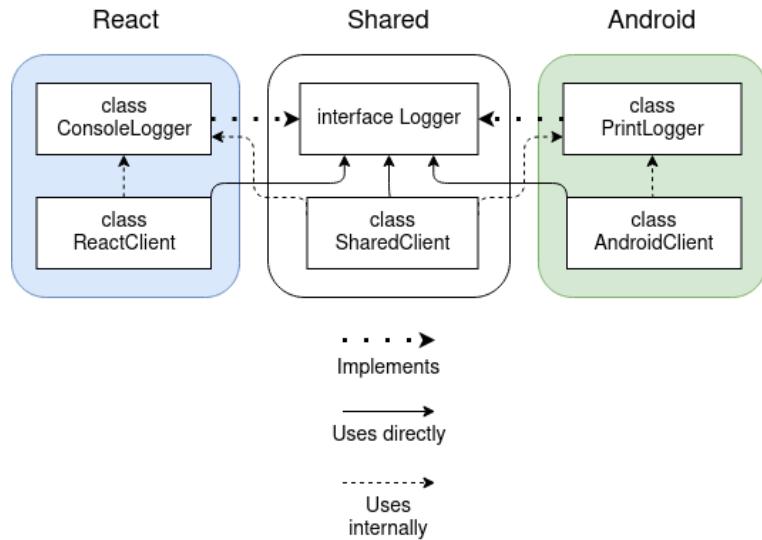


Figure 14: A diagram of dependency inversion allowing shared module to have an indirect dependency on external modules

```

4     override fun logMessage(message: String) {
5         println(message)
6     }
7 }
```

This technique is possible thanks to dependency injection described in section 2.3. All client have a dependency on the `Logger` interface without knowing about its internal implementation, which is injected into the clients constructor but are hidden behind an abstraction.

```

1 //CommonMain
2 class SharedClient(private val logger: Logger) {
3     ...
4 }

1 //React
2 class ReactClient(private val logger: Logger) {
3     ...
4 }

1 //Android
2 class AndroidClient(private val logger: Logger) {
3     ...
4 }
```

### 2.5.3 Module contents

#### 2.5.3.1 Features

Most of the apps features reside in the shared module:

- Creating notes;
- Retrieving notes;
- Updating notes;
- Deleting notes;
- Synchronizing between the local and API notes;
- Real time note updates from the web socket;
- Sorting;
- Searching.

Bold letters above form CRUD what refers to the four basic persistent storage operations.

Most of these operations make use of the local database and the API which differ between platforms. The React and Android modules provide those implementations.

Sorting and searching is implemented using only the Common Kotlin module shown in Figure 3 which comes from the programming language itself.

#### 2.5.3.2 Data structures

The most important data structures are the ones used for representing the notes. Its data flow is shown in Figure 15.

- Note - representation of the note in the UI on both client platforms;
- NoteEntity - database representation of the Note;
- NoteSchema - network representation of the Note.

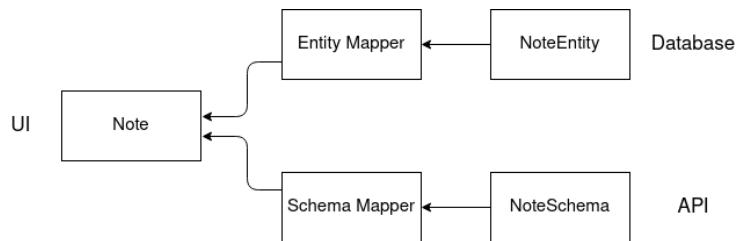


Figure 15: A representation of how the note data structure flows through the application

The database data structure uses the expect and actual declaration. This means that their implementation details are different between the client platforms but they behave in the same way.

The reason for this was described in section 2.5.2.1. The Android database data structures require additional information for the database to work correctly.

Other less important data structures include:

- Payloads - encapsulate data that is sent to the API;
- Structures with one value - This was discussed more in-depth in 1.3.7 section. For example, instead of using a number type for creation and last modification timestamp, they are represented by two data structures CreationTimestamp and LastModificationTimestamp.

```
1 data class LastModificationTimestamp(val unix: Long)
2 data class CreationTimestamp(val unix: Long)
```

#### 2.5.4 Architecture

Figure 16 shows a simplified model of the architecture of this project. Flow of control means that the view layer calls the presentation layer which then calls the domain layer and so on. The platform specific view and presentation layer will be briefly described here in an abstract manner and more in depth later in their respective chapters.

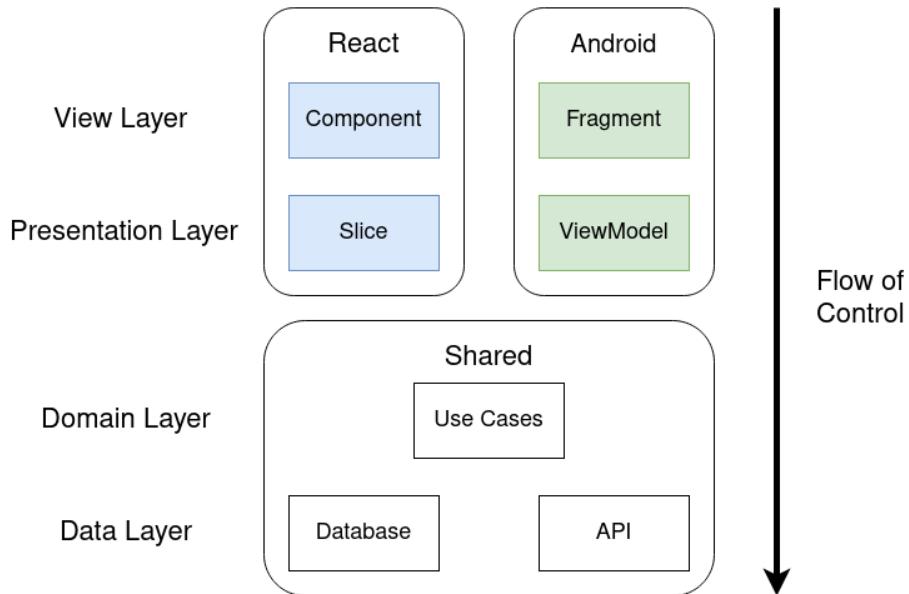


Figure 16: A model which shows the projects modules and their place in the architecture

##### 2.5.4.1 View layer

The view layer is responsible for showing the UI to the user and reacting to user input e.g clicking.

Usually most of the classes in this layer are coupled to a framework for example: Android or React.JS. The logic in this layer should only be related to showing the UI. Adding other logic to this layer makes it almost impossible to reuse in other layers. Additionally, this layer should only have a direct contact with the presentation layer meaning that the presentation layer orchestrates the UI related data flow.

#### 2.5.4.2 Presentation layer

The presentation layer orchestrates the data flow between the view layer and the domain layer. The view layer informs of a user (or a system) interaction which the presentation layer interprets and calls the domain layer. Most of the time the domain layer returns data which is then prepared before being shown in the view.

#### 2.5.4.3 Domain layer

The domain layer contains the “business logic” of the app (system). This logic defines how the app operates and what it can do. Most of the time the features of an app are the business logic. In case of this app most of the business logic is contained within use cases (sometimes called interactors).

Use cases usually represent a single user (client) interaction with the system, for example: the logic for updating a note is encapsulated inside a use case class called “UpdateNote”. All of the app features mentioned in the previous section, are represented by use cases.

From an architecture standpoint use cases help the codebase be more focused and follow the single responsibility principle [18]. Thanks to them the project structure is more clear, and just basing on use cases it is possible to predict what a system does. For example, given a set of use cases called: “GetNotes”, “AddNote” one can guess that a part of the system is responsible for managing notes.

The use cases used in this project are divided into two categories: "Synchronous" and "Asynchronous" shown in Figure 17. The asynchronous use cases contact outside systems like databases or APIs. In other words the asynchronous use cases have side-effects in the form of persisting data or retrieving them. The synchronous use cases are side-effect free.

Every use case is related to the main functionality of the app which are "Notes". Additional use cases like "SingUserIn", "SignUserOut" could have been made if the authentication flow behaved in a similar way on both the client platforms. Unfortunately the authentication libraries used vary vastly between the web and Android.

Use cases can also be composed of other use cases as is the case of the "SynchronizeNotes" use case, where it exploits other use cases underneath which is shown in Figure 18. This supports keeping the uses cases small or focused which, in turn, helps with the Single responsibility principle [18].

One of the most important things about the domain layer is that it should have no outside

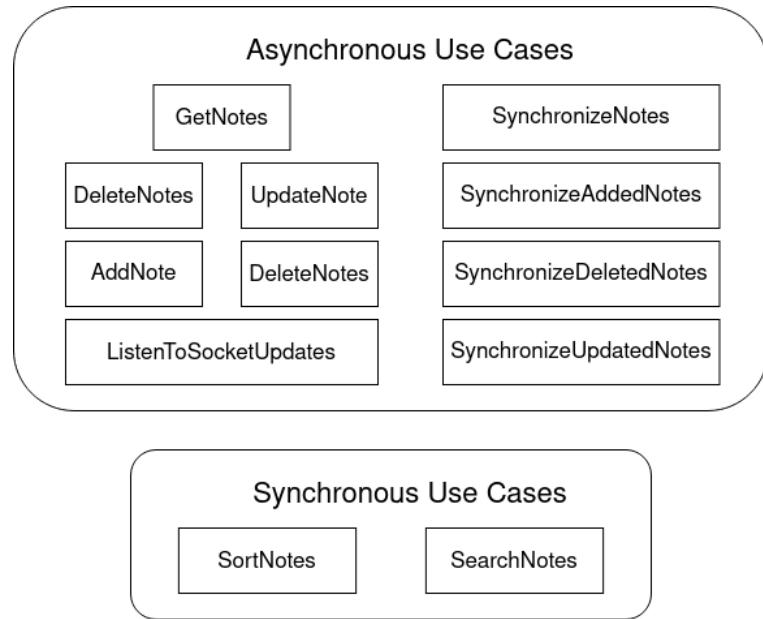


Figure 17: An overview of the use cases in the project

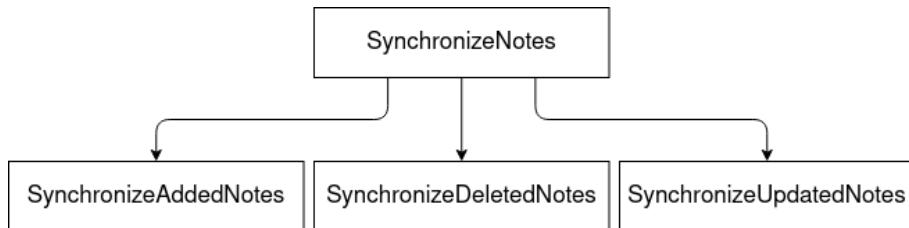


Figure 18: An example of how the "SynchronizeNotes" use case is composed

dependencies. The use of outside systems is possible thanks to the combination of dependency injection 2.3 and dependency inversion 2.5.2.2 shown in Figure 19. The use cases in this example use the database and the API but only through an interface which is defined in the domain layer.

#### 2.5.4.4 Data layer

The most notable things the data layer contains in this project are the data sources for the app, which in this case are a database, an API and a web socket. The interface for both of these data sources is defined in the domain layer, but their respective implementation resides in the client platforms shown in Figure 20

In this project besides the database and the API, there is an additional data persistence system which is used for settings. This persistence follows the same dependency rules as the previous data sources. The data source implementations on both platforms can be seen in Figure 21, the details of these implementation will be covered in the coming Android 2.7 and React 2.6 sections.

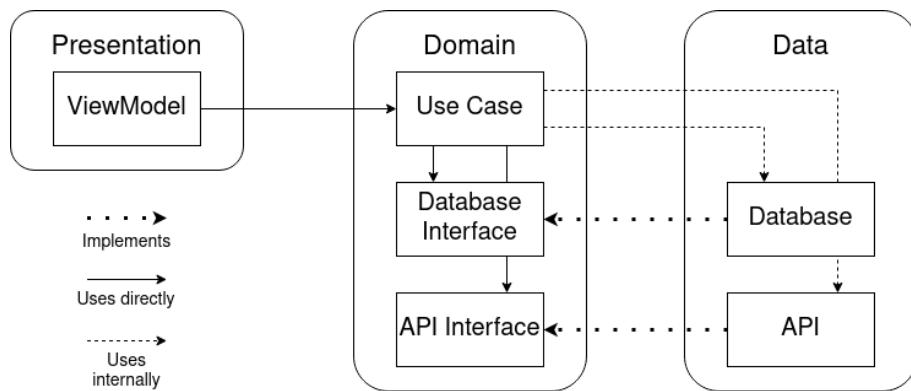


Figure 19: A simplified diagram of layer interaction on the Android client platform

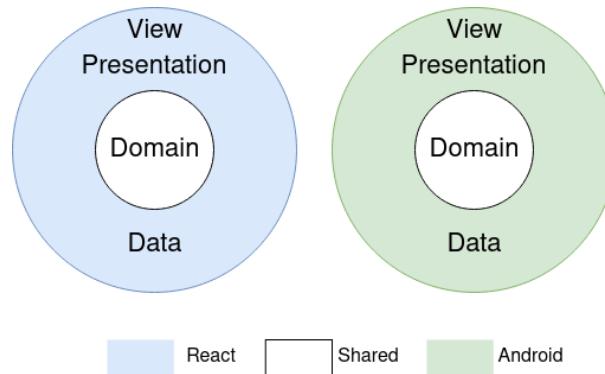


Figure 20: The layers depending on the client platforms

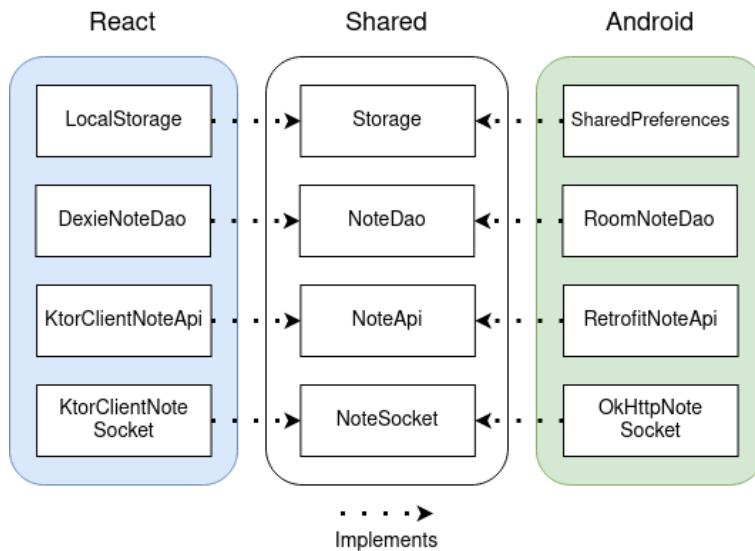


Figure 21: The data source implementations on both platforms

### 2.5.5 Testing

The shared module contains the business logic for both of the apps (system) and because of this the shared module contains the most unit tests. The features of the system are represented by use cases which are the most important part of the system. Therefore the test mainly focuses on the use cases which all have their own unit test.

Most of the features connect with the database and the API (both referred to as a storage later). Unit tests have to be fast and reliable which unfortunately is not possible with the real implementation. Therefore, they cannot be used in unit tests because they are neither fast nor reliable (e.g. bad internet connection). Instead of the real implementations a test double can be used, which can be achieved through a framework/library or written by hand e.g. fakes.

For simplicity sake when referring to mocking I mean using a framework/library for it. Mocking is a way of providing a test double and defining its behavior or state in the test. Every class has public properties and/or functions the clients can use. Both of these things make up the contract which describes how a class can be used. The most straightforward way of defining a contract is through an interface but classes without an interface also define a contract. Additionally, they allow for checking if a mocked property or a function were used. Unfortunately, because of the nature of mocking it is hard to implement a library such that it would work with Kotlin Multiplatform. At the time of writing the thesis there is no reliable way of creating mocks without implementing them by hand.

Another way of defining test doubles are fakes which usually work in a similar way as the real implementation which usually use an in-memory cache instead of a database or API. The downside is that fakes can only be implemented for interfaces. Because of the nature of Kotlin Multiplatform mocking is difficult so fakes were used instead.

In the Shared module test suite the most important fakes are the ones that represent the database and the API. They just hold the state in memory which makes them ideal for testing. Most of the tests just assert that the storage has the correct state after executing an action. An additional fake was created for providing a predefined timestamp in order to have consistency when operating with notes timestamps.

The framework which was used for writing the shared module tests is called Kotest [26]. It provides multiple ways of writing tests depending on the preference, however most of the testing styles leverage Koltins great support for functions. Additionally, Kotest provides an additional library for making test assertions which use Koltins capabilities for writing DSLs (section 1.3.12).

### 2.5.6 Technologies used

Because a lot of the implementations are provided by the client platforms which are only orchestrated by the shared module, libraries used in the shared module are sparse:

- Kotlin Coroutines [27] - provides asynchronous programming features to the Kotlin languages.

It is mainly used when communicating with outside systems like a database or the API;

- Klock [28] - library used for operating with notes timestamps and formatting them to dates;
- Kotlin Serialization [29] - provides JSON serialization for API requests and responses;
- Kodein [19] - allows multiplatform dependency injection.

## 2.6 Web app

The react app has the following screens:

- Sign in screen, figure 22 - the only screen an unauthenticated user sees. Clicking the button navigates the user to the Auth0 sign in page shown in figure 23;
- Home screen, figure 24 - the main screen of the app. It features the notes list and a note editor which can be opened and closed;
- Settings screen, figure 25 - this screen allows the user to sign out or change their date formatting preferences.

To use Fuller Stack you have to be authenticated

SIGN IN

Figure 22: React.JS app screen which asks the user to authenticate before using the app

### 2.6.1 Kotlin/JS

Besides the Kotlin/JS standard library, Kotlin leverages the existing JavaScript ecosystem and allows for the use of its libraries. A lot of popular libraries have a corresponding "Kotlin wrapper" which allows for a type safe and easier use from Kotlin. JetBrains [4] (with the help of the community) also provides a set of wrappers. These wrappers i.a. include React.JS [2] with helper libraries and Redux [30], both of which are used in this project.

#### 2.6.1.1 Calling JavaScript from Kotlin

Given the alert function in JavaScript which shows a pop-up with a message, one way of calling this function in Kotlin is inlining JavaScript:

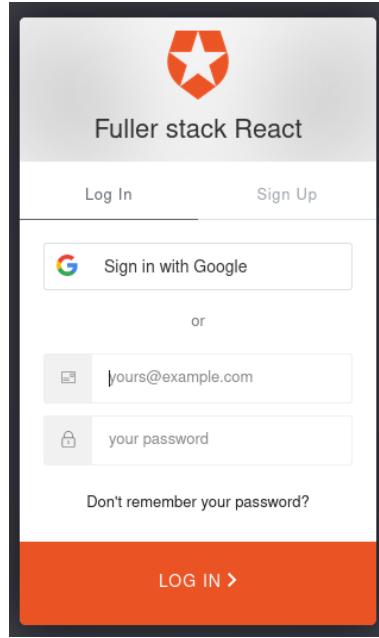


Figure 23: The Auth0 signing screen of the React.JS app

A screenshot of the main screen of the React.js application. The title bar says "Fuller stack". Below it is a toolbar with "ADD A NEW NOTE", a search icon, a filter icon, "CLOSE", "UPDATE", and a more options icon. The main area shows a list of notes: "Note 2" and "Note 1". "Note 2" is selected, showing its details: "26.04". To the right is a modal window for "Note 2", containing "Title: Note 2" and "Content: Content of two".

Figure 24: The main screen of the React.JS app

```

1 fun alert(message: String?) {
2     return js("alert(message)")
3 }
  
```

One thing to note here is that the `js` function returns a dynamic type. Kotlin is a statically typed language but JavaScript is not and does not have any concept of types. This difference requires Kotlin to support cases where the type is unknown or just unexpected and this is what the dynamic type is used for. This dynamic type basically disables Kotlin's type checker, which in hand removes compile-time safety. For example: JavaScript offers a function for calculating the minimum of the numbers. In Kotlin it could look like:

```

1 fun min(a: Int, b: Int): dynamic {
  
```

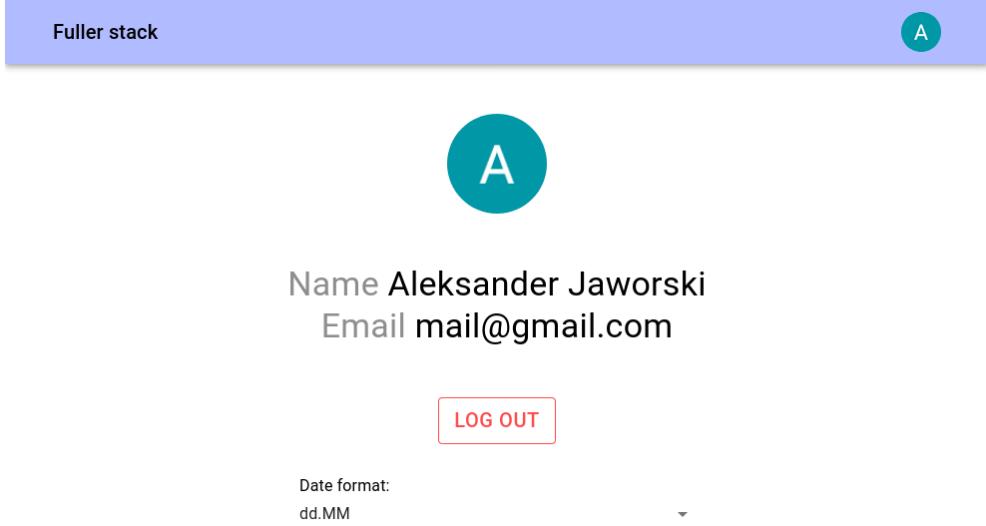


Figure 25: The settings screen of the React.JS app

```
2     return js("Math.max(a, b)")
3 }
```

In order to use the result of the `min` function this result needs to be casted from a dynamic type to a number type. This cast can only happen at run-time, which prevents the compiler from failing if there is a problem with types.

Another problem with inlining JavaScript is the fact that the execution is also not compile-time safe. In the alert example above, changing the `message` parameter to `text` is still a valid Kotlin code which compiles even if the string inside the `js` function remains the same. At run-time, however, the application would not work as expected and executing the `alert` function would result in an error: “Uncaught ReferenceError: message is not defined”. This is because Kotlin executes the JavaScript just as it is defined in the `js` function but the `message` variable does not exist anymore because it was replaced with `text` in Kotlin.

The safer option is to define a contract using the `external` keyword:

```
1 external fun alert(message: String?)
```

This contract assumes that the `alert` function is declared externally in the JavaScript world. This more or less suffers from the same dynamic problem as inlining JavaScript, but it leverages the Kotlin type system more. Changing the parameter names does not break the application at run-time.

### 2.6.1.2 Using JavaScript libraries

As it was mentioned before, Kotlin leverages the JavaScript ecosystem and a lot of Kotlin wrappers are provided for popular JavaScript libraries. These wrappers allow Kotlin to call those libraries

out of the box. Unfortunately, the less popular libraries do not have such wrappers. In such case there is a possibility to write these wrappers by hand using the external keyword mentioned in the previous section. Doing this by hand can be tedious and error prone, fortunately there is an experimental solution for this problem called Dukat [31]. This tool uses existing TypeScript [32] declaration files in order to generate matching Kotlin external definitions.

Given a TypeScript declaration file like follows:

```
1 export function doSomething(): number;
```

Generating a Kotlin file with Dukat gives a result similar to this:

```
1 import kotlin.js.*  
2 import org.khronos.webgl.*  
3 import org.w3c.dom.*  
4 import org.w3c.dom.events.*  
5 import org.w3c.dom.parsing.*  
6 import org.w3c.dom.svg.*  
7 import org.w3c.dom.url.*  
8 import org.w3c.fetch.*  
9 import org.w3c.files.*  
10 import org.w3c.notifications.*  
11 import org.w3c.performance.*  
12 import org.w3c.workers.*  
13 import org.w3c.xhr.*  
14  
15 external fun doSomething(): Number
```

Dukat adds a lot of additional imports which are not always needed. Fortunately, the generated Kotlin files can be modified to contain only what is needed.

In this project this tool has been used to generate Kotlin files for two libraries, one is for authentication and the other is for a database. These two libraries reside in two separate modules shown in Figure 5.

#### 2.6.1.3 Kotlin React.JS

This section will show some JavaScript React.JS examples and how they can be achieved in Kotlin.

The first example will be a page with an input for typing and a button, which is presented in Figure 26

```
1 class TextInputExample extends React.Component {  
2     render() {  
3         return (
```



Figure 26: A React.JS page with a text input and a button

```

4   <div>
5     <input
6       type="text"
7       onChange={(event) => {
8         const newVal = event.target.value;
9         console.log(newVal);
10      }}
11    />
12    <button onClick={() => console.log("Button clicked")}>
13      A button
14    </button>
15  </div>
16 );
17 }
18 }
19
20 ReactDOM.render(<TextInputExample />, document.getElementById("root"));

```

React.JS uses JSX [33] which is an extension for JavaScript and allows for writing code in an HTML-like syntax. In the example above TextInputExample is a class react component and it is used just as other HTML elements in React.JS (line 20).

```

1 class TextInputExample : RComponent<RProps, RState>() {
2   override fun RBuilder.render() {
3     input(InputType.text) {
4       attrs.onChangeFunction = { event ->
5         val newVal = (event.target as HTMLInputElement).value
6         console.log(newVal)
7       }
8     }
9     button {
10       + "A button"
11       attrs {
12         onClickFunction = { console.log("Button clicked") }
13       }
14     }
15   }
16 }
17
18 
```

```

13         }
14     }
15   }
16 }
17
18 fun main() {
19   window.onload = {
20     render(document.getElementById("root")) {
21       child(TextInputExample::class)
22     }
23   }
24 }
```

The Kotlin wrapper for React.JS does not use JSX but it uses a custom DSL for it (explained in more depth in section 1.3.12). Instead of using HTML-like tags, Kotlin uses predefined functions which correspond to a given HTML tag. Since using React components differs from the JSX usage, an additional **child** function (line 21) needs to be used in order to place the React component.

The second example does not use a class component but a functional component which has better performance and involves less boilerplate code. Additionally, the component uses props and state. Props could be thought of arguments which are passed in from the parent component. When a prop value changes, the component is re-rendered in order to reflect the new value. While props are passed in from the parent, the state is self contained in the component and optionally passed to its children components. Both props and state changes cause the component to re-render. The example in Figure 27 has a text label which value is based on a passed in prop and a checkbox which value is based on the components state.

## ReactJS

Figure 27: A React.JS page with a checkbox which uses state and props

```

1 function CheckBoxExample( props ) {
2   const [isChecked, setIsChecked] = useState( false );
3
4   return (
5     <label>
6       {props.name}
7       <input
8         type="checkbox"
9         checked={isChecked}
```

```

10         onChange={() => {
11             setIsChecked (!isChecked );
12         } }
13     />
14     </label>
15   );
16 }
17
18 ReactDOM. render (
19   <CheckBoxExample name="React . JS" />,
20   document . getElementById ("root")
21 );

```

```

1 interface ExampleProps : RProps {
2   var name: String
3 }
4
5 val checkboxExample = functionalComponent<ExampleProps> { props ->
6   val (isChecked , setIsChecked) = useState(false)
7
8   div {
9     label {
10       + props . name
11       input (InputType . checkBox) {
12         attrs . checked = isChecked
13         attrs . onChangeFunction = {
14           setIsChecked (!isChecked )
15         }
16       }
17     }
18   }
19 }
20 }
21
22 fun main() {
23   window . onload = {
24     render (document . getElementById ("root")) {
25       child (checkboxExample) {

```

```

26         attrs.name = "React.JS"
27     }
28 }
29 }
30 }

```

Because Kotlin is a static language, the props contract is defined in the form of an interface (line 1). Both JSX and Kotlin allow for easy extensions in the form of extracting existing logic into standalone components. But I personally think that modifying and creating functions in Kotlin is easier and safer than in JSX thanks to types and a less verbose syntax.

### 2.6.2 Modules

The main web app module consist of the React module along with its dependency modules shown in Figure 28.

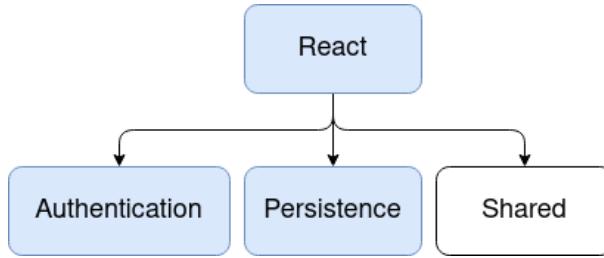


Figure 28: Module structure used in the web app

- Authentication - provides user authentication functionality using the Auth0 library;
- Persistence - provides database access thanks to the Dexie.js library [34];
- Shared - provides shared functionality.

Both the libraries for authentication and persistence were generated using Dukat which was described in section 2.6.1.2.

### 2.6.3 Architecture

Because the web app uses Redux [30] for the state management, most of the architecture revolves around it.

#### 2.6.3.1 Redux

Redux is not an integral part of the whole project, so it will be briefly explained with a simple checkbox example shown in Figure 29.

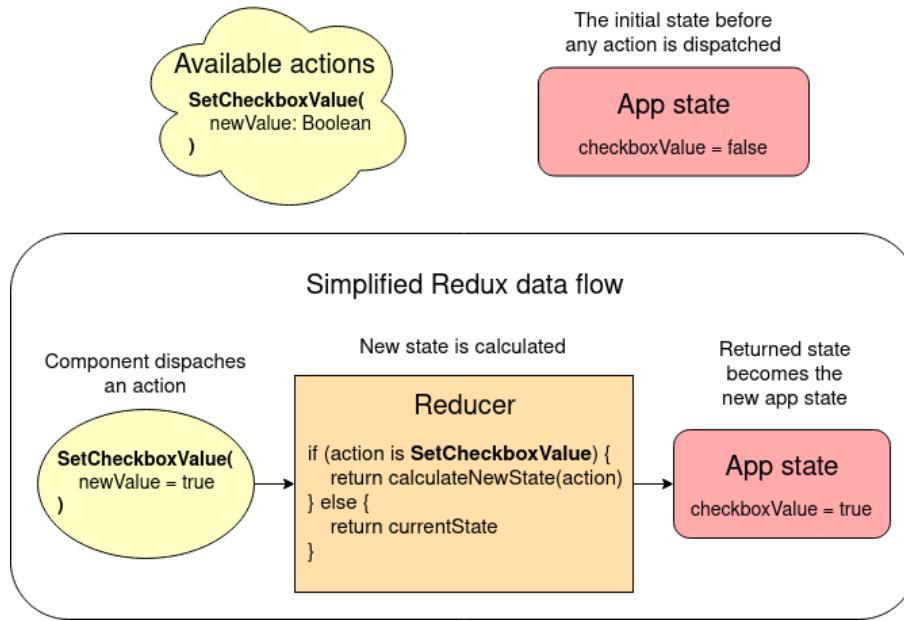


Figure 29: Simplified Redux data flow for an application with a single checkbox

The state of the app resides in a single state container branch (i.e a branch could relate to a screen or a feature) in this case App state. This state is only changed by the reducer, which is invoked when an action is dispatched. Because React.JS components do not ask for state but only react to state changes in combination with Redux this creates a unidirectional data flow shown in Figure 30.

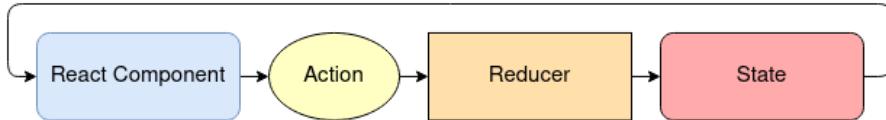


Figure 30: Unidirectional React.JS and Redux data flow in which the component only listens to state changes

### 2.6.3.2 View layer

The view layer is responsible for showing the user interface and reacting to user input. In React.JS the view is composed from components which were briefly touched in the Kotlin React.JS 2.6.1.3 section. The components could be thought of as small building bricks which are joined together in order to create something bigger like the notes list shown in Figure 31 with an explanation in Figure 32.

The whole notes list consists of an **action row** component and a number of **note items**. The Action row is composed from a **button** component and two **icon** components.

ADD A NEW NOTE			
Book notes	01.03		
Homework	27.02		

Figure 31: The notes list in the web app

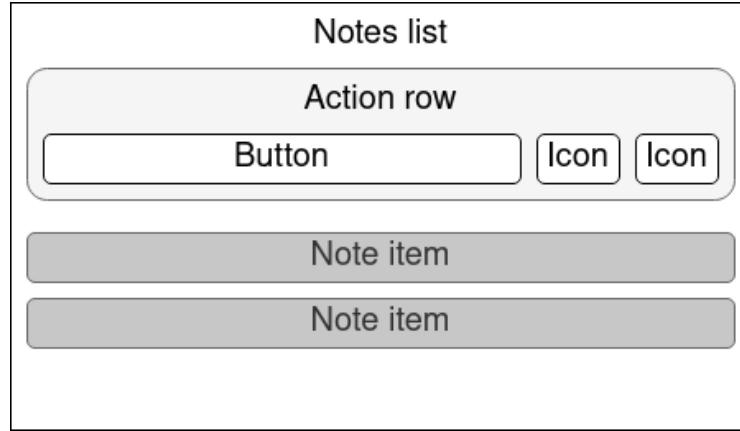


Figure 32: A diagram of components which are used for building the notes list

Some components hold their own state and some are derived from Redux. Usually things that relate to outside sources like the users notes are held in the Redux state.

#### 2.6.3.3 Presentation layer

The presentation layer mainly revolves around Redux, since it is an integral part of almost every main application part like the notes list or the note editor. The Redux usage was inspired by Redux-toolkit [35] and its convention of using “slices”.

Slices can be thought of containers for related Redux classes and functions. As it was explained in the Redux section 2.6.3.1, the main building blocks of Redux are actions reducers and a state. Slices are the classes which hold these main building blocks. Referring back to the checkbox example in Figure 29, below one finds the Kotlin Redux code which could be used for this:

```

1 object CheckboxSlice {
2     data class State(
3         val checkboxValue: Boolean = false
4     )
5
6     data class SetCheckboxValue(val newValue: Boolean) : RAction
7
  
```

```

8     fun reducer(state: State = State(), action: RAction): State {
9         return when (action) {
10            is SetCheckboxValue -> {
11                state.copy(checkboxValue = action.newValue)
12            }
13            else -> state
14        }
15    }
16 }
```

This slice class contains the classes for the state, actions and a reducer function which calculates the new state.

For asynchronous actions like reaching out to the database, Redux thunk [36] is used. The basic idea is such “Thunks” provide a callback function which can be invoked after an asynchronous task ends. This callback takes in an action which is sent to the reducer in order to modify the state.

#### 2.6.3.4 Summary

With all of this in mind the web app architecture for the View and Presentation layer can be summed with Figure 33.

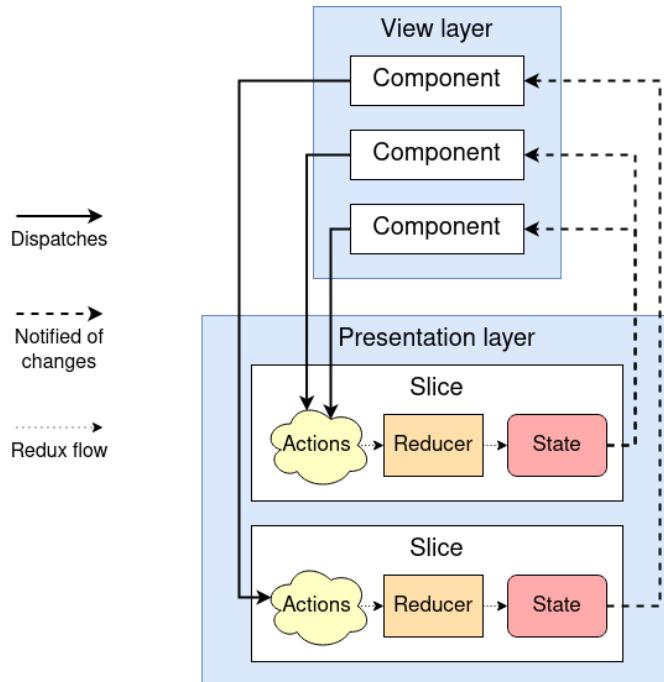


Figure 33: The architecture of the web app view and presentation layers

The React.JS components dispatch actions usually as a result of a user input. These actions

are sent to a reducer function which changes the state. Because React.JS components are reactive, any relevant state changes cause the components to re-render based on the new state.

#### 2.6.4 Technologies used

Most of these libraries make use of Kotlin wrappers, but I have omitted the ones which come from the official JetBrains [4] repository.

Libraries related to UI:

- React.JS [2] - allows for composing the user interface using small self contained components;
- Styled components [37] - provides easier styling for React.JS components;
- Material UI [38] - provides predefined React.JS components which adhere to material designs;
- Muirwik [39] - the Kotlin wrapper for Material UI.

Libraries related to logic:

- React router [40] - provides declarative navigation to React.JS apps;
- Redux [30] - a state container for JavaScript apps;
- Redux thunk [36] - an extension for Redux which allows for asynchronous actions;
- Auth0 [36] - allows for an easy integration of authentication in React.JS;
- Kotlin Coroutines [27] - provides asynchronous programming features to the Kotlin language.  
It is used in the React presentation layer;
- Klock [28] - library used for operating with notes timestamps and formatting them to dates on the notes list;
- Kotlin Serialization [29] - allows for an easy JSON serialization for API requests and responses;
- Kodein [19] - allows for injecting dependencies and retrieving them from the react module;
- Dexie [34] - the JavaScript library for the database. The Kotlin files were generated using Dukat [31].

### 2.7 Android app

The android app has the following screens:

- Sign in screen, Figure 34 - The only screen which unauthenticated users can see, clicking the button opens the Auth0 sign in website shown in Figure 35;
- Home screen, Figure 36 - The starting screen of the app for authenticated users which shows a list of notes. This screen also opens the note editor screen shown in Figure 37 by clicking the floating action button on the bottom right of the home screen;
- Profile screen, Figure 38 - A screen where the user can sign out of their account;
- Settings screen, Figure 39 - Allows for changing the notes date formatting preferences.

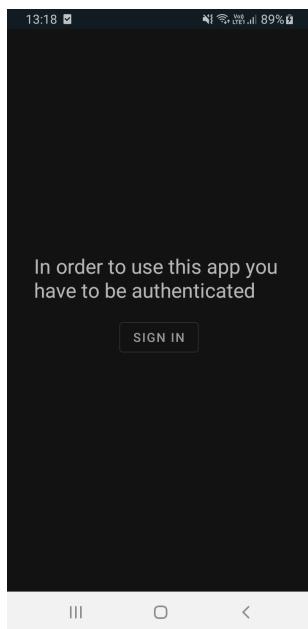


Figure 34: The Android screen used for authentication

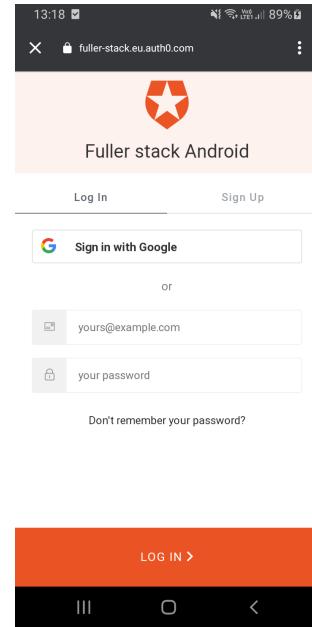


Figure 35: The Auth0 website used for authentication in the Android app

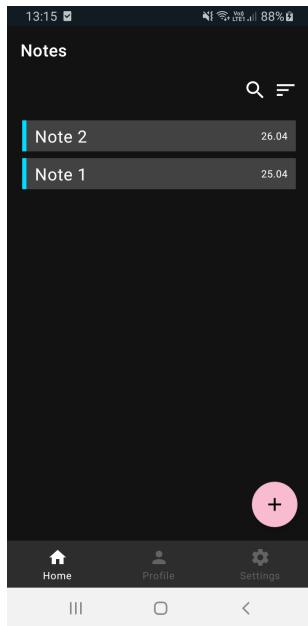


Figure 36: The main screen of the Android app

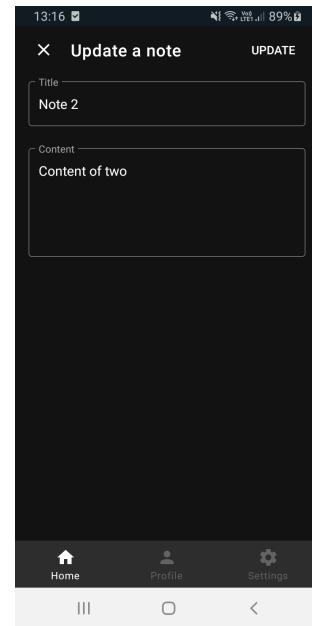


Figure 37: The Android screen for editing notes

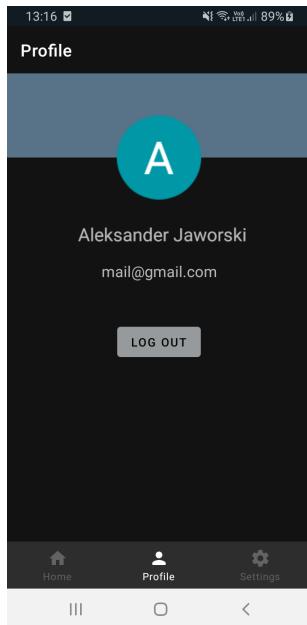


Figure 38: The user profile screen for the Android App

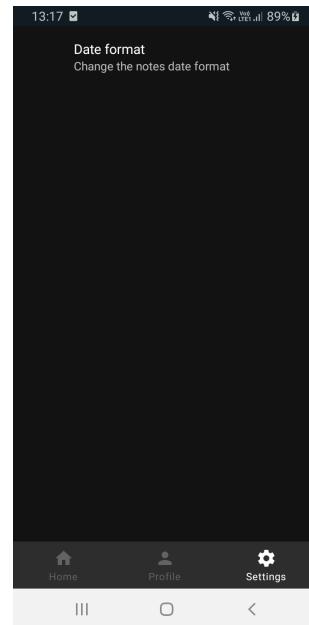


Figure 39: The settings screen for the Android App

### 2.7.1 Kotlin/JVM

Kotlin started out as a language for the Java Virtual Machine. This means that Kotlin targeting the JVM is the most used and refined part of the Kotlin ecosystem. As it was stated in the Kotlin programming language section 1.3, Kotlin offers 100% interoperability with Java. Consequently, Kotlin can be introduced gradually to a Java codebase.

Kotlin/JVM can be used for building applications in every environment that Java can be used in, which includes:

- Desktop applications;
- Android apps;
- Servers.

In this MSc project Kotlin/JVM is used on the Ktor server and in the Android app.

### 2.7.2 Modules

The main Android module uses three sub-modules shown in Figure 40.

- Shared - provides shared functionality;
- Authentication - provides user authentication functionality using the Auth0 library;
- Framework - contains shared Android specific classes and interfaces for the main and Authentication sub-module.

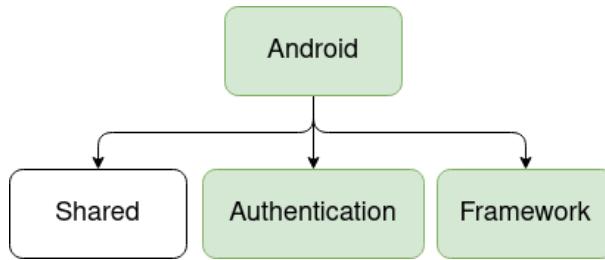


Figure 40: Module structure used in the Android app

### 2.7.3 Architecture

#### 2.7.3.1 Model-View-ViewModel

The Android architecture in this MSc project makes use of the official Android Architecture Components [41] which recommend using the MVVM (Model-View-ViewModel) architecture. The Model in MVVM should represent business logic which in this project resides in the domain layer in the form of use cases discussed before. A simplified diagram of the MVVM composition is shown in Figure 41



Figure 41: A simplified diagram of the Android MVVM Architecture for this project

#### 2.7.3.2 View layer

Without going into details, this MSc project uses Android Fragments as the View. Fragments are reusable portions of the UI but in this project they always represent a single screen. Figure 42 shows a mock-up of an Android app with three screens.

All three screens are represented by a fragment and the bottom buttons change which fragment is shown. The Android app for this MSc works in the same way.

Briefly speaking fragments are responsible for showing the UI and reacting to a user input. Sometimes they do more than that, but I will omit such discussion to keep the explanation concise. In this MSc project they work in the same way that React.JS components do. They react on data changes and send user events to the presentation layer.

#### 2.7.3.3 Presentation layer

The ViewModel is the heart of the presentation layer and is a mediator between the View and the Domain layer.

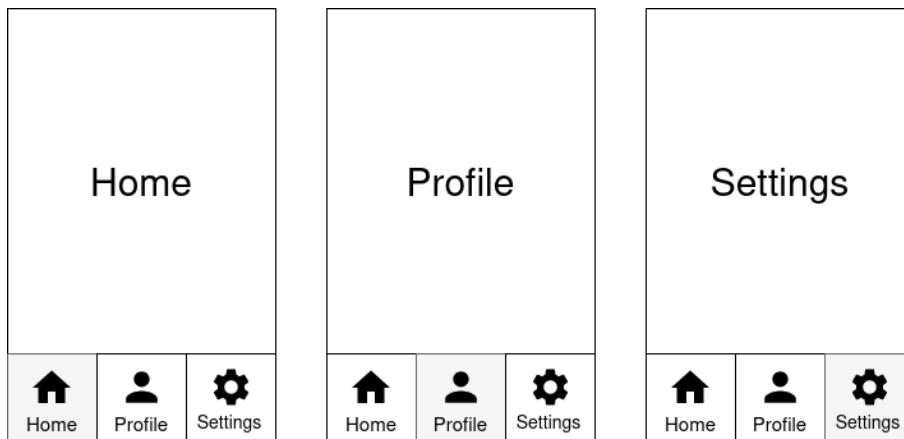


Figure 42: An Android app mock-up with three screens and bottom buttons

The data for the View is exposed through so called LiveData [42] which basically is an implementation of the Observer pattern [43]. It allows the View to listen to LiveData data changes but it is only emitting changes if the fragment is in correct state (i.e. the fragment is shown and not closing).

User interaction usually revolves around calling ViewModel functions based on the View callbacks. Most of the time the ViewModel functions reach out to the Domain layer and then update the corresponding LiveData which updates the UI.

On more complicated screens like the notes list or the note editor all of the UI related data are encapsulated in a single class. Thanks to this the View is only concerned with one LiveData which holds all the necessary View data. This pattern is often called “Single View States” [44]. These Single View states are either represented by data classes or sealed classes (section 1.3.7 and 1.3.9):

```

1 data class NoteEditorState(
2     val note: ParcelableNote? = null,
3     val titleError: String? = null
4 )

```

In this case **note** holds the current user input in the editor and **titleError** holds the error message in case the user gives an incorrect title. Both of these states can happen simultaneously and that is why a data class is used for this state.

```

1 sealed class NotesListState {
2     object Loading : NotesListState()
3     data class ShowingList(val notes: List<Note>) : NotesListState()
4 }

```

In this case the state can be in one of two states, either loading or showing a list. Both of the states cannot be shown at the same time, and that is why a sealed class was used for this state.

#### 2.7.3.4 Summary

The Android app architecture for the View and Presentation layers is shown in Figure 43.

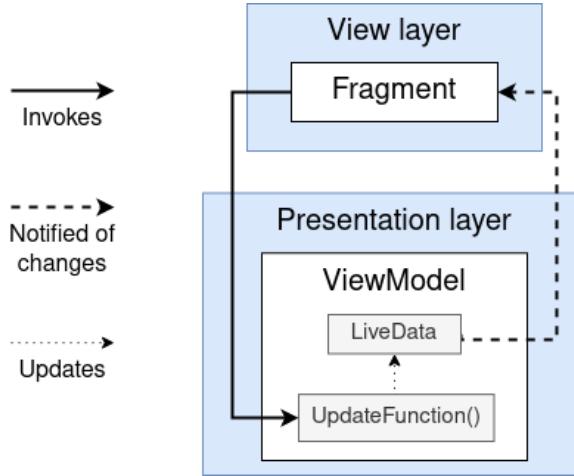


Figure 43: The architecture of the Android view and presentation layers

#### 2.7.4 Testing

The Android module only contains unit tests for the presentation layer and most of it are tests for ViewModels. The framework used for writing these tests is called JUnit [45] which is the standard for writing JVM tests. The particular version used is JUnit 5.

Because the shared module already contains fakes for the database and the API, the Android module makes use of them. This means that the UseCases in the ViewModels are not mocked, their real implementation is used but in place of the database and the API a fake is used. Using real implementations in unit tests instead of test doubles could be perceived as integration tests.

#### 2.7.5 Technologies used

A lot of libraries coming from the androidx package were omitted in order not to make this list too exhaustive. The list below contains the most notable ones:

- Android ViewModel [46] - the official implementation of the ViewModel in Android;
- Android LiveData [42] - an observer wrapper which is aware of the Android life cycle;
- Room [24] - persistence library;
- Auth0 [47] - provides an easy authentication integration in Android;
- Simple Stack [48] - a library for navigation between screens;
- Retrofit [49] - a type safe HTTP client;
- OkHttp [50] - a networking library;

- Coil [51] - provides fast image loading;
- Kotlin Coroutines [27] - allows for using asynchronous programming Kotlin features. It is mostly used in the Android presentation layer.;
- Klock [28] - library used for formatting the notes timestamps;
- Kotlin Serialization [29] - allows for retrofit to automatically serialize JSON responses and requests;
- Kodein [19] - provides dependency injection from the Android module.

Testing libraries:

- JUnit5 [45] - a testing framework for the JVM;
- Mockk [52] - allows for mocking in Kotlin using a DSL;
- Coroutines test [53] - contains utilities for easier coroutines testing.

## 2.8 Code distribution

The code distribution will be counted by Kotlin source code lines meaning that empty lines and comments are not counted. All of the modules and their production source code lines are shown in Figure 44. Issues like generated libraries (Dukat 2.6.1.2) and tests were not taken into account.

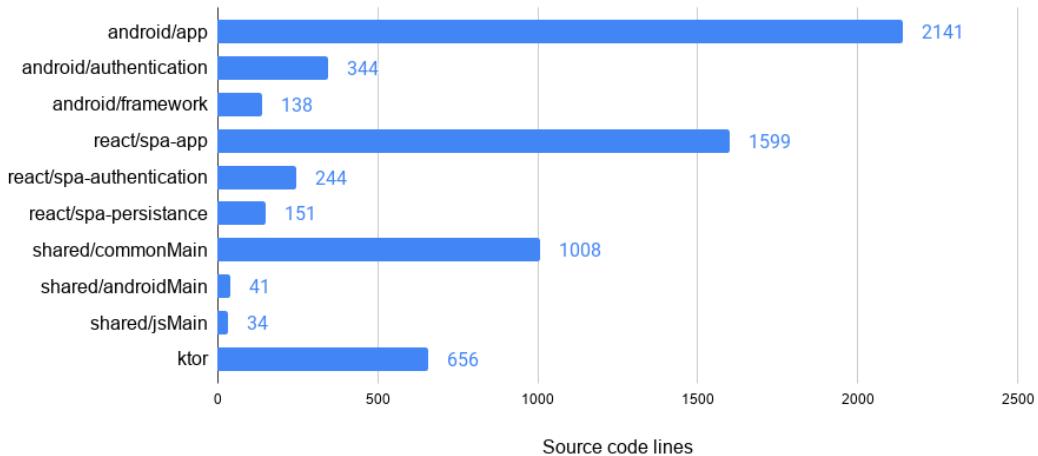


Figure 44: The source code lines distribution for all the modules in the project

Most of the code resides in the client platforms, i.e. Android and React, respectively. This is the only Figure where the Ktor (server) module is included.

A bar chart of the distribution based on the platform is shown in Figure 45. The percentage distribution is shown as a pie chart in Figure 46.

The platform modules contain a lot of code for the view and presentation layers. However, additionally these modules also contain a lot of implementations for the data layer. Some of the

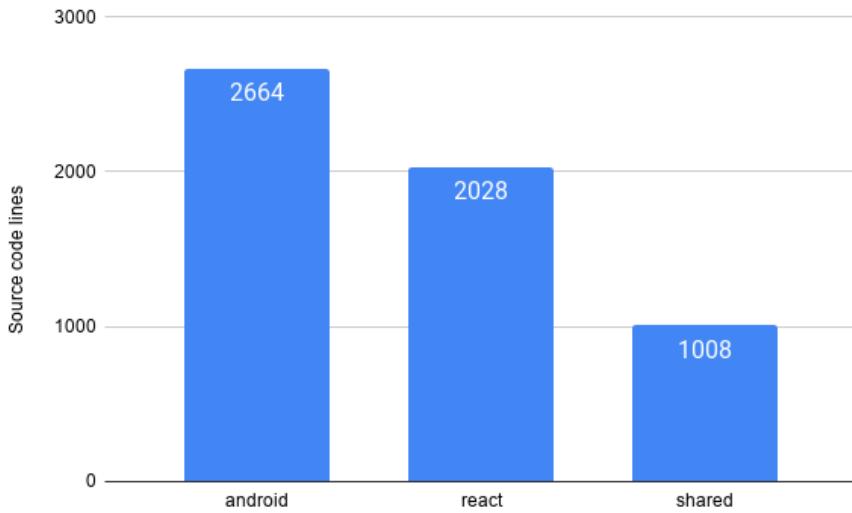


Figure 45: The production source code lines for the shared module and client platforms

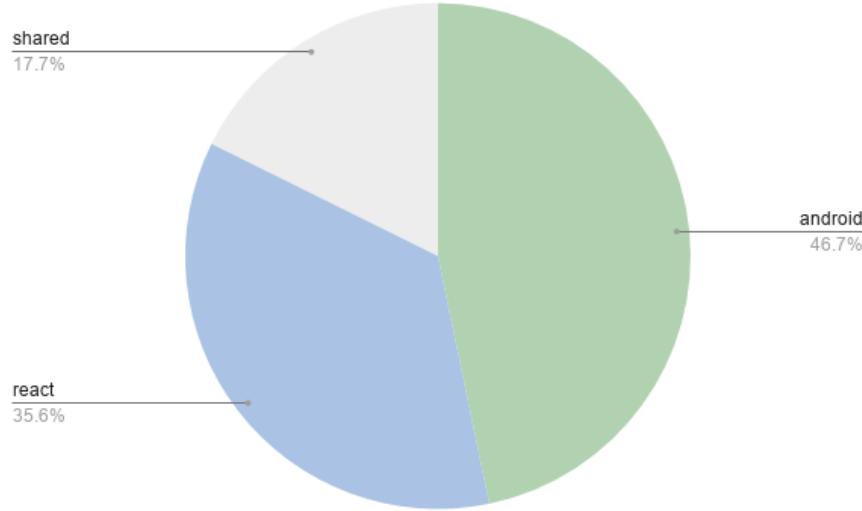


Figure 46: The production source code percentages for the shared module and client platforms

implementations could have been avoided by using Kotlin Multiplatform libraries which work on both platforms. Most notably in this project the networking, settings persistence and maybe the database could have been implemented fully in the shared module.

For projects with more business logic than the one for this MSc the shared module could potentially have more code compared to the client platforms. The Android and the React platforms

are significantly different from each other which makes the code re-use harder. If the project was comprised of two mobile platforms (Android and iOS) the code reuse would be higher because Kotlin Multiplatform for mobile community is much bigger. There exist more libraries for mobile and additionally the two platforms which are more similar to each other than Android and React.

The source code lines distribution with testing code included is shown as a bar chart in Figure 47 and as a pie chart in Figure 48.

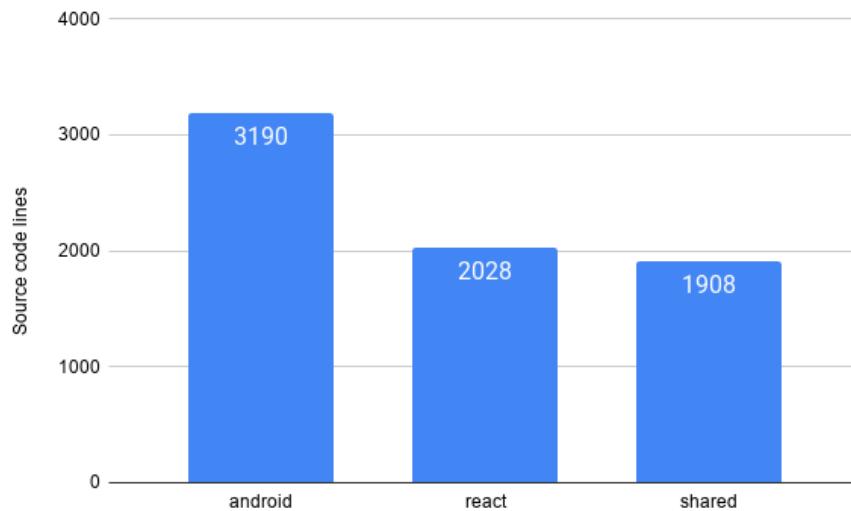


Figure 47: The production and test source code lines for the shared module and client platforms

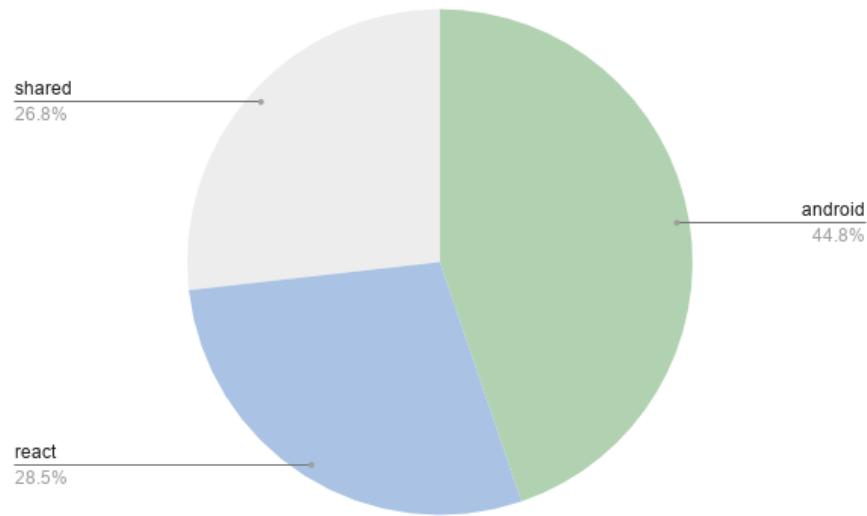


Figure 48: The production source code percentages for the shared module and client platforms

The React module does not contain any tests so its lines of code did not change. However, the shared module almost doubles in lines of code because it contains the most tests. In my opinion tests should also be considered in the overall metric, because if the platforms did not share code, both platforms would probably contain production and tests code which would be approximately the same.

## 3 Hiccups/ problems encountered

### 3.1 Kotlin/JS wrapper dependency

Some Kotlin/JS libraries (Mostly related to React.JS) are dependent on the official Kotlin wrapper libraries (discussed in section 2.6.1). From my experience I can state that some versions of the wrappers are not compatible with each other. Having a project like the one shown in Figure 49 could potentially not work correctly. The project uses the Kotlin wrapper library for React.JS

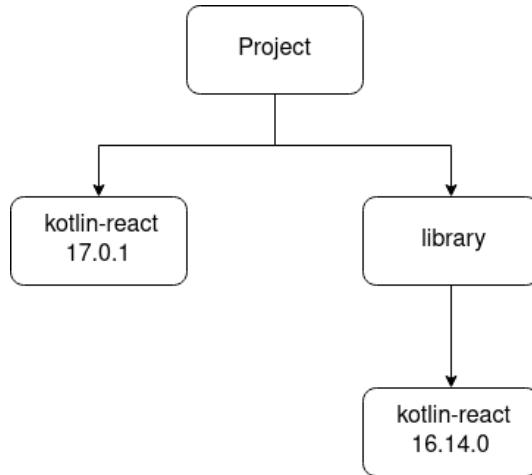


Figure 49: Kotlin/JS project with wrapper library dependency

with the 17.0.1 version. Additionally, the project uses an external library which internally uses the same Kotlin wrapper library but with a different version. This discrepancy in versions could lead to unexpected errors. The solution would be to find a version of the wrapper and the library that are compatible with each other (by trial and error).

### 3.2 Kotlin 1.4 update

Kotlin 1.4 was a feature release which brought major changes in the language. This update changed some things with how Kotlin multiplatform is defined making the project not compile. However, the fix for this was not hard. The most problematic part of the update was Kotlin/JS and its wrappers / libraries. I was forced to hold off on updating to Kotlin 1.4 until the wrappers and libraries I used were also up to date with this version. Because of this I think that in a project which uses Kotlin/JS feature release updates should not be performed straightaway.

### 3.3 Platform dependent implementation problems

The main focus of Kotlin Multiplatform is the mobile platforms (Android and iOS). Because of this some of the popular multiplatform libraries do not support Kotlin/JS. This forces some

implementations to be done on both platforms instead of the shared module. Because of this separation special care has to be taken when creating these implementations in order not to create a difference between the platforms.

In this project the local database was the biggest problem when keeping the platforms aligned. The JavaScript database library Dexie [34] had problems. The auto incrementing of the identifier only worked if the note data structure had an undefined value (similar to null in JavaScript) as the Id. Introducing a possibility for an undefined value in the shared module made no sense because it is only a local problem for the Kotlin/JS platform. The solution was creating an intermediary data structure between the JavaScript database and the shared module.

Additionally, the Android Room database disallows coroutine suspending functions when retrieving values from the database. This made the Kotlin/JS database even harder to implement, I won't go into the details because it is not relevant for this thesis.

### 3.4 Testing the shared module

As it was discussed in the Shared module section 2.5.5, Kotlin multiplatform does not have mocking frameworks which means that all test doubles need to be created by the developer. This is what was done for the API and the database, however sharing these test doubles in tests from different modules is troublesome. For example, in order to use a test double for an Android module it needs to be defined in the main common module. Defining it only in a test module does not make it visible outside of the shared module. This results in some name space pollution because production code can access these classes even though they should just be used by tests.

Another problem stems from testing coroutines. There exists an additional library for testing coroutines. Unfortunately, it is only available for the JVM which makes it unavailable in the shared module package. This means that the shared module does not have officially supported utilities for testing coroutines.

### 3.5 Broken navigation to declaration

The IDE I used for creating this MSc project is called IntelliJ IDEA [54]. It has a functionality called "Go to declaration". What it does is basically to open the file where the function/class etc. was declared. This makes it easier to reason about what is the implementation or what properties/functions (later called member) a class contains. Unfortunately, throughout working on the project I had a bug with this functionality. For example, navigating from the Android module to a Shared module declaration did not work as expected. I wanted to navigate to the exact Kotlin file where the member was declared, however instead of going to the Kotlin file I went to the generated Java class.

This is an ongoing issue with Kotlin Multiplatform. The oldest issue ticket I was able to find is three years old KTIJ-11683 [55]. Searching through the web I was not able to find a working

solution because there is little information about this issue. This makes me wonder if this is something that happens seldom and is caused by my project configuration. This is not a make or break problem, yet it is a significant inconvenience when crossing the Android and Shared module boundaries.

### 3.6 Kotlin/JS RAM usage

Every time I was working on the React side of this MSc project I was forced to restart my browser from time to time. The reason for this was that my machines memory was disappearing bit by bit every time I made a change in the React module. In web development a popular practice is Live reloading [56] which boils down to refreshing the page whenever a change is made. This can also be achieved with Kolin/JS. However, every time a live reload happened on the project, the memory usage went up a little bit. I suspect that the memory issue is related to live reloading however I do not have 100% certainty. This is just my personal intuition after developing the React module.

## 4 Summary

### 4.1 Synopsis

#### 4.1.1 Kotlin/JVM

The JVM platform was the initial target for Kotlin making it the most refined part of the Kotlin programming language. In this thesis it was used for creating the Android app along with the Server. Because the JVM is already a widely established platform it benefits Kotlin because it can be used everywhere Java and other JVM languages can be used, for example desktop applications.

Kotlin/JVM is also the most popular way of using Kotlin. The language has mostly gained traction in the Android development circle, however it is expanding to the server side. Spring [57] is a popular Java framework for creating robust enterprise applications and it also has an official support for Kotlin [58].

According to AppBrain [59] Kotlin is used in 75% of the top 500 USA apps (compare Figure 50). Keep in mind that the scale of the Kotlin use is unknown, i.e. it could be the whole application or one file.

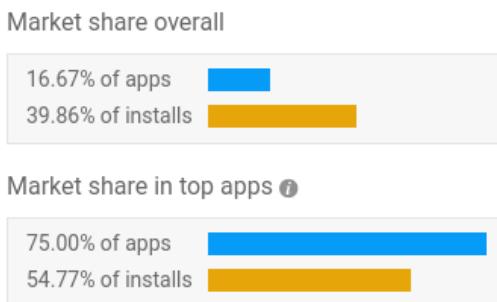


Figure 50: Kotlin Google Play Store statistics,

source: <https://www.appbrain.com/stats/libraries/details/kotlin/kotlin>

On the server side Kotlin is used in such companies as ING [60], Amazon [61], Adobe [62].

#### 4.1.2 Kotlin/JS

As shown in this thesis Kotlin can also be used for creating websites or single page applications. It has the ability of being transpiled into JavaScript. On top of this, Kotlin can also call JavaScript code and use JavaScript libraries. Leveraging the JavaScript ecosystem is an important part of the front-end development because there is no point in reinventing the wheel.

All of this comes with some issues like the one described in the previous section, however I still think Kotlin has a potential for building JavaScript applications. Kotlin/JS is not the main focus of Kotlin and because of this it has a lot more rough edges than for example Kotlin/JVM. As time goes by I do believe that it will be more refined.

The framework I used for creating the web application was called React.JS which is popular in the JavaScript world. In the near future there will be an official way of building UI for the web which is discussed in 4.2.1.

In my opinion for small projects or projects using Kotlin Multiplatform building web applications using Kotlin/JS might be the right choice. However, for more complex applications I would personally go with the standard which is JavaScript or TypeScript. These communities are much larger than Kotlin/JS which means that there are less issues and they are easier to be resolved.

#### 4.1.3 Kotlin/Native and more

Kotlin/Native was not discussed in this thesis, however it is the main driving force behind Kotlin Multiplatform. Kotlin/Native makes it possible to use Kotlin for writing applications for the following platforms (This is not a complete list):

- macOS;
- iOS;
- Linux;
- Windows.

I do not have any personal experience on Kotlin/Native, yet I will elaborate on how it is used for Mobile development in section 4.2.2.

Besides all of the above platforms Kotlin can also be used as a scripting language. I have not had a chance to use it but I suspect that it could be helpful for small scripts which need to be run from time to time. The main issue would be that Kotlin is not an established language which means that the script is not as portable as for example bash.

#### 4.1.4 Kotlin overview

It is a modern language which is not forced to provide backwards compatibility for choices made over 10 years ago (like Java). The Kotlin language has learned from the mistakes of other popular albeit old languages and connected their features into a cohesive package. Before releasing a stable version of Kotlin, Jetbrains was developing it for 6 years which allowed them to refine the language without the worry of backwards compatibility.

At its current state Kotlin can be used in a plethora of environments and the list just keeps growing. Kotlin Multiplatform offers one of a kind code sharing between platforms where the UI stays native. It might be a solution where an application needs to be released for multiple platforms.

According to the StackOverflow 2020 survey [1], Kotlin is the 4<sup>th</sup> most *Loved* language. The survey is shown in Figure 51.

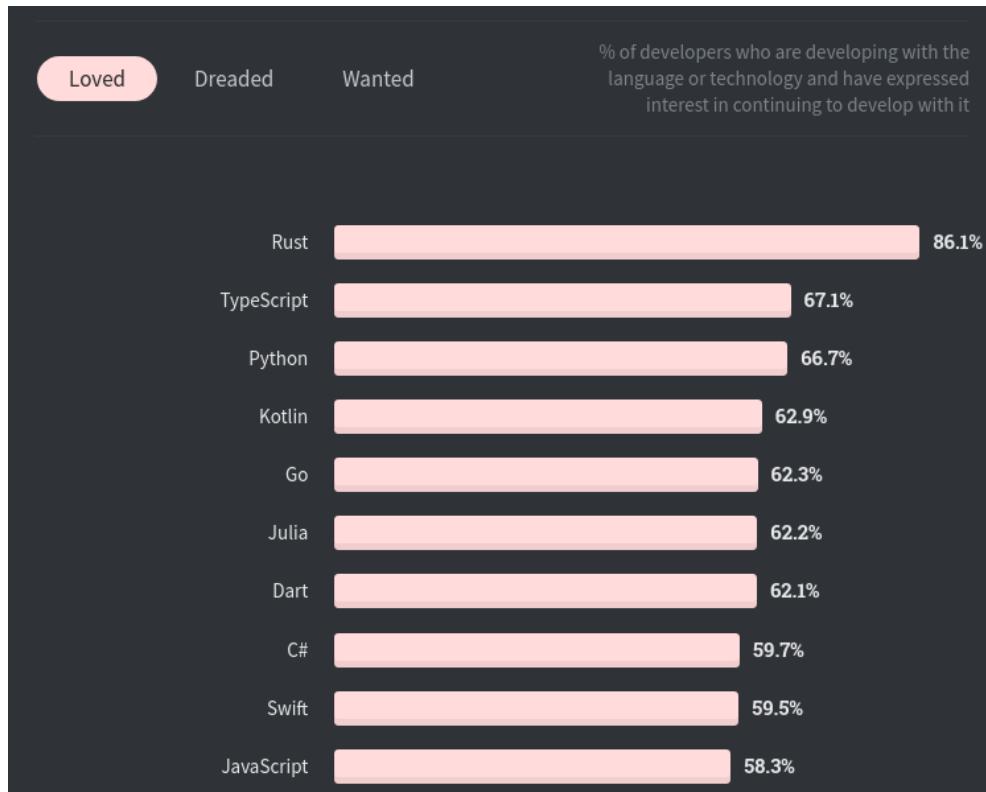


Figure 51: Top 10 most loved languages in the StackOverflow 2020 survey

credit: <https://insights.stackoverflow.com/survey/2020>

## 4.2 Future outlook

### 4.2.1 Compose

Jetpack Compose [63] is a UI framework which works in a similar way that React.JS where the UI is driven by state. The Compose framework exposes a DSL 1.3.12 for building the UI which looks in a similar way to how it was done in this project 2.6.1.3.

At first the framework was focused on the Android platform with the promise of replacing the cumbersome way UI is currently built on Android. In the same vein that Kotlin offers interoperability with Java, Compose also offers interoperability with the old Android UI system. Thanks to this Compose can be introduced into a project step by step, just like Kotlin in a Java codebase without a need of a ground-up rewrite.

Jetpack Compose is developed by Google with the focus being Android, however JetBrains has also developed different versions of Compose. The most mature is Compose for Desktop [64]. Although still in alpha it proves that Compose can potentially be used for sharing the UI between platforms. Recently it has been announced that Compose will also be coming to the web [65] with the help of Kotlin/JS. I think that the addition of Compose for the web will make Kotlin/JS more

popular in the future.

When I started working on this MSc thesis project Kotlin Multiplatform was still experimental and Jetpack Compose in alpha. Additionally, Compose was not supported in my IDE yet this prompted me to use it in the Android app. If I were to start working on this project in a year from now I would definitely have chosen Compose for the Android UI. Depending on the state of Compose for web it is possible I would have used it for the web UI.

#### 4.2.2 Kotlin Multiplatform Mobile

In my opinion this is bread and butter of Kotlin Multiplatform, the mobile platforms have always been split. Currently there are two main ones: Android and iOS. The mobile platforms usually require two separate development teams where each one is focused on their own platform. Both the teams work on similar things but the only difference is the targeting platform. To reduce the development overhead mobile cross-platform solutions were created like Xamarin [66], React Native [67] or Flutter [68] which have a varying level of success.

All of the solutions mentioned above have problems which are UI related. Both of the Mobile platforms differ vastly in the way they create the UI, solutions like Flutter have to come up with a way in which both the platforms UI behave and look in a similar way. In some situations these cross-platform apps might look out of place because they use custom UI elements instead of using what the native platform has to offer. Keep in mind that this is an oversimplification because I am only focusing on the UI without other native parts of the applications.

Kotlin Multiplatform works in a different way because instead of sharing everything between the platforms it only focuses on the logic which for sure has to be the same between both of them. Thanks to this the UI stays native along with other native parts which means that both platform teams can leverage everything they have learned about their corresponding platform. The one big thing that is added is the Shared module which from the Android platform perspective will be easier to integrate than on the iOS platform.

In my opinion one big advantage of Kotlin Multiplatform is that if the solution does not meet the requirements, the Android platform will still have more or less a working App without Kotlin Multiplatform because Android also uses Kotlin. The iOS platform will be in a worse place because there will be a need to recreate the Shared module in their programming language. However, everything native or related to the UI already exists and can be used freely. In the case a different cross-platform solution was used the whole app would have to be rewritten from scratch on both of the platforms.

## References

- [1] *StackOverflow 2020 survey.* <https://insights.stackoverflow.com/survey/2020>. [Accessed 2021.05.22].
- [2] *React.JS.* <https://reactjs.org/>. [Accessed 2021.02.22].
- [3] *Kotlin.* <https://kotlinlang.org/>. [Accessed 2020.09.04].
- [4] *JetBrains.* <https://www.jetbrains.com/>. [Accessed 2020.09.04].
- [5] *Java.* <https://www.java.com/en/download/>. [Accessed 2020.09.04].
- [6] *Kotlin Java interoperability.* <https://kotlinlang.org/docs/reference/faq.html#is-kotlin-compatible-with-the-java-programming-language/>. [Accessed 2020.09.04].
- [7] *Kotlin Android.* <https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017/>. [Accessed 2020.09.04].
- [8] *Arrow Kotlin.* <https://arrow-kt.io/>. [Accessed 2020.09.04].
- [9] *Kotlin Multiplatform.* <https://kotlinlang.org/docs/reference/mpp-intro.html>. [Accessed 2020.09.04].
- [10] *Kotlin Native.* <https://kotlinlang.org/docs/native-overview.html>. [Accessed 2021.05.29].
- [11] *iOS.* <https://en.wikipedia.org/wiki/iOS/>. [Accessed 2021.05.29].
- [12] *Builder pattern.* [https://en.wikipedia.org/wiki/Builder\\_pattern/](https://en.wikipedia.org/wiki/Builder_pattern/). [Accessed 2021.03.25].
- [13] *Telescoping constructor pattern.* <http://www.javabyexamples.com/telescoping-constructor-in-java/>. [Accessed 2021.03.25].
- [14] *Singleton pattern.* [https://en.wikipedia.org/wiki/Singleton\\_pattern](https://en.wikipedia.org/wiki/Singleton_pattern). [Accessed 2021.03.25].
- [15] *GitHub.* <https://github.com/>. [Accessed 2020.12.5].
- [16] *Gradle.* <https://gradle.org/>. [Accessed 2020.09.13].
- [17] *Composition over inheritance.* [https://en.wikipedia.org/wiki/Composition\\_over\\_inheritance/](https://en.wikipedia.org/wiki/Composition_over_inheritance). [Accessed 2021.04.22].
- [18] *Single Responsibility Principle.* <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>. [Accessed 2020.09.22].

- [19] *Kodein DI*. <https://kodein.org/di/>. [Accessed 2020.09.22].
- [20] *Auth0*. <https://auth0.com/>. [Accessed 2020.12.14].
- [21] *Google*. <https://www.google.com/>. [Accessed 2020.12.14].
- [22] *JWT*. <https://jwt.io/>. [Accessed 2020.12.14].
- [23] *Heroku*. <https://www.heroku.com/>. [Accessed 2020.12.17].
- [24] *Room persistence library*. <https://developer.android.com/training/data-storage/room/>. [Accessed 2021.01.11].
- [25] *Dependency inversion principle*. [https://martinfowler.com/articles/dipInTheWild.html/](https://martinfowler.com/articles/dipInTheWild.html). [Accessed 2021.04.23].
- [26] *Kotest*. <https://kotest.io/>. [Accessed 2021.04.24].
- [27] *Kotlin coroutines library*. <https://kotlinlang.org/docs/coroutines-overview.html>. [Accessed 2021.02.19].
- [28] *Klock date time library*. <https://korlibs.soywiz.com/klock/>. [Accessed 2021.02.19].
- [29] *Kotlin serialization library*. <https://github.com/Kotlin/kotlinx.serialization/>. [Accessed 2021.02.19].
- [30] *Redux*. <https://redux.js.org/>. [Accessed 2021.02.22].
- [31] *Dukat*. <https://github.com/Kotlin/dukat>. [Accessed 2021.02.23].
- [32] *TypeScript*. <https://www.typescriptlang.org/>. [Accessed 2021.02.23].
- [33] *JSX*. <https://reactjs.org/docs/introducing-jsx.html>. [Accessed 2021.02.27].
- [34] *Dexie database*. <https://dexie.org/>. [Accessed 2021.05.04].
- [35] *Redux toolkit*. <https://redux-toolkit.js.org/>. [Accessed 2021.03.02].
- [36] *Redux thunk*. <https://github.com/reduxjs/redux-thunk/>. [Accessed 2021.03.02].
- [37] *Styled components*. <https://styled-components.com/>. [Accessed 2021.03.04].
- [38] *Material UI*. <https://material-ui.com/>. [Accessed 2021.03.04].
- [39] *Material UI React wrapper written in Kotlin*. <https://github.com/cfnz/muirwik/>. [Accessed 2021.03.04].
- [40] *React router*. <https://reactrouter.com/>. [Accessed 2021.03.04].
- [41] *Android architecture components*. <https://developer.android.com/topic/libraries/architecture>. [Accessed 2021.03.07].

- [42] *Android LiveData*. <https://developer.android.com/topic/libraries/architecture/livedata>. [Accessed 2021.03.09].
- [43] *Observer pattern*. [https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern). [Accessed 2021.03.09].
- [44] *Android view state*. <https://zsmb.co/designing-and-working-with-single-view-states-on-android/>. [Accessed 2021.03.09].
- [45] *JUnit 5*. <https://junit.org/junit5/>. [Accessed 2021.03.10].
- [46] *Android ViewModel*. <https://developer.android.com/topic/libraries/architecture/viewmodel/>. [Accessed 2021.03.10].
- [47] *Auth0 Android*. <https://auth0.com/docs/libraries/auth0-android/>. [Accessed 2021.03.10].
- [48] *Simple Stack*. <https://github.com/Zhuinden/simple-stack/>. [Accessed 2021.03.10].
- [49] *Retrofit*. <https://square.github.io/retrofit/>. [Accessed 2021.03.10].
- [50] *OkHttp*. <https://square.github.io/okhttp/>. [Accessed 2021.03.10].
- [51] *Coil*. <https://github.com/coil-kt/coil/>. [Accessed 2021.03.10].
- [52] *Mockk*. <https://github.com/mockk/mockk/>. [Accessed 2021.03.10].
- [53] *Coroutines testing*. <https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/>. [Accessed 2021.03.10].
- [54] *IntelliJ IDEA*. <https://www.jetbrains.com/idea/>. [Accessed 2021.05.21].
- [55] *Navigate to declaration issue*. [https://youtrack.jetbrains.com/issue/KTIJ-11683/](https://youtrack.jetbrains.com/issue/KTIJ-11683). [Accessed 2021.05.21].
- [56] *Live reloading*. <https://stackoverflow.com/questions/41428954/what-is-the-difference-between-hot-reloading-and-live-reloading-in-react-native/>. [Accessed 2021.05.21].
- [57] *Spring*. <https://spring.io/>. [Accessed 2021.05.22].
- [58] *Spring Kotlin*. <https://spring.io/guides/tutorials/spring-boot-kotlin/>. [Accessed 2021.05.22].
- [59] *AppBrain Kotlin using*. [<https://www.appbrain.com/stats/libraries/details/kotlin/kotlin>]. [Accessed 2021.05.22].
- [60] *ING using Kotlin*. <https://medium.com/ing-blog/introducing-kotlin-at-ing-a-long-but-rewarding-story-1bfcd3dc8da>. [Accessed 2021.05.22].

- [61] *Amazon using Kotlin.* <https://talkingkotlin.com/qlldb>. [Accessed 2021.05.22].
- [62] *Adobe using Kotlin.* <https://medium.com/adobetech/streamlining-server-side-app-development-with-kotlin-be8cf9d8b61a>. [Accessed 2021.05.22].
- [63] *Jetpack compose.* <https://developer.android.com/jetpack/compose>. [Accessed 2021.05.23].
- [64] *Compose for Desktop.* <https://www.jetbrains.com/lp/compose/>. [Accessed 2021.05.23].
- [65] *Compose for Web.* <https://blog.jetbrains.com/kotlin/2021/05/technology-preview-jetpack-compose-for-web/>. [Accessed 2021.05.23].
- [66] *Xamarin.* <https://dotnet.microsoft.com/apps/xamarin>. [Accessed 2021.05.23].
- [67] *React native.* <https://reactnative.dev>. [Accessed 2021.05.23].
- [68] *Flutter.* <https://flutter.dev/>. [Accessed 2021.05.23].