Responsive Multiplatform HMI-Anwendungen mit Kotlin Multiplatform und Jetpack Compose

Table of Contents

Mit fortgeschrittener Digitalisierung wachsen Anforderungen auch auf die HMI-Anwendungen in verschiedenen industriellen Bereichen. Nicht nur sicher, zuverlässig und schlank, aber auch modern, gut aussehend und vernetzt sollten sie werden. Mit einer einfachen Desktop-App, die mehrere Protokolle beherrscht, schauen wir uns in diesem Artikel die Stärken von Kotlin und Kotlin Multiplatform, wie man einfach mehrere Plattformen Desktop und Mobil (besonders mit Fokus auf Betriebssysteme von Apple) ohne Kompromissen unterstützen kann, wie wunderbar die Interoperabilität zwischen Kotlin und JVM funktioniert und wie deklarative UI Frameworks, wie Jetpack Compose und Swift UI, die Art und Weise verändern, wie man reaktive Bedienoberfläche heutzutage entwickelt.

IoT Explorer

Um die Konzepte und Ideen für eine moderne Bedienoberfläche zu validieren, implementieren wir als ein Konzept eine App, die man zum Betrachten und verändern von Werten angebundener Geräte nutzen kann. Die Kommunikation mit den Geräten wird über maschinenorientierte Protokolle realisiert. Die Geräte können Maschinen, industrielle Anlagen oder Endverbrauchergeräte sein. Der Kommunikation bringt mehrere Herausforderungen und wir schauen uns die Möglichkeiten genauer an, die uns erlauben Bibliotheken, die für das jeweilige Protokoll am besten geeignet sind in die unsere App zu integrieren.

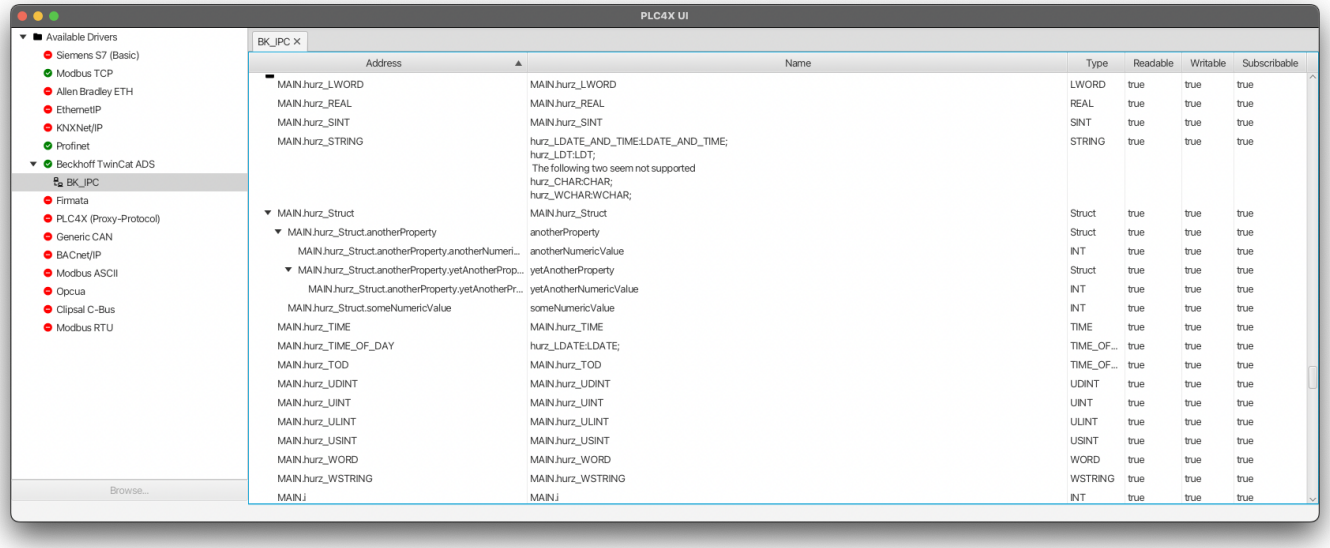


Figure 1. Hauptbildschirm

Das Bild 1 zeigt den Hauptbildschirm der App. In dem linken Teil, in einer Baumansicht sind Geräte nach Protokollen gruppiert. Die dazugehörigen Werte und zugehörige Möglichkeiten sie zu verändern, sind als eine Liste in rechten Teil des Fensters dargestellt. Wichtig ist, dass die Werte im Gerät selber, bei der Übertragung und wenn sie in der App angezeigt werden unterschiedliche Representation haben können. Genauso wichtig sind die physikalischen Einheiten und die Formatierung, es könnte sogar möglich sein, dass in den unterschiedlichen Teilen der App der gleiche Wert eine unterschiedliche Darstellung haben kann (zum Beispiel als 0 und 1 oder grafische Darstellung als ein Schalter).

Wenn wir über der Implementierung der App weiter nachdenken sollten wir auch berücksichtigen, wie schnell sich manche Werte verändern, ob sie beim Gerät angefragt werden muss, oder eine Wertveränderung automatisch übermittelt wird. Ein weiters Spezifikum der App ist der Datenfluß: Die App is Daten getrieben, der Update von Werten muss automatisch stattfinden und soll einfach zu implementieren sein.

Geteilte Freude ist doppelte Freude

Eines der Ziele, die wir uns für die App gesetzt haben war, dass die App auf unterschiedlichen Betriebssystemen und Endgeräten laufen soll. Eine separate App für jede Platform ist einerseits aufwendig, und das aus vielen Gründen (Entwickler:innen mit entsprechenden Fähigkeiten, Kosten, Zeit oder Wartbarkeit etc). Auf der anderen Seite steht die native Leistung oder die Art wie sich die App in ihren jeweiligen Ökosystemen anfühlt oder integrieren lässt. Die Suche nach dem perfekten Weg ist schon alt. Für Mobile Apps liegt der Anfang vermulich bei HTML-basierten Web-Apps (Cordova, Ionic), mit schwergewichtigen Cross Platform Frameworks wie Xamarin, bis zu hybriden Ansätzen wie React Native, Vue Native oder Flutter. Alle haben ihre Vor und Nachteile. Als eine ernstzunehmende Alternative zeichnet sich Codesharing mit Kotlin in Multiplatform-Projekten ab, wo in Kotlin geschriebener Code auf unterschiedlichen Plattformen laufen kann. Das Thema ist nicht neu und wurde schon öfter behandelt (Beispielsweise in Java Aktuell 01 2020). Mit der Zeit haben sich die Vorteile für Code Sharing mit Kotlin nicht nur bestätigt, sondern es sind auch mit weitere neue dazu gekommen, wie zum Beispiel Jetpack Compose.

Kotlin und Kotlin Mutliplatform

Warum überhaupt Kotlin? Code sharing ist nichts Neues und funktioniert auch mit anderen Programmiersprachen. Zunächst ist es Kotlin als Sprache selbst: Kotlin bezeichnet sich als kompakt und

präzise, sicher, ausdrucksvoll, vollständig kompatibel und multiplatform fähig. Schon aleine die ersten Eigenschaften machen Kotlin gerade für Java Entwickler:innen interessant. Und für unsere IoT App spielt gerade die Multiplatorffähigkeit eine bedduetende Rolle. Ein wichtiger Teil der Kotlin multiplatformfähig macht, sind in die Sprache selber integrierte Schlüsselwort **expected/actual**, die ein fester Bestandteil der Programmiersprache sind.

Ein weiterer wichtiger Baustein, der das Bauen von reaktiven Apps vereinfacht und zur Sprache selber gehört, sind coroutines und Flows. Generell läuft die Kommunikation mit den Geräten im Hintergrund, dazu kommt dass wir mit mehrere Geräten gleichzeitig Daten austauschen wollen. Das wird durch Einsatz asynchroner Programmierung gelöst, bei der die einzelnen Abfragen nicht mehr hintereinander, sondern parallel ausgeführt werden. Das Aktualisieren der Bedienoberfläche findet nur in einem (anderen) Thread statt. Solcher Code ist normalerweise nicht einfach zu schreiben, und wenn dann auch noch Bildschirmrotation hinzukommt, die eine neue Anzeige des Objekts erfordert (bekannt als Activity Lifecycle) sind die schlaflosse Nächste beim Debuggen vorprogrammiert.

Kotlin löst das Problem mit dem Schlüsselwort **suspend** und führt ein Programmierparadigma benannt als **Strukturierte Parallelität** (structured concurency) ein. Die Hauptidee ist das Kapseln der Ausführung von parallelen Ausführungen in eigenen Blöcken, so dass der zuständige Code zusammen mit einem klarem Anfang und Ende geschrieben ist. Dies erzeugt nicht nicht nur besere Lesbarkeit und Fehlerbehandlung, sondern auch besseren Resource Management mit Resourcen wie zum Beispiel Threads.

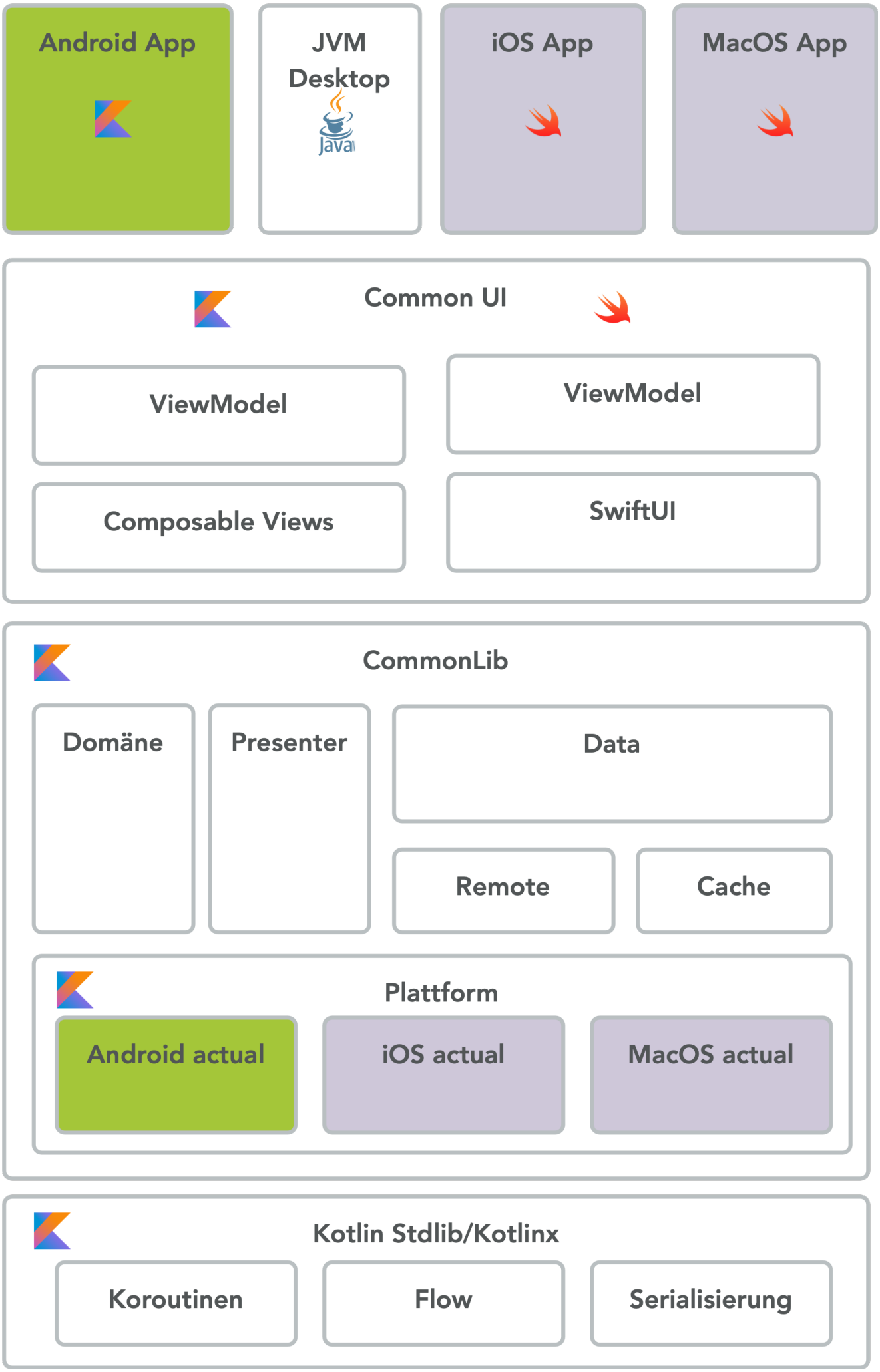
Eine vollständige kompatibilität zum JVM war und bleibt eins der Hauptmerkmale vom Kotlin und erlaubt uns auch beim Thema Kotlin Multiplatform die Integration von JVM Bibliotheken. Gerade in Bereich maschinenorientierter Protokolle ist die Auswahl von qualitativ hochwertigen Bibliotheken sehr gut. Als Beispiele sind die Bibliotheken für die Kommunikation in KNX oder das Apache Projekt PLC4X mit mehreren SPS Protokollen im Portfolio(siehe Projekt Homepage https://plc4x.apache.org/)

Kotlin hat noch ein weiteres Ass im Ärmel: Kotlin/Native. Kotlin/Native is eine Technologie, die das Kompilieren vom Kotlin Code in den nativen binären Code erlaubt, der ohne eine virtuelle Maschine laufen kann. Kotlin/Native beinhalted ein LLVM basiertes Backend für den Kotlin Kompiler und eine native Implementierung der Kotlin Standard Bibliothek. Das ist genau der Weg, wie wir die nativen iOS und macOS Apps in Kotlin bauen können.

Softwarearchitektur

Die Softwarearchitektur einer App trägt maßgeblich dazu, wie viel von dem in Kotlin geschriebenen Code wiederverwendet kann. Beim Entwurf einer Softwarearchitektur für ein Kotlin-Multiplatform-Projekt steht neben den üblichen nichtfunktionalen Anforderungen wie Lesbarkeit oder Wartbarkeit auch der hohe Grad an Wiederverwendbarkeit von Code im Vordergrund. Eine einmal programmierte Lösung spart nicht nur Aufwand nach dem DRY-Prinzip („Don’t Repeat Yourself“), sondern kann so auch eine höhere Qualität und fachliche Korrektheit durch mehrfaches Anwenden erreichen. Für die IoT Explorer App haben wir uns an eine stark an „Clean Architecture“[6] angelehnt. Die Tatsache, dass aus Sicht der Clean Architecture die UI Android Frameworks nur am Rande stehen und der Fokus auf Entities und Use Cases in der Domain liegt, macht es einfach, eine Umsetzung unabhängig von der Plattform zu gestalten.

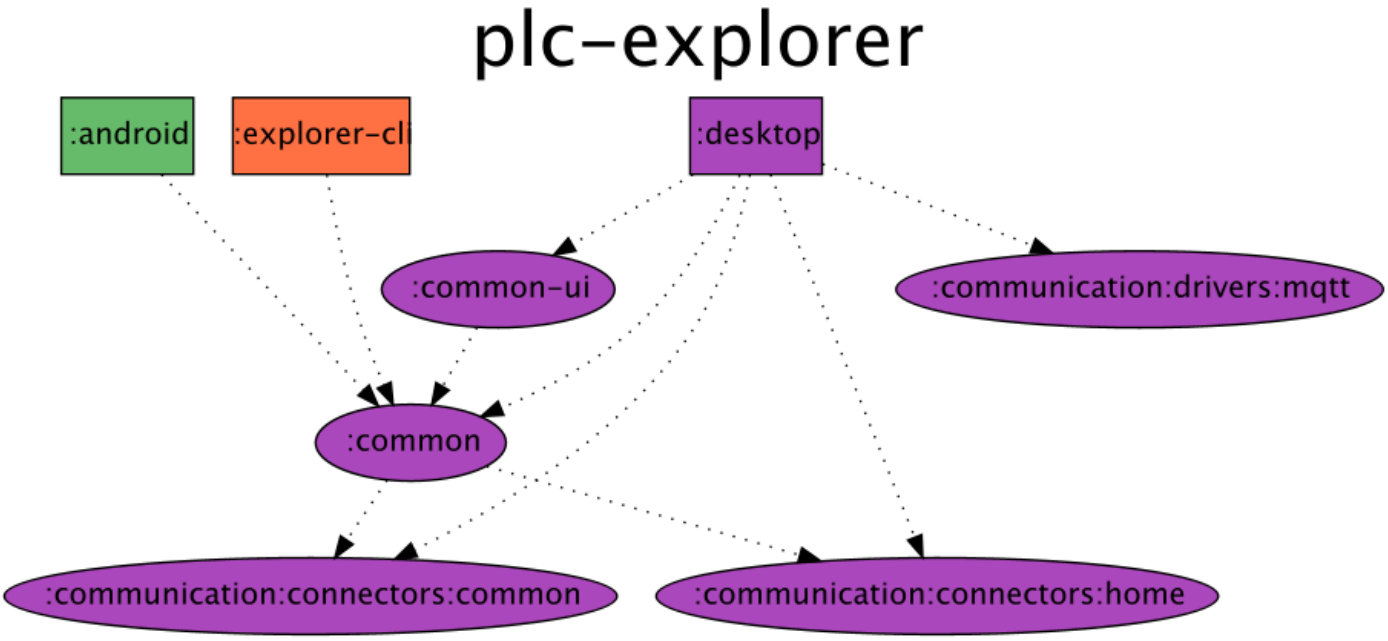
Kotlin multiplatform


Android App
JVM Desktop
iOS App
MacOS App
Common UI
ViewModel
ViewModel
Composable Views
SwiftUI
CommonLib
Domäne
Presenter
Data
Remote
Cache
Plattform
Android actual
iOS actual
MacOS actual
Kotlin Stdlib/Kotlinx
Koroutinen
Flow
Serialisierung

Das Herzstück ist die CommonLib, die den Teil des Kodes beinhaltet, den man auf unterschiedlichen Platformen benutzt werden kann. Zusätzlich beinhaltet sie die tatsächliche Implementierung für native Code Teile, die nicht mit Kotlin umgesetzt werden können. Es steht auf soliden Grundlagen der Standard Biblithek mit den Implementirugen für „kotlinx"-Serialisierung. Koroutinen werden zusätzlich zur Kotlin Standard Library mit FLow benutzt und in Abbildung 1 als ein wichtiger Baustein dargestellt. Durch eine weitere Trennung der Implementierung, die sich um die Netzwerkkommunikation kümmert, und den Code, der für das Caching und die lokale Datenspeicherung zuständig ist, erzeugen wir zwar einen kleinen Mehraufwand (zum Beispiel bei den notwendigen ModelMappers), erreichen dafür allerdings die Freiheit, unterschiedliche Datenklassen für Netzwerk und Speicher zu nutzen. Dies wiederum ermöglicht eine Austauschbarkeit von Kommunikationsmodulen für unterschiedliche Protokolle mit unterschiedlichen Datentransfer-Objekten und Endpunkten.

Gradle

Gradle ist weiterer Baustein, der die IoT Explorer App in dieser Form möglicht macht und spielt in diesem Projekt seine Stärken aus. Die erste Stärke ist die Möglichkeit, die Baukette mit eigenen Plugins zu erweitern. An dieser Stelle kommt die Kotlin Compiler Plugin Extension für Jetpack Compose zum Einsatz. Eine weitere Stärke vom Gradle sind die Gradle Module. Mit Modulen erreichen wir eine physische Trennung zwischen Schichten in der Software Architektur, sodass man gezwungen ist sich an den Kontrakt der Schicht zu halten und nicht auf die Idee kommt, zum Beispiel im Code für das UI schnell mal eine Backendabfrage abzufeuern. Durch Gradle Module ist das auch das Teilen vom gemeinsamen Code für die jeweiligen Platformen möglich.



Auf dem Bild 2(Gradle modules) sind die Kotlin Multiplatform Module farbig hervorgehoben. Mit Linien sind die Abhängigkeiten zwischen dann einzelnen Modulen dargestellt.

Inzwischen unterstützt Gradle auch Kotlin als eine Skriptsprache, so es möglich ist die Konfigurationsskripte direkt in Kotlin zu schreiben, was gerade in den Entwicklungsumgebungen, die Kotlin unterstützen einen Vorteil ist. Und seit dem Android Plugin in der Version 4.0 unterstützt auch Google Kotlin als die Skriptsprache, für Gradle und es ist sogar der empfohlene Weg.

Bedienoberflächen mit Jetpack Compose

Jetpack compose ist ein Bestandteil der von Google als **Jetpack** bezeichneten Bibliotheken, die helfen sollen,

Best Practices zu befolgen und Boilerplate-Code zu reduzieren. Dies ist eigentlich eine nüchterne Beschreibung von Google, in Wirklichkeit ist Jetpack Compose ein vollständiger Paradigmenwechsel, wie Bedienoberflächen programmiert werden, weg vom Imperativen Ansatz über XML zu einem mit Kotlin geschriebenen deklarativen Code, mit allen Vorteilen, die ein Einsatz einer vollwertigen Programmiersprache bietet.

Um zu verstehen, warum Jetpack Compose die passende Lösung für unsere IoT Explorer App ist, wir schauen zuerst das Code-Listing 2.

Jetpack Compose DevicesTreeView

```
@Composable
fun DevicesTreeView(viewModel: DevicesTreeViewModel) = Surface {
    val viewState by viewModel.state.collectAsState()
    Box {
        if (viewState.items.isNotEmpty()) {
            LazyColumn {
                items(viewState.items.size) {
                    DeviceTreeItemView(viewState.items[it])
                }
            }
        }
    }
}
```

Das Erste, was unsere Aufmerksamkeit wecken sollte, ist die erste Zeile mit der Annotation *Composable*. Composable Funktionen sind die Grundbausteine zum Definieren der Bedienoberfläche. Die Markierung einer Funktion (oder einer Lambdafunktion) als Composable weist darauf hin, dass die Funktion/Lambda ein Teil der Composition ist und beschreibt eine Transformation der Daten in eine Baumdatenstruktur. Diese ist durch das UI Framework zum Rendern der Bedienoberfläche benutzt. Als weiteres können wir die kompakte Form bemerken, die durch Einsatz von Lambda Funktionen und der Unterstützung von Kotlin für Type Builders (die gerne zum Definieren eigenen DSL Sprache benutzt werden) ermöglicht wird. Wichtig ist auch *collectAsState*: Damit wird eine Flow Liste mit Geräten emittiert, und somit werden Änderungen im Datenmodel automatisch in die Oberfläche gesendet.

Jetpack Compose ist nicht nur UI

Wichtig bei Jetpack Compose ist, dass das Framework selbst aus zwei Hauptteilen besteht: Jetpack Compose und Jetpack Compose UI. Der Core hat keine Abhängigkeiten zu Android als Betriebsystem (obwohl die Klassen im Package-name *AndroidX* leben). Weder hat es eine Beziehung zur UI. Es ist nur zuständig für die Jetpack Compose Runtime und das Verwalten von der Baumdatenstrukturen, die für den State zuständig sind. Daraus endstand auch die Idee, die Fähigkeiten von Jetpack Compose auf weitere Platrofmen zu übertragen, wodurch Jetpack Compose Desktop und Jetpack Compose Web entstanden sind. Es existiert auch eine Variante für das Terminal mit dem Namen Mosaic, und es gibt auch erste Anzeichen, dass auch eine Portierung für iOS kommen könnte. Jetpack Compose kommt zwar von Google und ist primär zum Bauen von Bedienoberflächen auf Android ausgelegt, aber durch die konsequente Trennung zwischen Core und UI ermöglich es die Umsetzung einer weiteren, pfiffigen Idee. Wenn wir uns erinnern, was der Core Teil des Jetpack Compose macht, also das Verwalten vom State einer Baumdatenstruktur, und wenn wir uns überlegen, wie man den Zustand eines physikalischen Gerätes am besten abbilden kann, kommt man wieder zu Baumdatenstrukturen. So ist die Bibliothek Molecule (Link X) entstanden, die Kotlin Multiplatform-fähig ist. Ein Beispiel, wie ein Composable Presenter aussehen könnte, ist in Listing (y) dargestellt. Diese Composable

Funktionen können in der Domainschicht unserer Architektur liegen, weil sie keine Abhängigkeiten an UI oder Betriebssystem haben.

Composable

```
sealed interface DeviceModel {
    object Loading : DeviceModel
    data class Data(val name: String) : DeviceModel
}

@Composable
fun DevicePresenter(
    deviceFlow: Flow<Device>
): DeviceModel {
    val device by deviceFlow.collectAsState(null)
    return if (device == null) {
        Loading
    } else {
        Data(user.name, balance)
    }
}
```

UI

Wie es aus dem Architekturbild ersichtlich ist, Block zuständig für die Bedienoberfläche benannt CommonUI, nutzt gemeinsames Code geschrieben in Kotlin am alle Platformen, die Jetpack Compose in einer von den unterstützten Variante unterstützt. Wir haben als erstes die IoT Explorer App für Desktop als JVM gebaut und mit einer Variante für MacOS X experimentiert.

ViewModel

ViewModels haben sich als eine perfekte Begleitung für reaktive Bedienoberflächen erwiesen. Das gilt nicht nur für Jetpack Compose, sondern genau so für SwiftUI. Als Implementierung haben wir zum Kotlin Multiplatform Konzept von expect/actual gegriffen. So ist es zum Beispiel möglich für Android eine native Variante zu nutzen, die Application Lifecycle berücksichtigt.

ViewModel

```
expect open class CommonViewModel() {
    val clientScope: CoroutineScope
    protected open fun onCleared()
}
```

ViewModel on Android

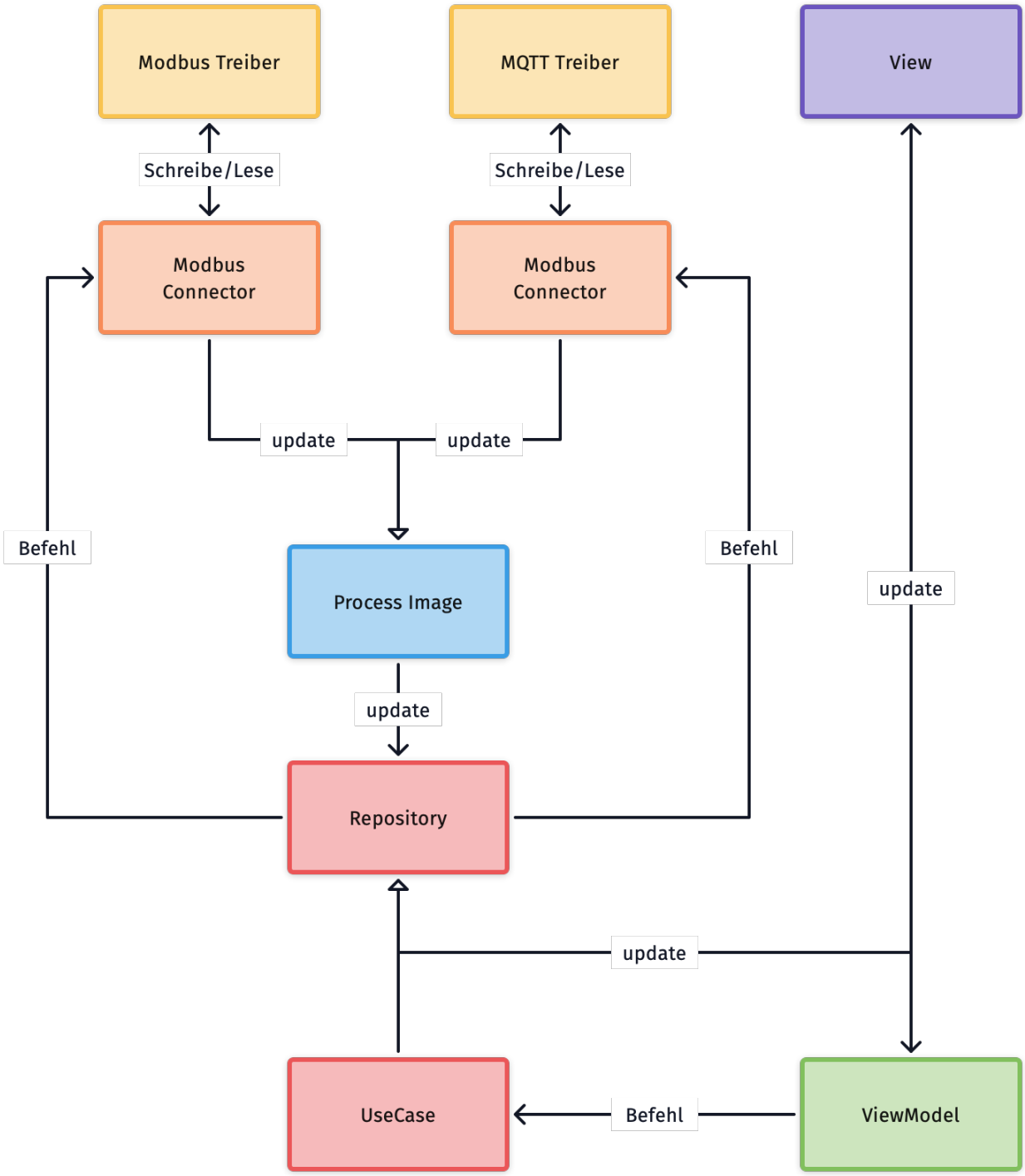
```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.CoroutineScope

actual open class CommonViewModel actual constructor() : ViewModel() {
    actual val clientScope: CoroutineScope = viewModelScope
    actual override fun onCleared() {
        super.onCleared()
```

```
    }
}
```

Es hat sich als eine gute Idee erwiesen, die ViewModels so definieren, dass der Konstruktor sehr wenig oder keine Parameter hat, um eine Möglichkeit zu haben, die ViewModel Objekte auch in den Composable Funktionen zu instanziieren. Dabei hat sich *dependency injection* als ein gangbarer Weg ergeben, um notwendige Objekte in ViewModels zu referenzieren. Hier können wir den Vorteil nutzen, dass Kotlin und Kotlin Multiplatform auch von einer wunderbaren Community leben und somit könnten wir *Koin*(link X), als ein Kotlin Multiplatform fähiges Framework einsetzen.

Unidirectional Data Flow

Reaktive Oberflächen leben davon, dass die Daten in der UI nur in einer Richtung laufen, sich die Anzeige anhand der Deklaration durch Recomposition wieder anpasst und die neue UI gerendert wird. Wie können wir aber die Daten verändern oder wie gehen wir mit Benutzereingaben oder UI Ereignissen um? Wenn wir bei der IoT Explorer App bleiben: Um den Zustand eines Gerätes, zum Beispiel einen Schalter aktivieren oder um die Rollladen im Smarthome runterzufahren, muss entsprechendes Kommando über ein Protokoll, welches das Gerät versteht, geschickt werden. Das ist deutlich mehr, als eine Bedienoberfläche wissen sollte. In diesem Fall nutzen wir wieder ein weiters Konzept aus der Clean Architecture, den UseCase. Die Implementierung davon lebt wieder in der Domain Schicht, hat kein Wissen über der Bedienoberfläche und muss nur das Object kennen, das mit dem Gerät kommuniziert.



iOS

Technische integration

Kotlin on iOS

Kotlin Multiplattform hat einen anderen Ansatz als beispielsweise ReactNative oder Flutter: Während sich mit diesen Frameworks vor allem User Interfaces für verschiedene Platformen bauen lassen, überlässt KMP genau das dem nativen Code. Das ermöglicht es nicht nur, auf jeder Platform alle Möglichkeiten zu nutzen, es erlaubt auch, insbesondere auf Apple Platform, die freie Wahl des Frameworks: KMP arbeitet genau so gut mit UIKit

zusammen wie mit SwiftUI. Technisch stellt sich auf iOS Seite der Kotlin Multiplattform Teil als Cocoapods Dependency dar, und der Kotlin Code wird direkt aus Xcode gebaut — es ist nicht notwendig, erst mit einer anderen Toolchain eine Library ohne ähnliches zu bauen. Das ermöglicht es auch den iOS Entwickler:innen, schnell und einfach Änderungen am Kotlin Code vorzunehmen, und selbst zu testen, und möglicherweise ins Repository zurückzuschreiben. Technisch wird KMP als eine CocoaPods Dependency in ein Xcode Projekt eingefügt. Beim Anlegen eines KMP Projekts in Android Studio wird auch ein Xcode Workspace erzeugt, im Prinzip kann aber auch ein bestehender oder anderer Workspace den Kotlin Code integrieren. Das bedeutet auch, dass die Xcode Dateien völlig unabhängig von der Android Version sind, und somit beliebig umbenannt oder verändert werden können. Die vom Kotlin Code bereitgestellten Interfaces lassen sich aus Swift direkt aufrufen, und auch problemlos in SwiftUI Integrieren. Es können also zum Beispiel im Shared Kotlin Code direkt ViewModels erzeugt werden, um dann entsprechend angezeigt zu werden. Kotlin Multiplattform Code muss aber nicht immer gleich sein: Selbst in der geteilten Bibliothek kann es unterschiedlichen Code für Android und iOS geben, der dann trotzdem mit einem einheitlichen API an die jeweiligen Plattformen weitergegeben wird. Das ist insbesondere deshalb interessant da hier im iOS Teil auch auf Apple Frameworks zugegriffen werden kann. Beispielsweise wäre es möglich, mit Kotlin für iOS schon mit den System Frameworks von iOS lokalisierte Zeit und Datumsangaben zu übergeben, während das auf Android mit den entsprechenden Frameworks funktioniert. Aber hier hören die Möglichkeiten nicht auf: Es lassen sich sogar für den Kotlin Code native Bibliotheken für iOS mit CocoaPods einbinden, und dann mit Kotlin aufrufen. Dies erlaubt es, auch bei klarer Trennung der Aufgaben plattformspezifisch im gemeinsamen Code zu arbeiten. Wer sich die Integration etwas genauer ansieht stellt fest, dass Kotlin in der Apple Welt noch über eine Objective-C Brücke integriert wird. Während es damit zwar auch möglich wäre, Kotlin aus einer Objective-C App aufzurufen, ist der Hauptgrund vermutlich, dass das Swift Application Binary Interface (ABI), dass also die Schnittstelle zu Swift definiert, vor noch nicht all zu langer Zeit nicht stabil war, sondern sich noch regelmäßig geändert hat. Ob es in der Zukunft ein wirklich natives Interface zu Swift geben wird, ist noch unklar. Das erzeugt an manchen Stellen etwas Reibung: Kotlin Coroutines werden auf iOS als Funktionen mit einer Closure umgesetzt, da Objective-C dieses Konzept von asynchronen Funktionen nicht kennt. Wer möchte, kann diese Funktionen natürlich in Swift in Combine Publisher, oder auch als Async/Await Funktionen verpacken. Damit aber auch Cancellations aus dem Kotlin Code an Swift übergeben werden können, ist ein Plugin nötig, das die entsprechende Funktionalität von Kotlin über Objective-C in Swift transportiert. Kotlin Multiplattform ist extrem flexibel, und wirkt auf jeder Platform heimisch. Durch eine sehr aktive Community und die Unterstützung von JetBrains ist zu erwarten, dass dies in den kommenden Jahren noch besser wird.

Integration ins development teams

Kotlin Multiplattform hat aber nicht nur technische Eigenschaften, sondern auch etwas, was „Soft Skills“ gleichkommt. Ein zentrales Problem bei vielen Cross-Platform Projekten ist weniger die verwendete Technology, als das Staffing —also die Auswahl der Entwickler:innen. Nicht selten wird ein Projekt ausschliesslich mit zum Beispiel JavaScript oder Dart Expert:innen besetzt, in der Hoffnung, dass diese die gesamte Entwicklung bis hin zum Deployment übernehmen können. Das mag bei einfachen Anwendungen funktionieren; viele Projekte stellen sehr schnell fest, dass es ohne genaue Kenntnisse der Platform nicht funktioniert. Alle Cross Platform Tools sind auf Plugins zur nativen Platform angewiesen, wenn sie etwas spannendere Dinge tun möchten — zum Beispiel per Bluetooth kommunizieren oder Passwörter sicher abspeichern. Wenn es dann ein Problem mit diesen Plugins gibt, egal ob durch die Verwendung, neue SDKs oder neue Toolchains, ist genaues Wissen über das Zielsystem und die verwendeten Programmiersprachen unerlässlich. Insgesamt können OS Updates, neue SDKs und Werkzeuge immer für Überraschungen sorgen. Und auch beim Deployment in AppCenter, TestFlight, Google Play oder den AppStore kann es Feinheiten geben, für die es spezialisiertes Wissen braucht. Wie löst Kotlin Multiplattform dieses Problem? Nicht durch Technik, jedenfalls nicht durch neue Technik: KMP setzt konsequent auf Tools, die Android und iOS Entwickler:innen absolut geläufig sind: Die IDEs Android Studio, Xcode sind ihre täglichen Werkzeuge, und der

Paketmanager CocoaPods ist bei der iOS Entwicklung ausserordentlich populär. Das bedeutet, dass Teams wirklich nur die Dinge neu lernen müssen, die absolut notwendig sind. Sie können sich sonst in den Umgebungen bewegen, die ihnen vertraut sind, und mit denen sie sich auskennen. Kotlin und Swift sind sich als Sprachen sehr ähnlich in Syntax und Konzepten, wer die eine kann, kann die andere zumindest lesen und auch leicht erlernen. Und weil bei KMP nicht nur einfach Binärdateien über den Zaun geworfen werden, kann gemeinsam genutzter Code auch von allen angesehen, debugged und geändert werden. Abschliessend bleibt zu sagen, dass sogar der Kotlin Compiler Konan einfach nur ein Frontend für llvm ist, der „Low Level Virtual Machine“ die auch schon das Backend für Swift und Apples C-Compiler Clang ist. Das macht auf iOS vieles einfacher, auch weil damit der Standard Debugger lldb automatisch mit diesen Binaries umgehen kann. So ermöglicht es KMP, Projekte direkt mit Profis für die Plattformen zu besetzen, die gleichzeitig auch zusammen an Cross Platform Code arbeiten können.

Zusammenfassung

Mit Kotlin und Kotlin Multiplatform haben wir ein Konzeptapp entwicklet, welhce die Vorteile der Sprache als moderne und gut lesebare Programmiersprache dokumentieren soll. Es ist immer wieder erfreulich festzustellen, wie mächtig Kotlin und die Standard Bibliothen von Kotlin sind, sodass man mit hauseigenen Mitteln wie coroutines und Flow früher sehr aufwendig zu implementierende Konzepte wie asynchrone Programmierung elegant umsetzen kann. Jetpack Compose sogar weiter und mit der Unterstützung von Kotlin ist es möglich, reaktive Bedienoberflächen deklarativ zu entwickeln. Die Idee, das man kann auch an nicht visuelle Elemente der App die Composable Funktionen mit ihrer Fähigkeit, einen Status zu halten, zu verwenden und auf Veränderung mit recomposition zu reagieren, zeigt auch die Stärke von Jetpack Compose. Ein Zusammenspiel mit einer durchdachten Architektur macht das Entwicklen einer moderner HMI App möglich. Neben der Technik haben wir auch festgestellt wie wichtig die SoftSkills sind und wie sich Kotlin in multiplatform Teams anfühlt und etablieren kann.