

Multiplatform App Development in Kotlin

Appdevcon 2024

About Your Instructor - Pamela

- Kotlin Developer Advocate at JetBrains



- From Pretoria, South Africa



About Your Mentor - Kostya

- Software Developer in Kotlin Multiplatform at JetBrains
- Lives in Leiden, Netherlands



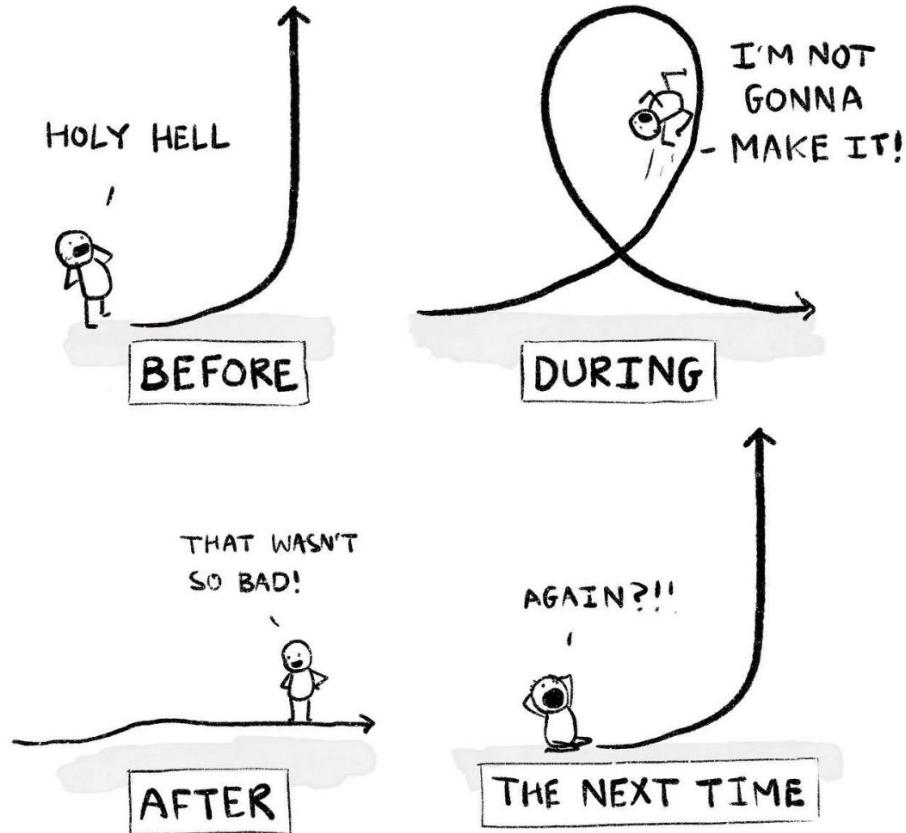
Morning Agenda

- Setup verification
- Welcome and speaker introduction
- What is Kotlin Multiplatform?
- UIs with Compose Multiplatform
- Networking with Ktor, serialization with `kotlinx.serialization`
- The `expect / actual` mechanism

Afternoon Agenda

- Testing in KMP Applications
- Persistence
- Coroutines and Flows
- Sharing View Models
- Dependency Injection with Koin
- Discussion
- Closing

HOW LEARNING CURVES FEEL...



Break Schedule

- 08:30: Welcome and registration
- 09:00: Workshop starts
- 10:30-10:45 Informal break
- 12:30-13:30: Lunch
- 15:00-15:15 Informal break
- 17:00: Workshop ends

What is Kotlin Multiplatform?

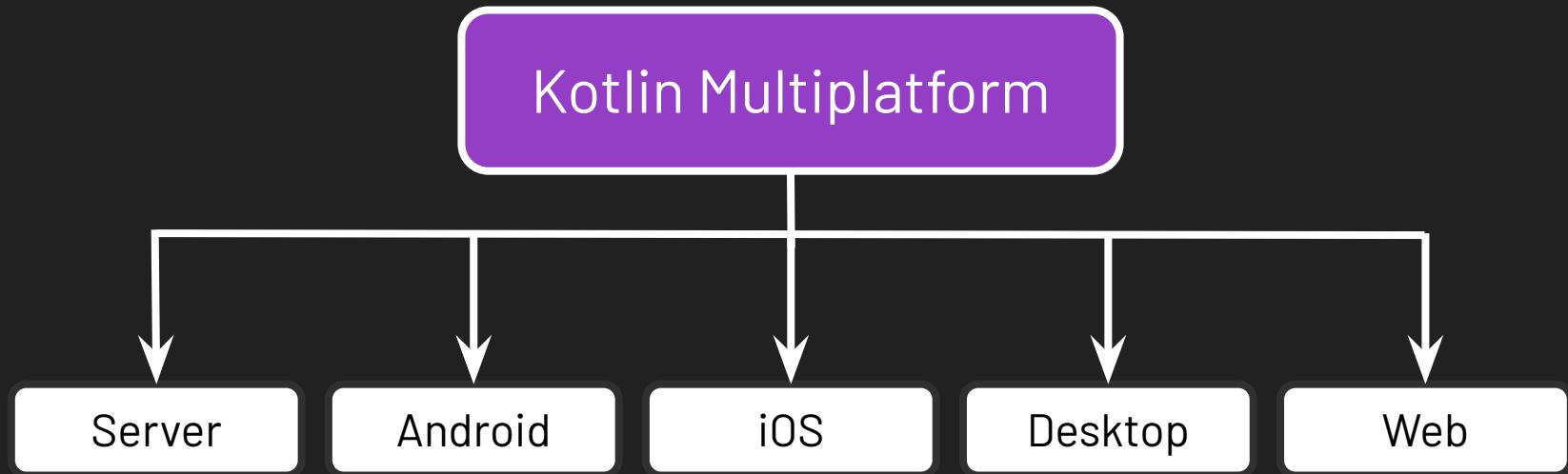
What is Kotlin Multiplatform?

- Technology by JetBrains
- Allows development teams to share common code between platforms
- Doesn't impose restrictions on non-shared code

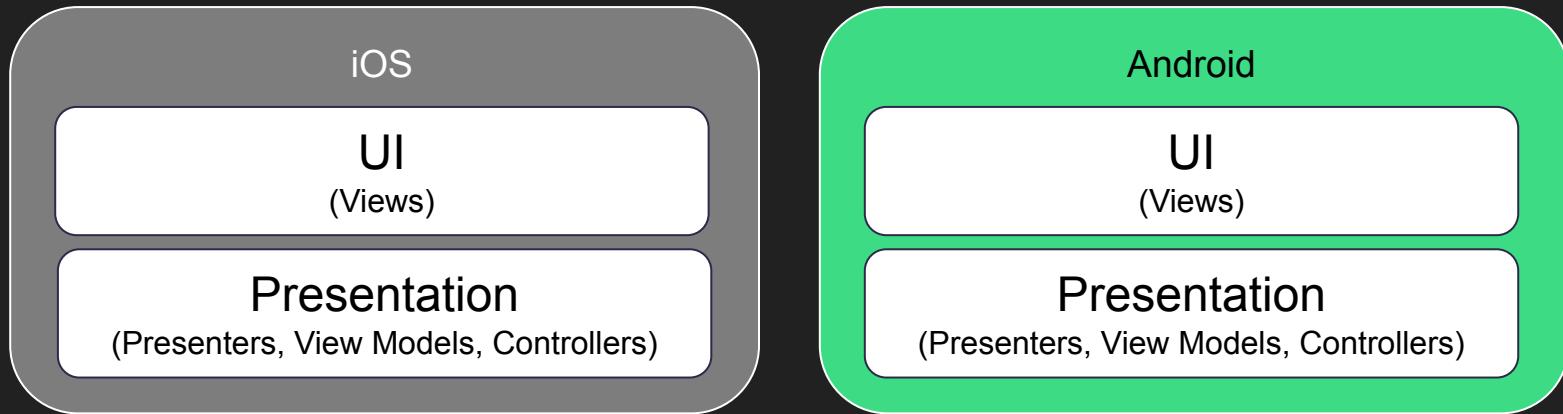
What is Kotlin Multiplatform?

- Technology by JetBrains
- Allows development teams to share **common code** between **platforms**
- Doesn't impose restrictions on non-shared code

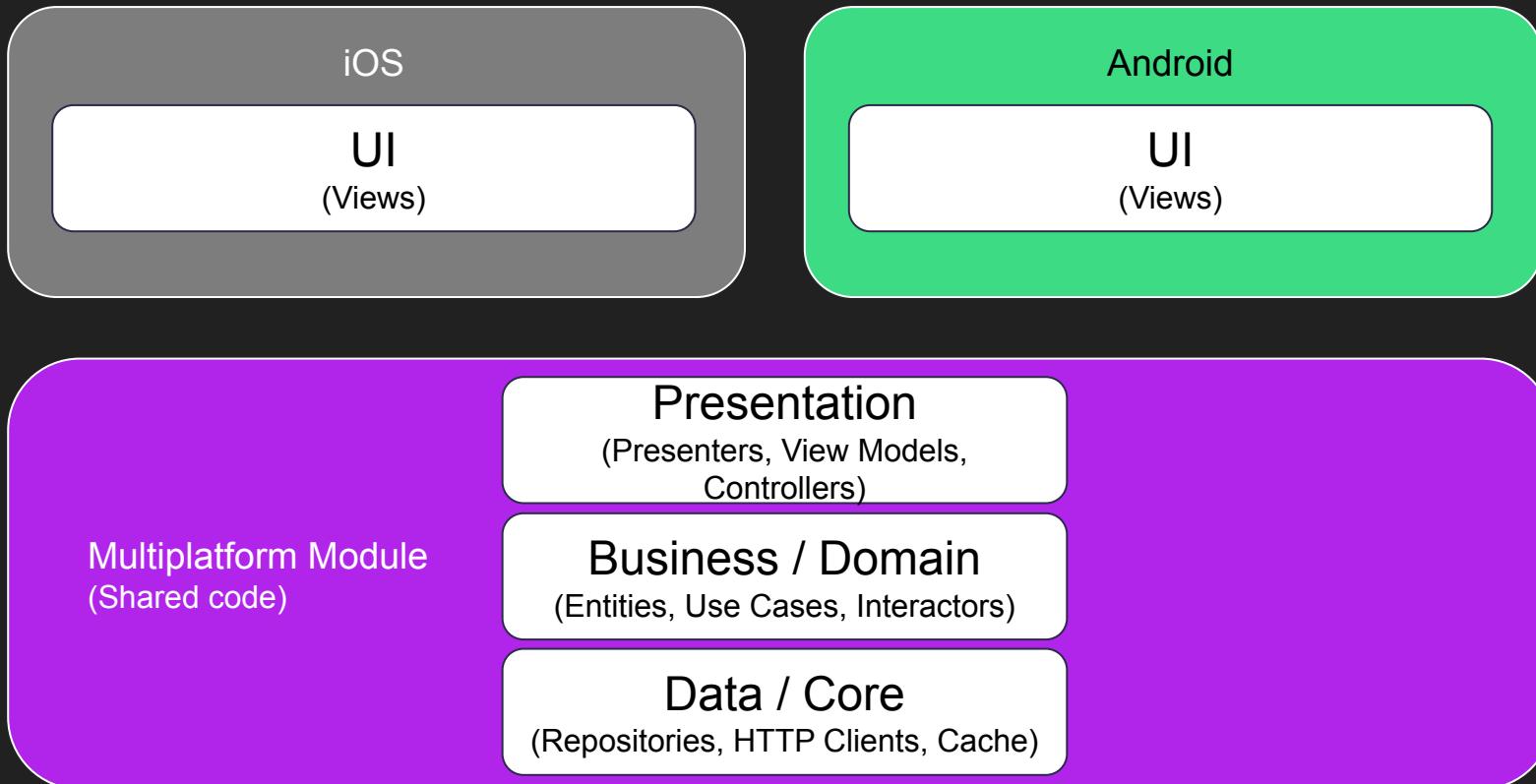
What kind of platforms?



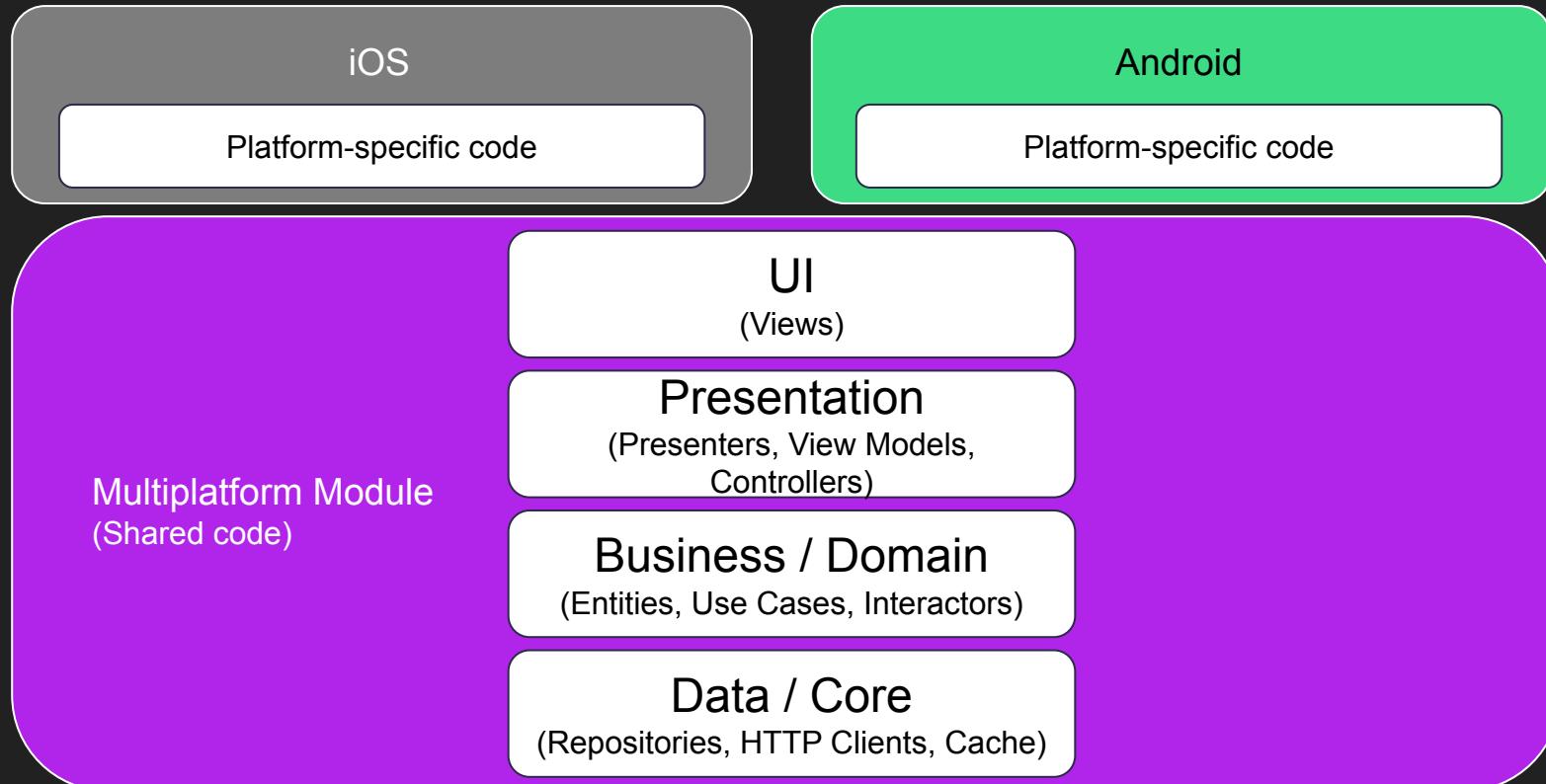
What kind of common code?



What kind of common code?



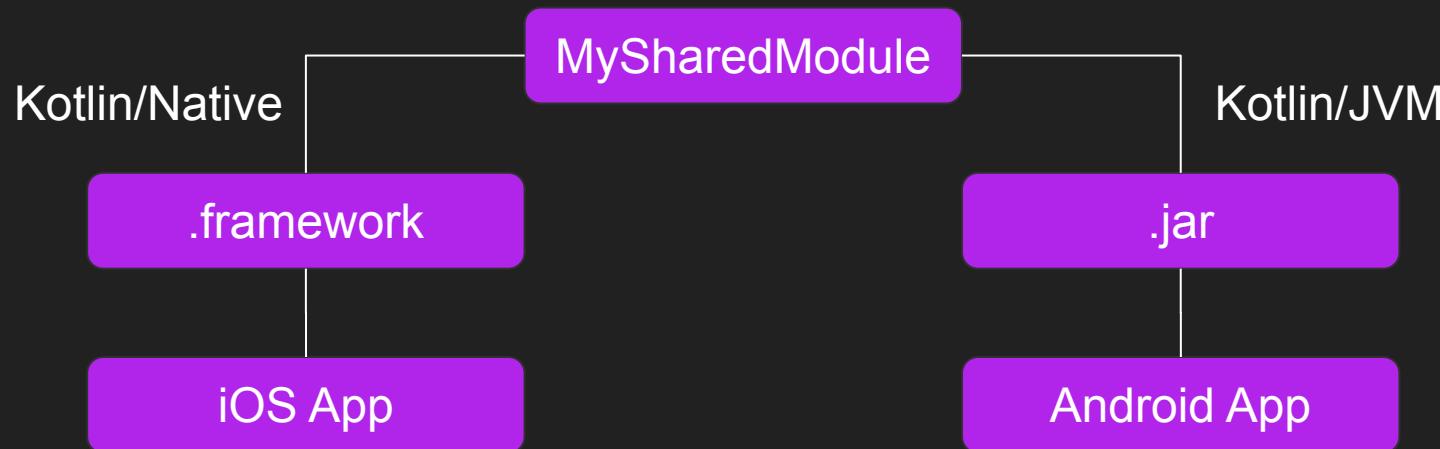
What about ✨*Compose Multiplatform*✨?



Kotlin Multiplatform == Sharing

- With Kotlin Multiplatform you can share what you want, eg:
 - Networking
 - Data storage & validation
 - Business logic etc.
- And write the rest natively - just as before

How does Kotlin Multiplatform work?



What are the benefits of Kotlin Multiplatform?

- **Flexibility** to choose what is shared and what isn't
- Choose to share in a **new or existing** project
- Shared code lowers the **effort and cost** per feature
- Shared code ensures **consistency** amongst platforms
- Access to platform capabilities **without overhead**

Demo: Using Project Wizards

Introducing Compose Multiplatform

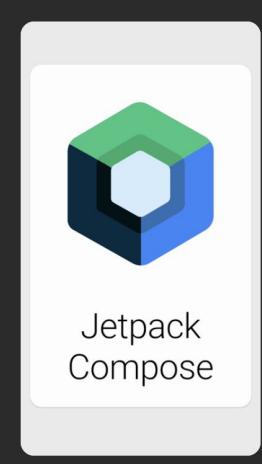
The TL;DR on Compose

- Compose lets you create awesome UIs
- Compose Multiplatform is produced by **JetBrains**
- Built on Kotlin Multiplatform by **JetBrains**
- Powered by Jetpack Compose from **Google**

Build better apps faster with Jetpack Compose

Jetpack Compose is Android's recommended modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Quickly bring your app to life with less code, powerful tools, and intuitive Kotlin APIs.

```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = "Jetpack Compose",
                    style = MaterialTheme.typography.bodyLarge,
                )
            }
        }
    }
}
```



<https://developer.android.com/jetpack/compose>

JET BRAINS

Compose Multiplatform

Develop stunning shared UIs for Android, iOS, desktop, and web.

Get started

Image Viewer

20. Mar Mountain K2

Note

K2 at 8,848 meters above sea level, is the second-highest mountain on Earth, after Mount Everest... (see more)

Related memories

Place

Mountain K2

Kina The Calico

Blue City

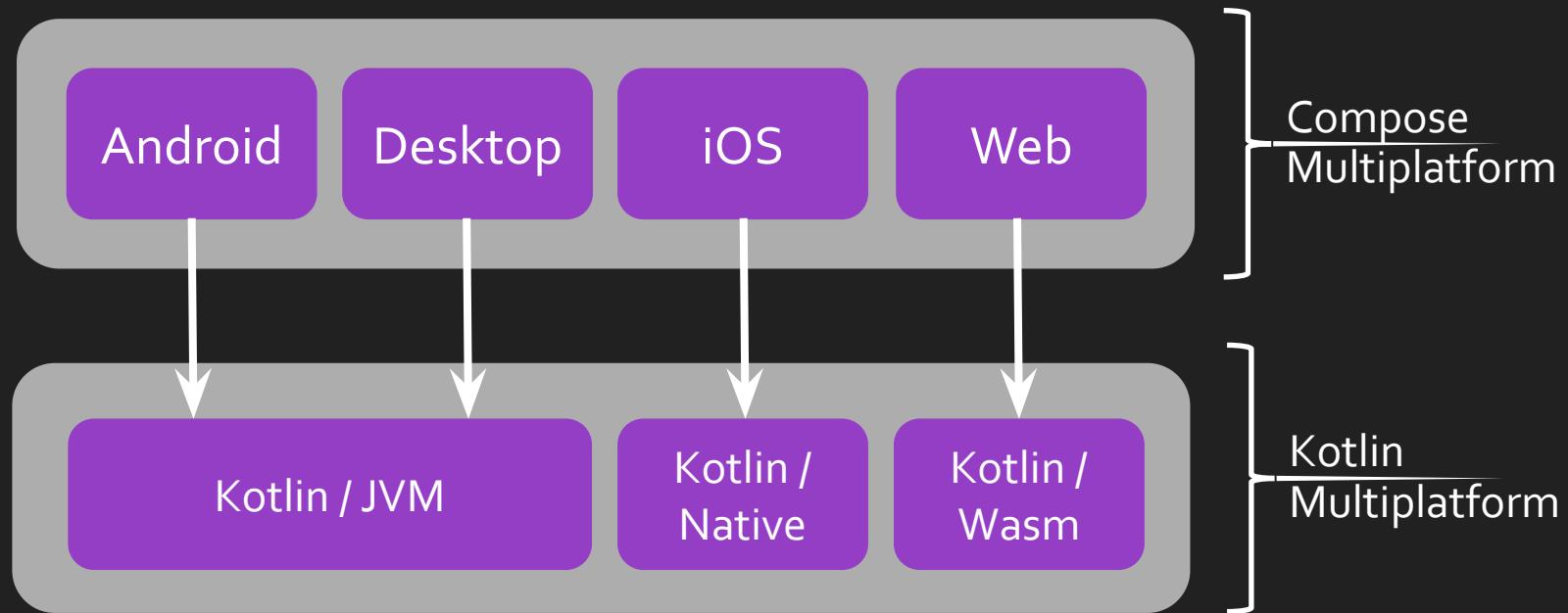
Tokyo Skytree

Taranaki

Auckland Sky City

<https://jb.gg/compose>

Compose & Kotlin Multiplatform



What is Wasm?

- WebAssembly (Wasm) is a binary instruction format for portable virtual machines
- Smaller, faster than JavaScript
- Write code in languages like Kotlin / Go / Rust, compile to Wasm
- Already supported in most modern browsers, Node.js

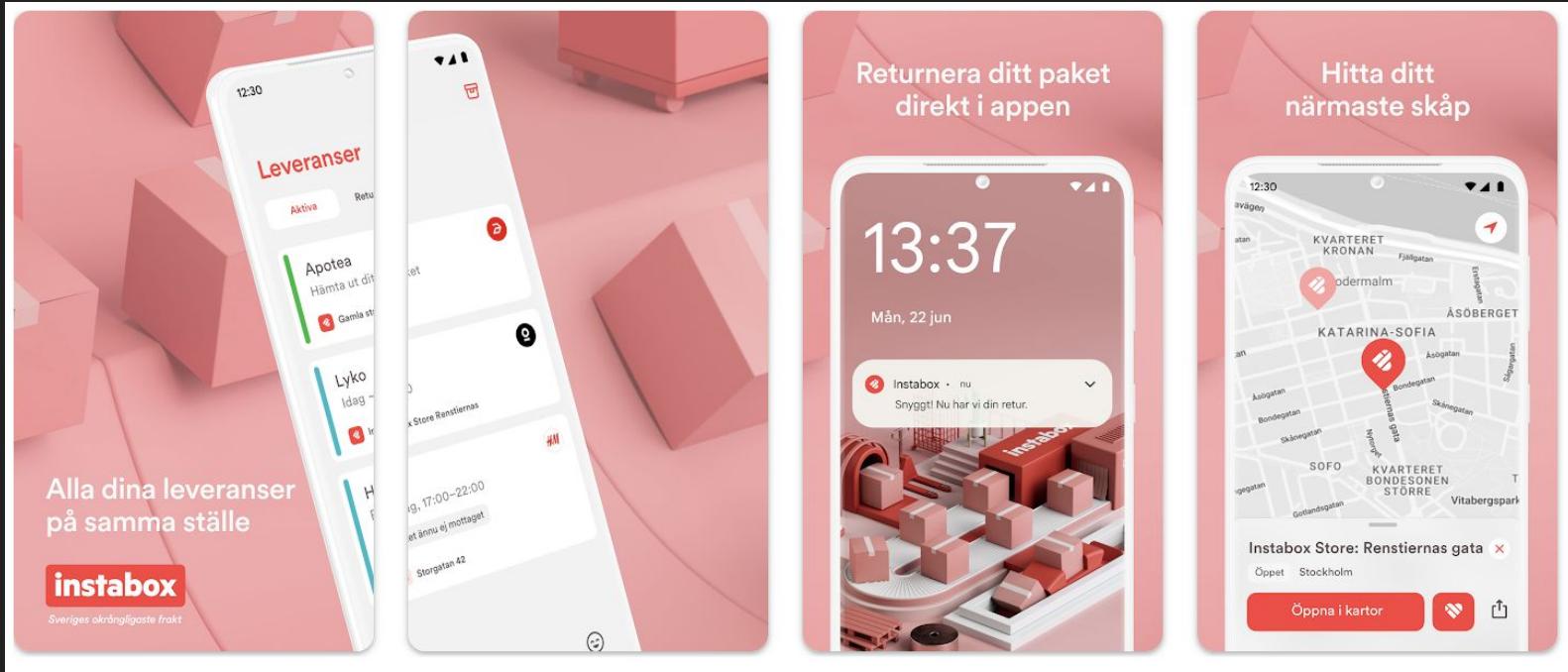
The screenshot shows a web browser window with the JetBrains website open at jetbrains.com. The main heading is "Compose for Desktop". Below it is a purple hexagonal icon representing the toolkit. A sub-headline reads: "Fast reactive desktop UIs for Kotlin, based on Google's [modern toolkit](#) and brought to you by JetBrains." A paragraph explains: "Compose for Desktop simplifies and accelerates UI development for desktop applications, and allows extensive UI code sharing between Android and desktop applications." At the bottom is a blue "Getting started" button.



The screenshot shows the JetBrains Toolbox application interface. The top navigation bar includes "Tools", "Projects", "Services", and a search bar. The main area is titled "Installed" and lists several JetBrains tools with their icons and versions: Rider 2022.3.2, IntelliJ IDEA Community Edition 2022.3.3, PhpStorm 2022.3.3, and PyCharm Community Edition 2022.3.3. Below this, a section titled "Available" lists other tools with "Install" buttons: Fleet (PREVIEW), Aqua (New PREVIEW), IntelliJ IDEA Ultimate, and Android Studio by Google.

Instabee

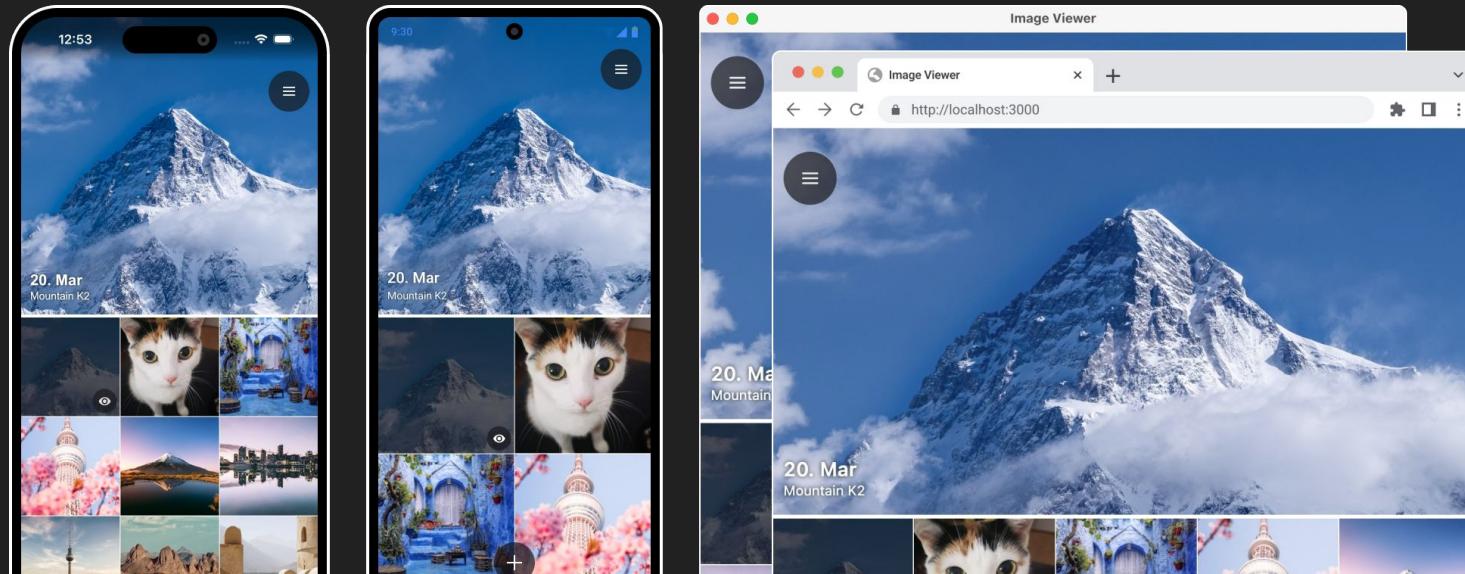
Built on Compose Multiplatform (Android and iOS)



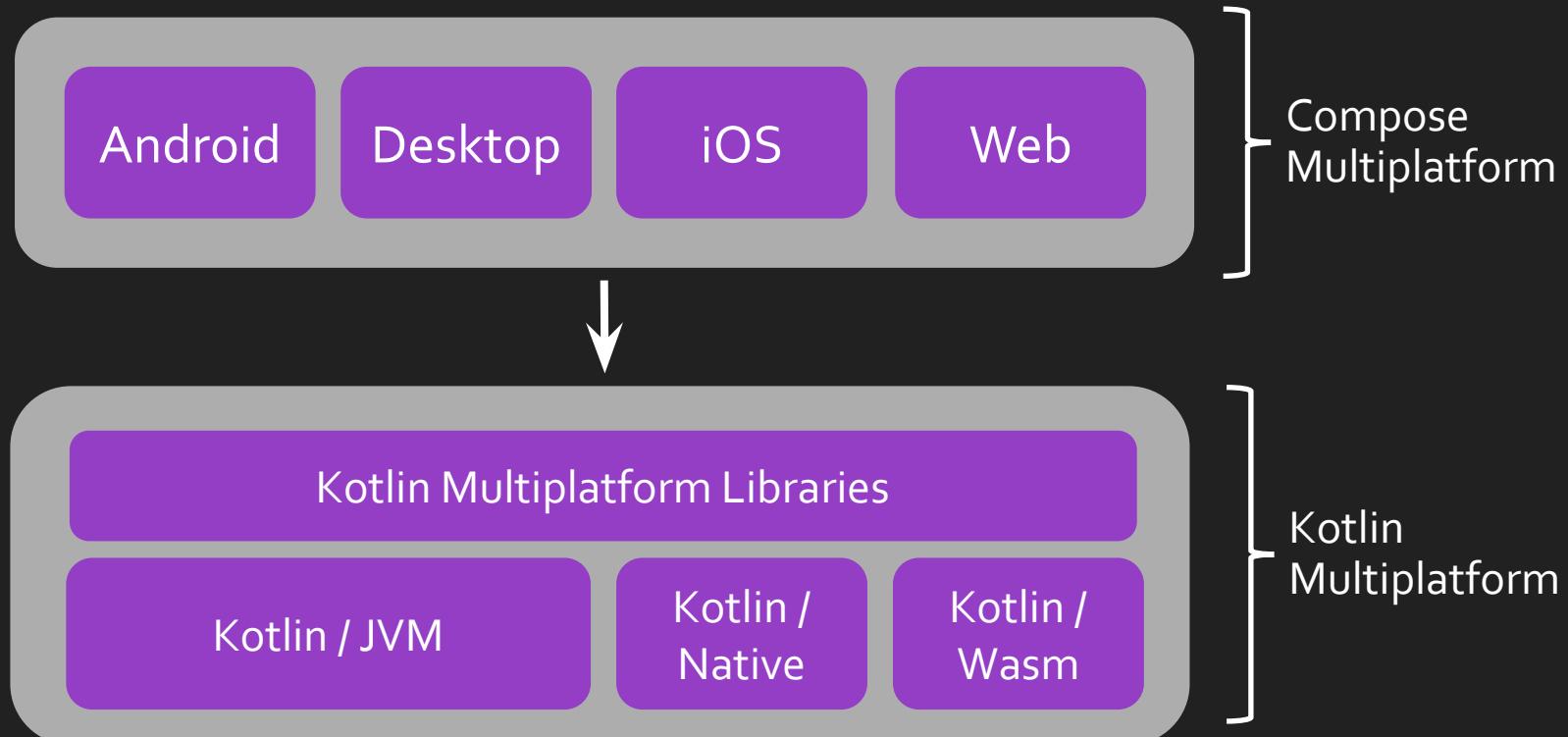
The ImageViewer Sample Application

An example image gallery with camera and map support

Built on Compose Multiplatform (Desktop, Android and iOS)



Compose & Kotlin Multiplatform & Libraries



Some Kotlin Multiplatform Libraries

Koin	Dependency Injection framework
Ktorfit	Adds declarative functionality to Ktor, similar to Retrofit
KMM-ViewModel	Allows sharing of ViewModels across Android and iOS
KMP-NativeCoroutines	Allows coroutines to be used / cancelled in Swift code
SQLDelight	Generates type safe Kotlin API's from SQL
Multiplatform Settings	Simplifies persisting key/value pairs in multiplatform code
Jetpack DataStore	Stores key/value pairs and objects using coroutines and flows
Circuit	Architectural framework for Compose Multiplatform

github.com

Awesome KMM

Native Code

Shared Code

Business logic and core

View

View

iOS specific APIs

Android specific APIs

PRs welcome awesome stars 1.3k maven-central v1.8.20

Kotlin Multiplatform Mobile (KMM) is an SDK designed to simplify creating cross-platform mobile applications. With the help of KMM, you can share common code between iOS and Android apps and write platform-specific code only where it's necessary. For example, to implement a native UI or when working with platform-specific APIs.

Resources

1.3k stars
48 watching
60 forks
Report repository

Releases 9

Issue 9 Latest last month

+ 8 releases

Contributors 23

+ 12 contributors

<https://github.com/terrakok/kmp-awesome>

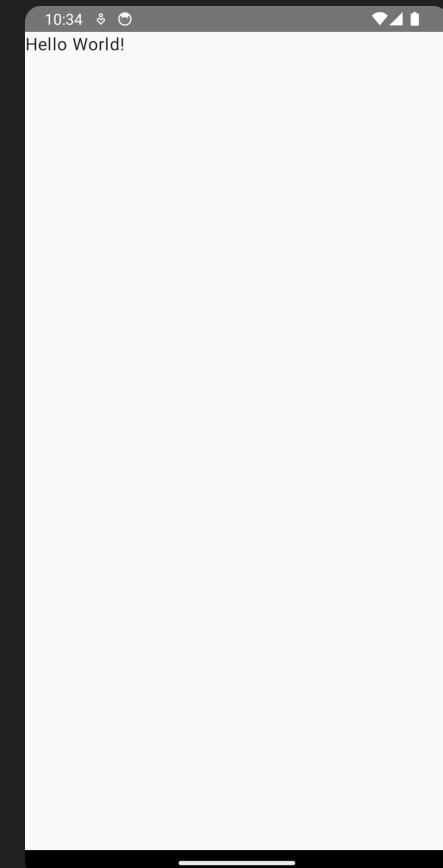
Hello Compose Multiplatform 🙌

Concepts

- Composable functions
- Layouts
- Images
- Themes
- Lists
- Buttons

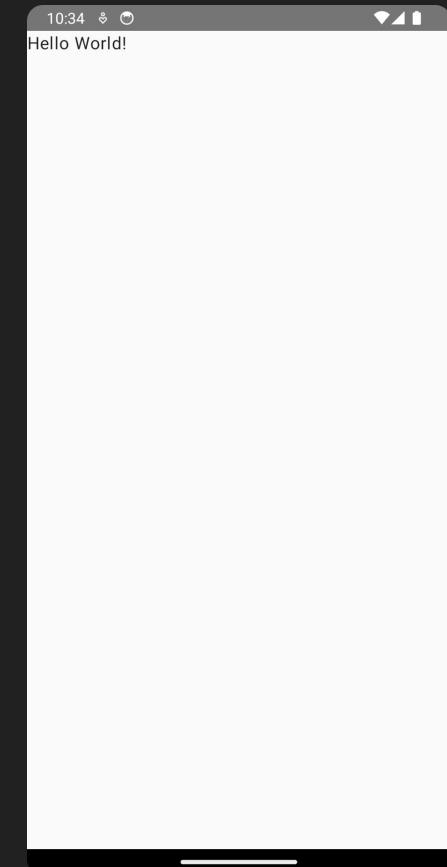
Adding a Text Composable

```
@Composable  
fun App() {  
    MaterialTheme {  
        Text("Hello World!")  
    }  
}
```



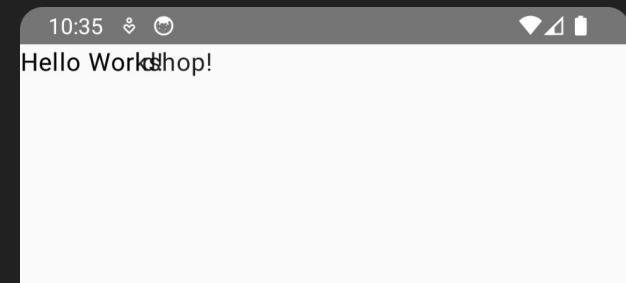
Defining Your Own Composable

```
@Composable  
fun App() {  
    MaterialTheme {  
        Hello()  
    }  
}  
  
@Composable  
fun Hello(){  
    Text("Hello World!")  
}
```



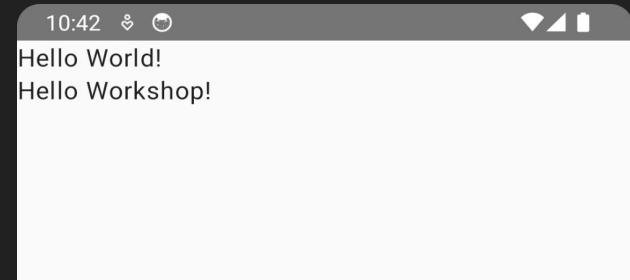
Adding Multiple Texts

```
@Composable  
fun Hello(){  
    Text("Hello World!")  
    Text("Hello Workshop!")  
}
```



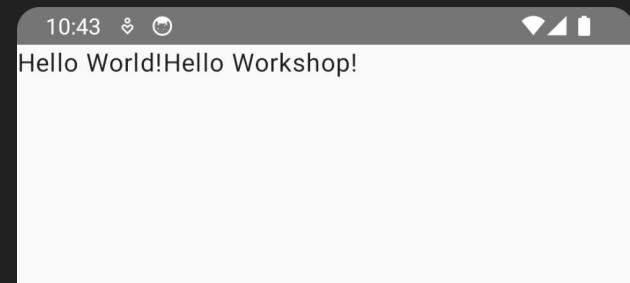
Adding Multiple Texts - Vertical

```
@Composable  
fun Hello(){  
    Column {  
        Text("Hello World!")  
        Text("Hello Workshop!")  
    }  
}
```



Adding Multiple Texts - Horizontal

```
@Composable  
fun Hello(){  
    Row {  
        Text("Hello World!")  
        Text("Hello Workshop!")  
    }  
}
```



Adding an Image

```
PokemonImage("https://unpkg.com/pokeapi-sprites@2.0.2/sprites/pok  
emon/other/dream-world/87.svg", "A white pokemon")
```

```
@Composable  
fun PokemonImage(url: String, altText: String){  
    val painterResource =  
        rememberImagePainter(url = url)  
    Image(  
        painter = painterResource,  
        contentDescription = altText,  
        contentScale = ContentScale.FillBounds  
    )  
}
```



```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

Configuring Padding

```
@Composable
fun Hello() {
    Column(modifier = Modifier.padding(32.dp)) {
        Text("Hello World!")
        Text("Hello Workshop!")
    }
}
```

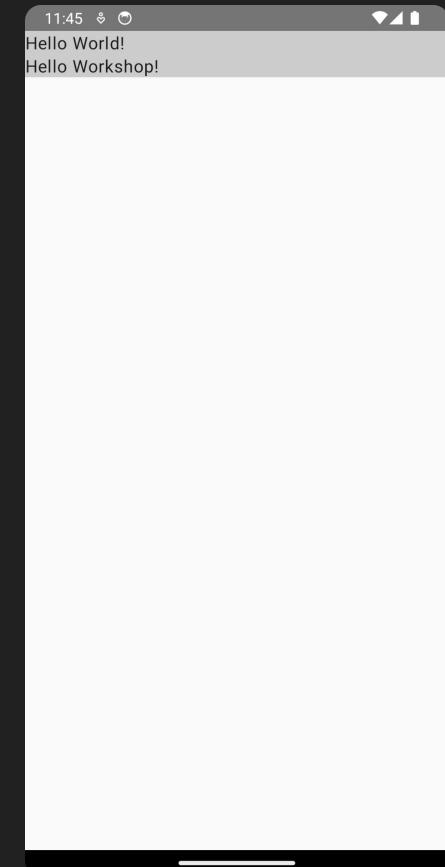
Hello World!
Hello Workshop!

Configuring Sizing - Width

```
@Composable  
fun Hello() {  
    Column(  
        modifier = Modifier.fillMaxWidth()  
            .background(Color.LightGray)  
    ) {  
        Text("Hello World!")  
        Text("Hello Workshop!")  
    }  
}
```

OR

```
modifier = Modifier.width(130.dp)
```

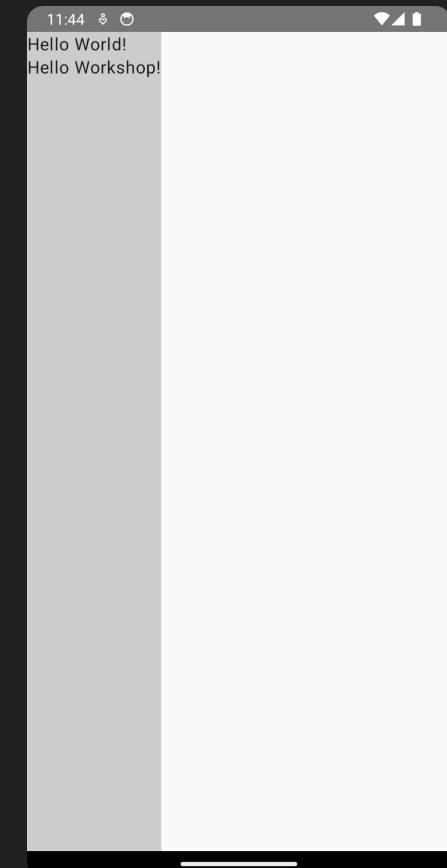


Configuring Sizing - Height

```
@Composable
fun Hello() {
    Column(
        modifier = Modifier.fillMaxHeight()
            .background(Color.LightGray)
    ) {
        Text("Hello World!")
        Text("Hello Workshop!")
    }
}
```

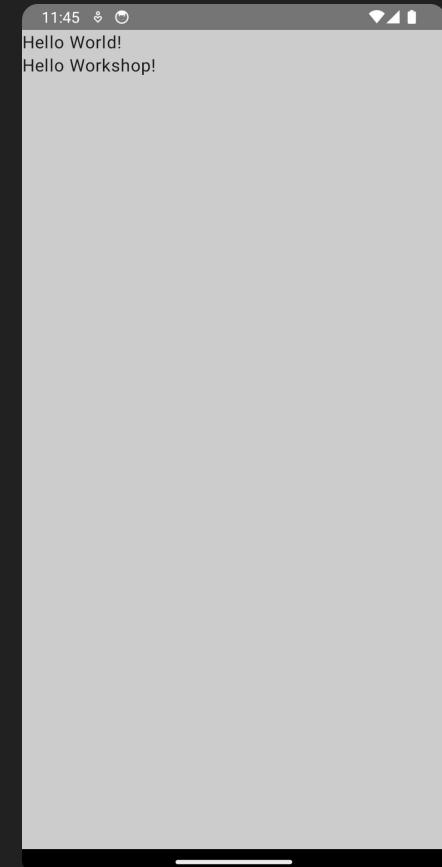
OR

```
modifier = Modifier.height(128.dp)
```



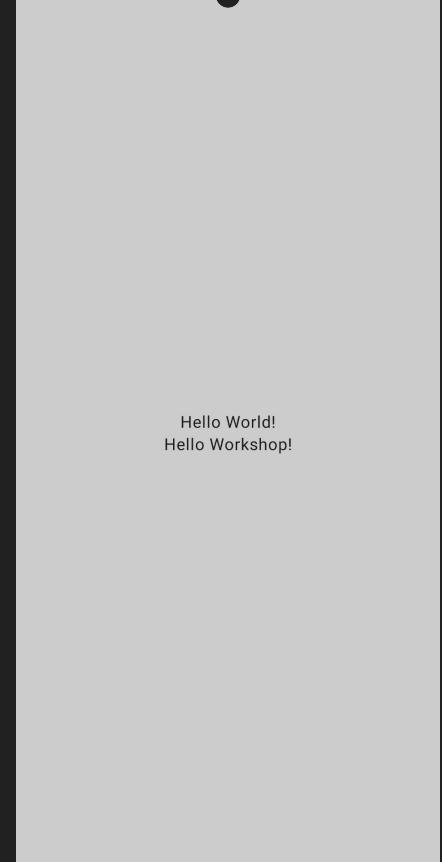
Configuring Sizing - Size

```
@Composable
fun Hello() {
    Column(
        modifier = Modifier.fillMaxSize()
            .background(Color.LightGray)
    ) {
        Text("Hello World!")
        Text("Hello Workshop!")
    }
}
```



Centering

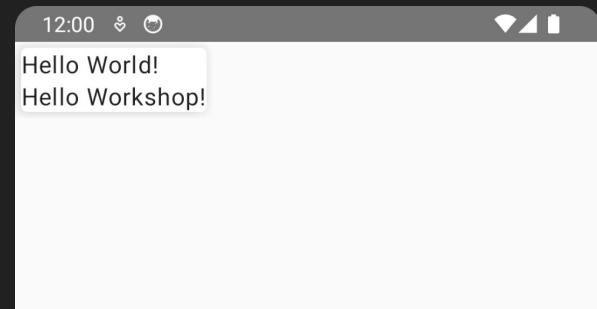
```
@Composable
fun Hello() {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier
            .fillMaxSize()
            .background(Color.LightGray)
    ) {
        Text("Hello World!")
        Text("Hello Workshop!")
    }
}
```



Hello World!
Hello Workshop!

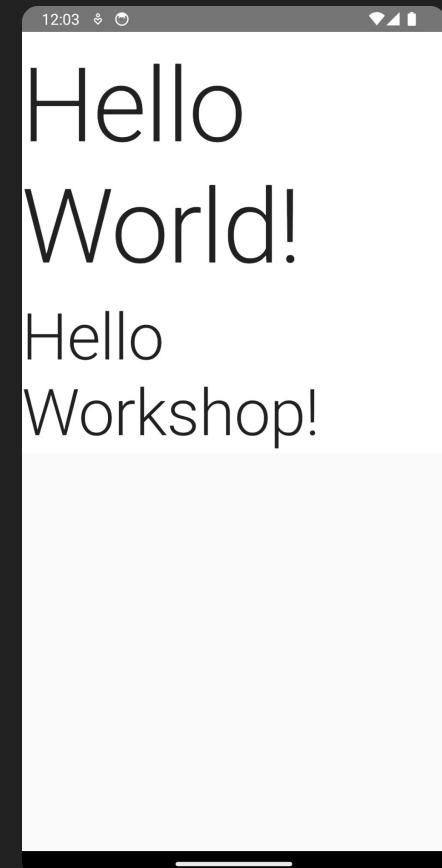
Creating a Card

```
@Composable
fun Hello() {
    Card(
        modifier = Modifier.padding(4.dp),
        elevation = 10.dp,
        shape = MaterialTheme.shapes.small
    ) {
        Column {
            Text("Hello World!")
            Text("Hello Workshop!")
        }
    }
}
```



Formatting Text

```
@Composable
fun Hello() {
    Column {
        Text(
            "Hello World!",
            style = MaterialTheme.typography.h1
        )
        Text(
            "Hello Workshop!",
            style = MaterialTheme.typography.h2
        )
    }
}
```



Formatting Text

Scale Category	Typeface	Weight	Size	Case	Letter spacing
H1	Roboto	Light	96	Sentence	-1.5
H2	Roboto	Light	60	Sentence	-0.5
H3	Roboto	Regular	48	Sentence	0
H4	Roboto	Regular	34	Sentence	0.25
H5	Roboto	Regular	24	Sentence	0
H6	Roboto	Medium	20	Sentence	0.15
Subtitle 1	Roboto	Regular	16	Sentence	0.15
Subtitle 2	Roboto	Medium	14	Sentence	0.1
Body 1	Roboto	Regular	16	Sentence	0.5
Body 2	Roboto	Regular	14	Sentence	0.25
BUTTON	Roboto	Medium	14	All caps	1.25
Caption	Roboto	Regular	12	Sentence	0.4
OVERLINE	Roboto	Regular	10	All caps	1.5

Lists - Number of Items

```
@Composable
fun HelloList() {
    LazyColumn {
        items(count = 100) {
            Text("This is item: $it")
        }
    }
}
```

```
This is item: 0
This is item: 1
This is item: 2
This is item: 3
This is item: 4
This is item: 5
This is item: 6
This is item: 7
This is item: 8
This is item: 9
This is item: 10
This is item: 11
This is item: 12
This is item: 13
This is item: 14
This is item: 15
This is item: 16
This is item: 17
This is item: 18
This is item: 19
This is item: 20
This is item: 21
This is item: 22
This is item: 23
This is item: 24
This is item: 25
This is item: 26
This is item: 27
This is item: 28
This is item: 29
This is item: 30
This is item: 31
This is item: 32
This is item: 33
This is item: 34
This is item: 35
This is item: 36
This is item: 37
This is item: 38
```

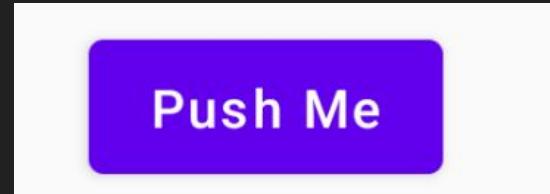
Lists - Collection

```
@Composable
fun HelloList() {
    val countries = listOf<String>(
        "Denmark",
        "South Africa"
    )
    LazyColumn {
        items(countries) {
            Text("This is country: $it")
        }
    }
}
```

This is country: Denmark
This is country: South Africa

Buttons

```
@Composable  
fun HelloButton() {  
    Button(onClick = { doSomething() }) {  
        Text("Push Me")  
    }  
}
```



Practical 1: Creating a Compose Multiplatform Card [20 mins]

remember stores values
across recompositions.

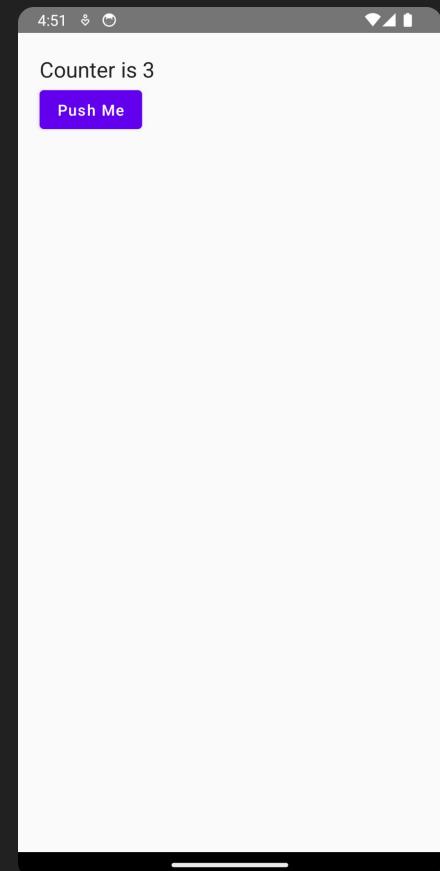
```
var counter by remember { mutableStateOf(0) }
```

mutableStateOf creates an observable.

```
var counter by remember { mutableStateOf(0) }
```

```
@Composable
fun CounterExample() {
    var counter by remember { mutableStateOf(0) }

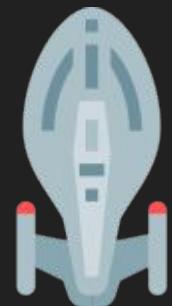
    Column(modifier = Modifier.padding(20.dp)) {
        Text(
            style = TextStyle(fontSize = 20.sp),
            text = "Counter is $counter"
        )
        Button(onClick = { counter++ }) {
            Text("Push Me")
        }
    }
}
```



Navigation is not supported
by Compose out-of-the box,
so we need to use a 3rd
party library.

Voyager - Multiplatform Navigation Library for Compose

- **Linear navigation**
- **Back navigation**
- BottomSheet navigation
- Tab navigation
- Multi-module navigation
- Nested navigation



Linear Navigation - Screens

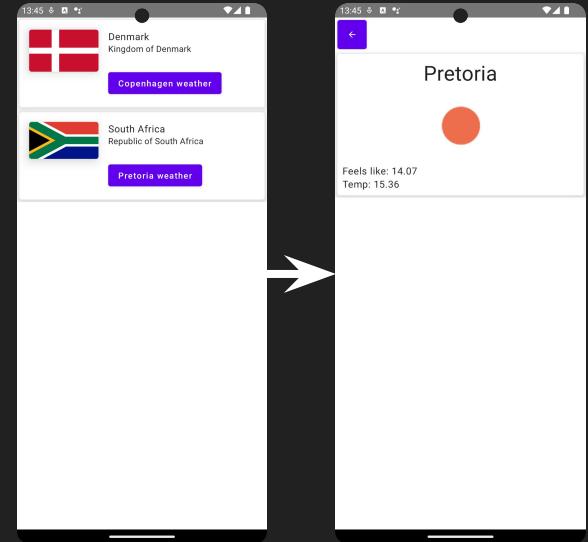
```
class HomeScreen : Screen {  
    @Composable  
    override fun Content() {  
        // Contents of Screen  
    }  
}
```

Linear Navigation - Setup

```
@Composable
fun App() {
    MaterialTheme {
        Navigator(HomeScreen())
    }
}
```

Linear Navigation - Navigation

```
val navigator = LocalNavigator.currentOrThrow  
navigator.push(OtherScreen())
```



Back Navigation - Poppable / Popping

Function	Description
navigator.canPop	If it's not the “root” or “home” screen
navigator.pop()	Move back to the previous screen
CurrentScreen()	Display the current screen contents

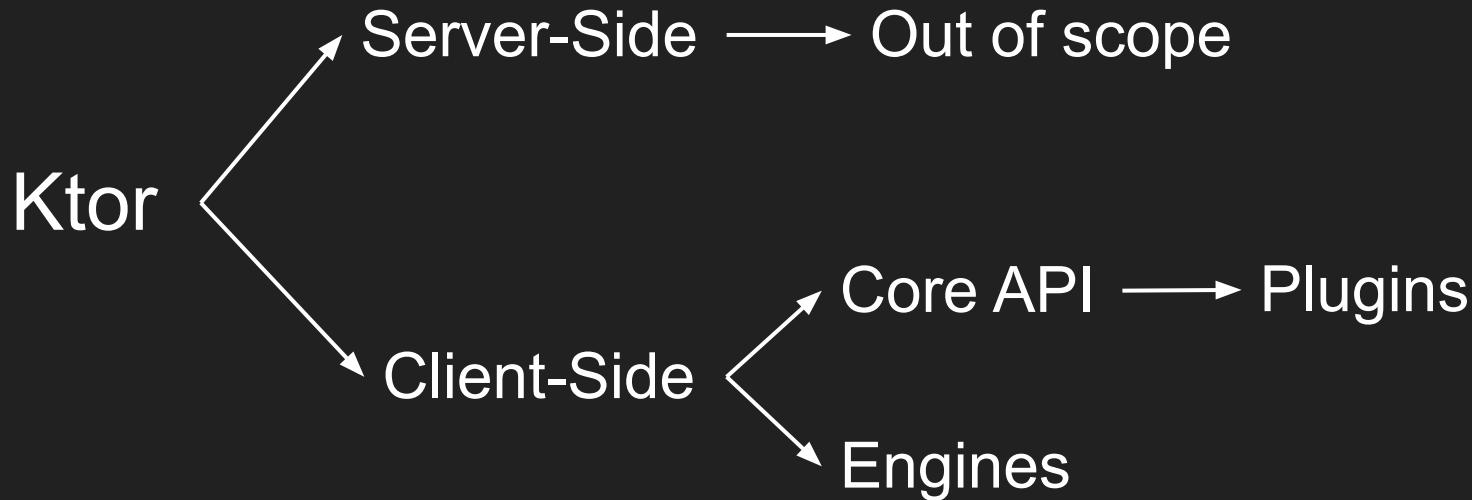
Summary

- We are coding with **Composable Functions / Composables**.
- **Modifiers** allow you to decorate or augment a Composable.
- **remember** stores values across recomposition.
- **mutableStateOf** creates a Compose observable.
- **Voyager** is a navigation library for Compose Multiplatform.

Ktor Client & Kotlinx.serialization

Introducing Ktor

- Ktor is a framework for managing services
- You can both create and consume services
- You create consumers via the Ktor Client
- Ktor Client has platform-specific Engines
- Both client and server depend on Plugins



Configuring Ktor Client

To configure Ktor Client we declare dependencies:

- On the core and logging, in the common source set
- On an engine, in each platform source set

Configuring Ktor Client - Common Source Set

```
commonMain.dependencies {  
    implementation(compose.runtime)  
    implementation(compose.foundation)  
    implementation(compose.material)  
    implementation(compose.ui)  
  
    implementation("io.ktor:ktor-client-core:2.3.7")  
    implementation("io.ktor:ktor-client-logging:2.3.7")  
}
```

Configuring Ktor Client - Desktop Source Set

```
desktopMain.dependencies {  
    implementation(compose.desktop.currentOs)  
  
    implementation("io.ktor:ktor-client-cio:2.3.7")  
}
```

Configuring Ktor Client - iOS Source Set

```
iosMain.dependencies {  
    implementation("io.ktor:ktor-client-darwin:2.3.7")  
}
```

Configuring Ktor Client - Android Source Set

```
androidMain.dependencies {  
    implementation(libs.compose.ui.tooling.preview)  
    implementation(libs.androidx.activity.compose)  
  
    implementation("io.ktor:ktor-client-android:2.3.7")  
}
```

Using the Ktor Client - Part 1

To use the Ktor Client in our code we:

1. Create an instance of the `HttpClient` type
2. Add required plugins via the `install` method
3. Further configure the plugins as required

Using the Ktor Client - Part 1

```
class ItemRepository {  
  
    private val client = HttpClient {  
        install(ContentNegotiation) {  
            json(Json {  
                prettyPrint = true  
                ignoreUnknownKeys = true  
            })  
        }  
    }  
  
    suspend fun allItems(): List<Item> { ... }  
    suspend fun singleItem(id: Long): Item? { ... }  
}
```

This plugin manages serialization, based on the HTTP Accept header. We configure it to format our JSON for viewing.

Using the Ktor Client - Part 2

For each call to the server we:

- Invoke request to acquire a Response
- Check the HTTP status code is 2xx
- Extract the body from the response

Using the Ktor Client - Part 2

```
suspend fun allItems(): List<Item> {
    val response = client.request("$ENDPOINT_URL/items")

    if (!response.status.isSuccess()) {
        //Return an empty list if the
        // response code is not 2xx
        return listOf()
    }
    return response.body()
}
```

Using the Ktor Client - Part 2

```
suspend fun singleItem(id: Long): Item? {
    val response = client.request("$ENDPOINT_URL/items/$id")

    if (!response.status.isSuccess()) {
        //Return null if the response
        // code is not 2xx
        return null
    }
    return response.body()
}
```

Introducing Serialization

The previous example will not work (yet!)

Because we have not considered serialization

Ktor does not convert objects to and from JSON

We need to add another library to accomplish this

Introducing Kotlinx.serialization

Platform specific serialization libraries exist

But we want to be multiplatform wherever possible

Hence we should use the kotlinx.serialization library:

<https://github.com/Kotlin/kotlinx.serialization>

Configuring Kotlinx.serialization

Kotlinx.serialization does not make use of reflection

Code to marshall objects is generated at build time

Hence we need to add two items into Gradle:

- The dependency on the library itself
- A compiler plugin to generate the code

Configuring Kotlinx.serialization - Common Source Set

```
commonMain.dependencies {  
    ...  
    implementation("io.ktor:ktor-client-serialization-kotlinx-json:2.3.7")  
}  
}
```

Configuring Kotlinx.serialization - Compiler Plugin

```
plugins {  
    ...  
    kotlin("plugin.serialization") version "1.9.22"  
}
```

Using kotlinx.serialization

Ktor will use the `kotlinx.serialization` library for you

You simply need to annotate your types with `@Serializable`

The qualified name is `kotlinx.serialization.Serializable`

Using kotlinx.serialization

It is possible to customize the serialization process in many ways

See the guide in the library documentation for details:

<https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md>

Practical 2: Connecting to a Web Service [20 mins]

Understanding Expect / Actual

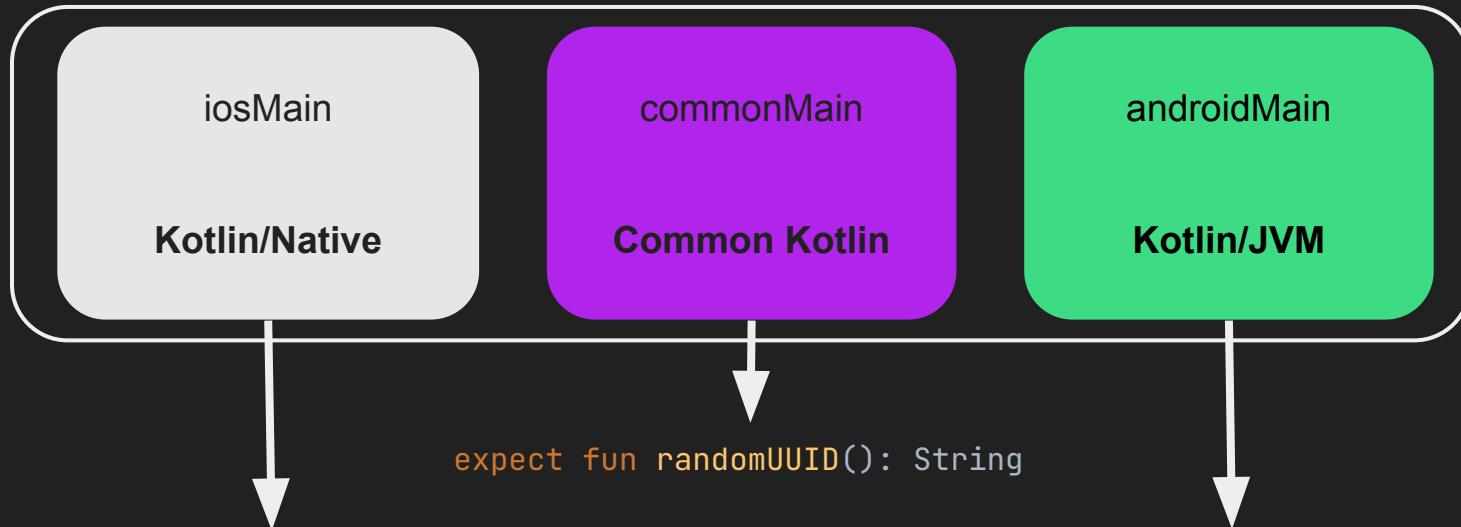
Demo: Wizard expect/actual Usage

What We Saw

- The Compose Multiplatform template uses `expect/actual` for `getPlatform()`
- The `expect` definition is in `commonMain` - we define what we're expecting
- The `actual` implementation is in `androidMain/desktopMain/iosMain` - we actually implement what we promised

It's important to have actual fun 😊

Expect/actual functions



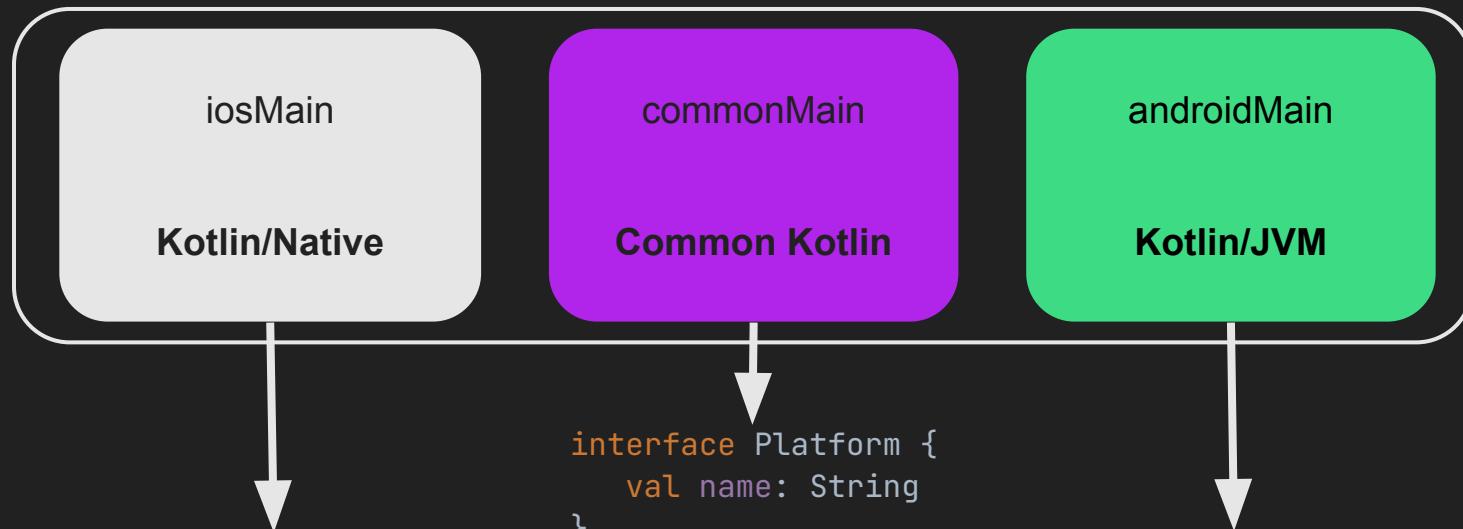
```
import platform.Foundation.NSUUID
actual fun randomUUID(): String =
    NSUUID().UUIDString()
```

```
import java.util.*
actual fun randomUUID() =
    UUID.randomUUID().toString()
```

Use expect/actual
functions for simple cases

Use interfaces for more
complex cases

interfaces

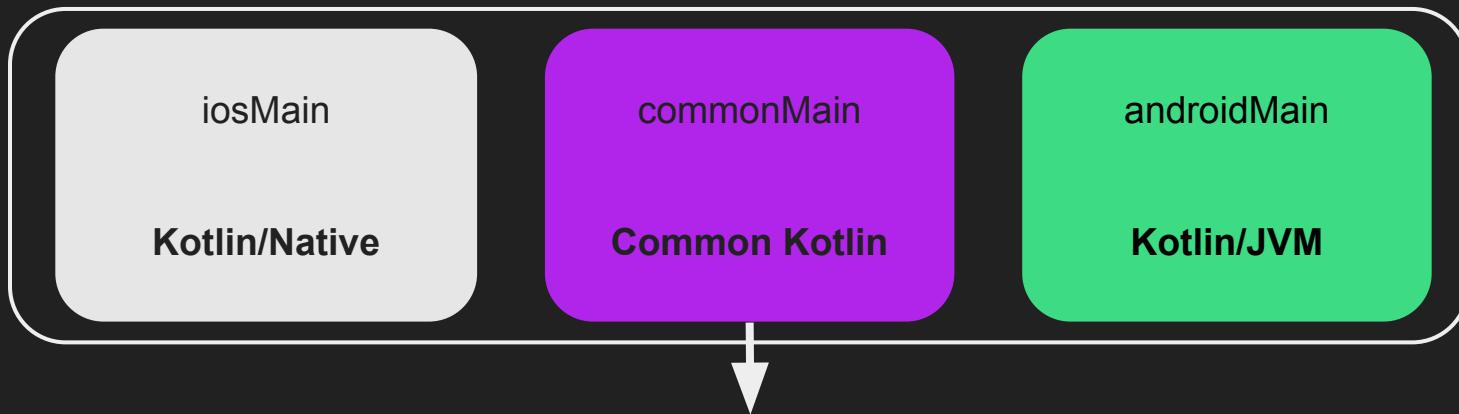


```
class iOSPlatform: Platform
```

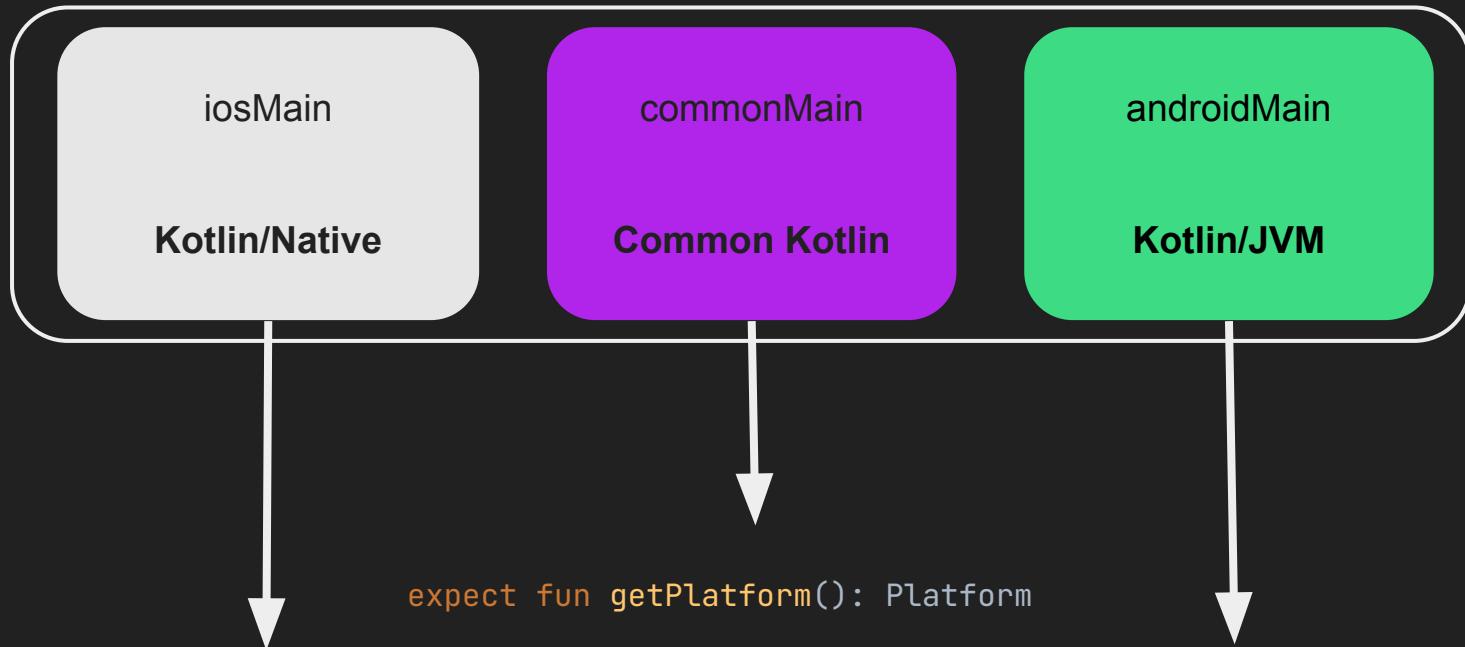
```
interface Platform {  
    val name: String  
}
```

```
class AndroidPlatform: Platform
```

Factory Functions



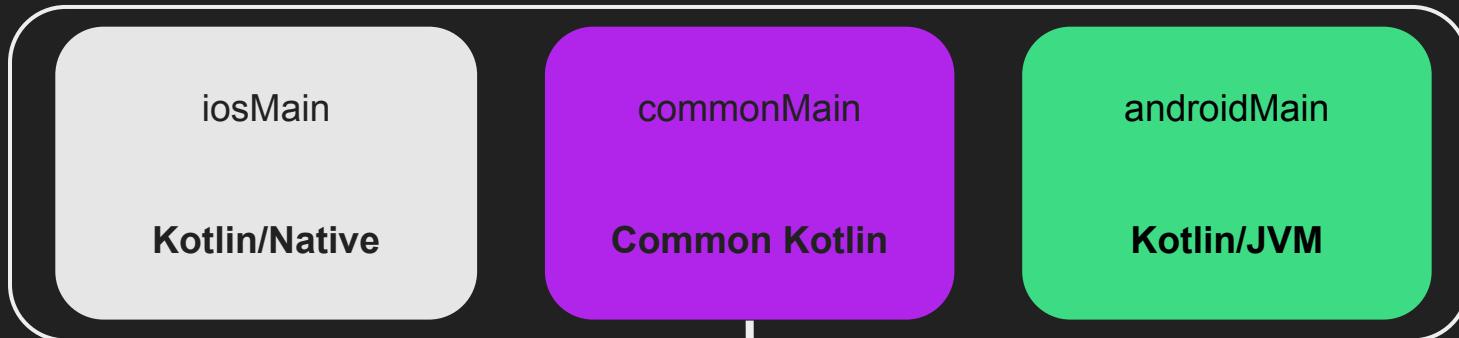
```
interface Platform {  
    val name: String  
}  
  
expect fun getPlatform(): Platform
```



```
class iOSPlatform: Platform  
...  
actual fun getPlatform() = iOSPlatform()
```

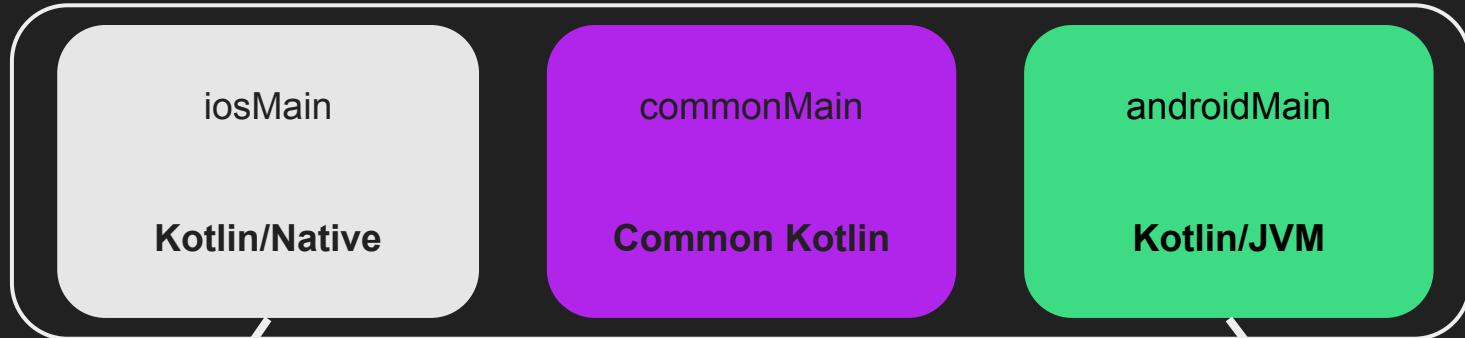
```
class AndroidPlatform: Platform  
...  
actual fun getPlatform() = AndroidPlatform()
```

Early Entry Points



```
interface Platform {  
    val name: String  
}
```

```
fun app(p: Platform) {  
    // app logic  
}
```

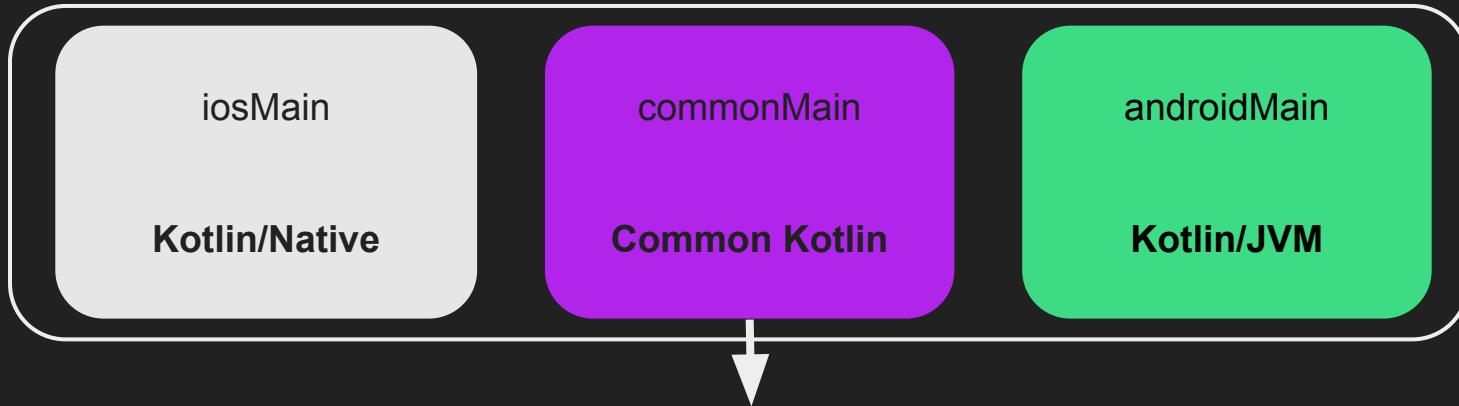


```
class iOSPlatform: Platform  
...  
  
@main  
struct iOSApp : App {  
    init() {  
        app(iOSPlatform())  
    }  
}
```

```
class AndroidPlatform: Platform  
...  
  
class MyApp : Application() {  
    override fun onCreate() {  
        super.onCreate()  
        app(AndroidPlatform())  
    }  
}
```

Use Dependency Injection
frameworks to provide
interface implementations

Dependency Injection with Koin



```
interface Platform {  
    val name: String  
}  
  
expect val platformModule: Module
```



```
class iOSPlatform: Platform  
...  
actual val platformModule: Module = module {  
    single<Platform> {  
        IOSPlatform()  
    }  
}
```

```
class AndroidPlatform: Platform  
...  
actual val platformModule: Module = module {  
    single<Platform> {  
        AndroidPlatform()  
    }  
}
```

Find out more about
Dependency Injection
later

Practical 3: Getting the Device Country Code [20 mins]

- Your expect/actuals must be in the same package
- Your Android actual goes in androidMain
- Your Desktop actual goes in desktopMain
- Your iOS actual goes in iosMain
- Use the gutter icons to navigate between the expect/actuals

Lunch

Post-Lunch Review

What Have We Achieved So Far?

- Described Kotlin Multiplatform and Compose Multiplatform as technologies in general
- Learnt how to create a project from scratch using the Compose Multiplatform wizard
- Created the user interface for the project using Compose Multiplatform
- Replaced the dummy data with real network calls
- Learnt how to call platform code from common code

Testing in Kotlin Multiplatform

Testing in Kotlin Multiplatform

Testing is a massive topic in it's own right

Here we are only concerned with unit and integration testing

- System and non-functional testing are out of scope
- By definition they test the application from the outside

Testing in Kotlin Multiplatform

In a KMP application we need to distinguish between:

- Writing tests for platform specific code
- Writing / running tests for common code

Tests for Platform Specific Code

These will use a platform specific testing tool

They may need framework specific extensions

- Such as the Ktor extensions to JUnit

We can include them in a multiplatform project

- But only within a platform-specific source set

Tests for Common Code

These are our main focus in this topic

The code being tested will execute on any platform

Hence we can use any supported testing framework

Tests for Common Code

We don't want to tightly couple our code to one testing framework

Hypothetically say we wrote all our tests for common code in JUnit

...then we could only run our unit tests on the JVM

Writing Tests for Common Code

Tests for common code use the [kotlin.test API](#)

This provides platform agnostic annotations and assertions

The tests you create can be run:

- With JUnit4, JUnit5 and TestNG on the JVM
- With a bespoke native test runner on iOS

kotlin.test

The `kotlin.test` library provides [annotations](#) to mark test functions and a set of [utility functions](#) for performing assertions in tests, independently of the test framework being used.

The test framework is abstracted through the [`Assertor`](#) class. A basic `Assertor` implementation is provided out of the box. Note that the class is not intended to be used directly from tests, use instead the top-level assertion functions which delegate to the `Assertor`.

The library consists of the modules:

- `kotlin-test-common` – assertions for use in common code;
- `kotlin-test-annotations-common` – test annotations for use in common code;
- `kotlin-test` – a JVM implementation of assertions from `kotlin-test-common`;
- `kotlin-test-junit` – provides an implementation of [`Assertor`](#) on top of JUnit and maps the test annotations from `kotlin-test-annotations-common` to JUnit test annotations;
- `kotlin-test-junit5` – provides an implementation of [`Assertor`](#) on top of JUnit 5 and maps the test annotations from `kotlin-test-annotations-common` to JUnit 5 test annotations;
- `kotlin-test-testng` – provides an implementation of [`Assertor`](#) on top of TestNG and maps the test annotations from `kotlin-test-annotations-common` to TestNG test annotations;
- `kotlin-test-js` – a JS implementation of common test assertions and annotations with the out-of-the-box support for [Jasmine](#), [Mocha](#), and [Jest](#) testing frameworks, and an experimental way to plug in a custom unit testing framework.

Demo of Testing Common Code

```
fun grep(  
    lines: List<String>,  
    pattern: String,  
    action: (String) → Unit  
) {  
    val regex = pattern.toRegex()  
    lines.filter(regex::containsMatchIn)  
        .forEach(action)  
}
```

Demo of Testing Common Code

```
import kotlin.test.Test
import kotlin.test.assertContains
import kotlin.test.assertEquals

class GrepTest {
    companion object {
        val sampleData = listOf(
            "123 abc",
            "abc 123",
            "123 ABC",
            "ABC 123"
        )
    }
}
```

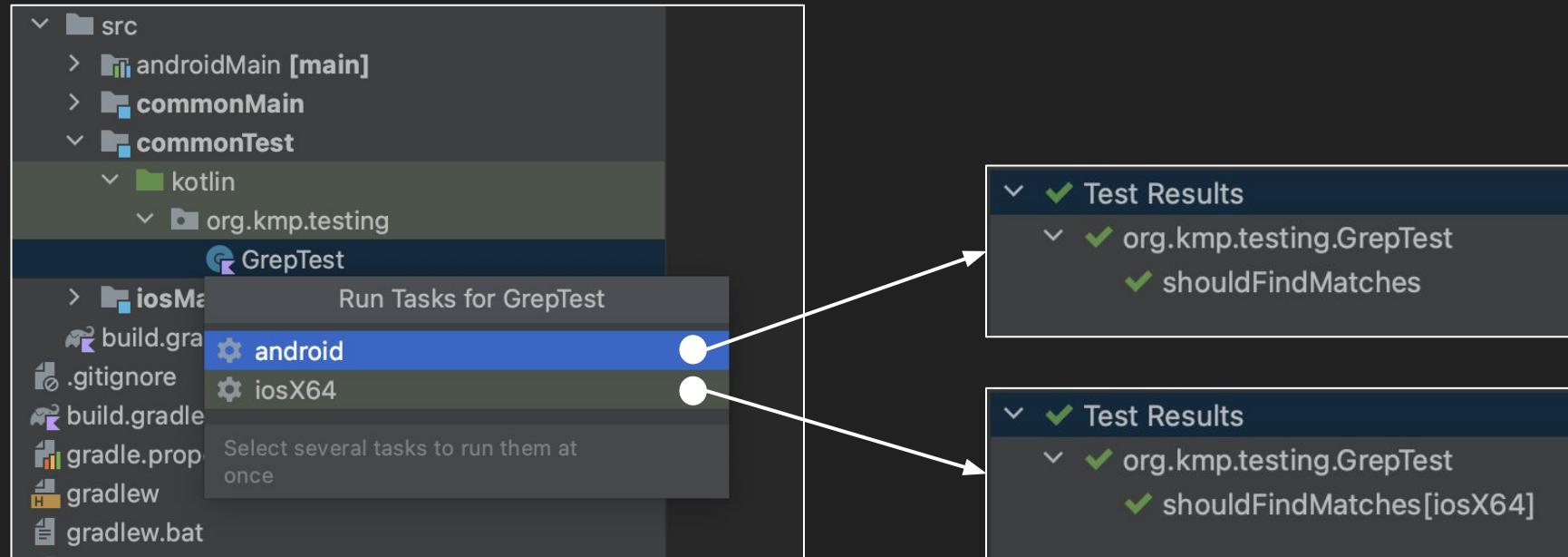


Demo of Testing Common Code

```
@Test
fun shouldFindMatches() {
    val results = mutableListOf<String>()
    grep(sampleData, "[a-z]+") {
        results.add(it)
    }

    assertEquals(2, results.size)
    for (result in results) {
        assertContains(result, "abc")
    }
}
```

Demo of Testing Common Code



Writing Tests for Common Code

Tests in common code cannot use platform specific types

- Using JUnit types will give compiler errors
- The [kotlin.test API](#) is all that you have
- It's not just a good idea - it's the (compiler) law 😊

The assertions are specified in the `Asserter` interface

- Within the API these are mapped to the tool in use

The Asserter Interface

```
interface Asserter {  
    fun fail(message: String?): Nothing  
    fun fail(message: String?, cause: Throwable?): Nothing  
    fun assertTrue(lazyMessage: () → String?, actual: Boolean)  
    fun assertTrue(message: String?, actual: Boolean)  
    fun assertEquals(message: String?, expected: Any?, actual: Any?)  
    fun assertNotEquals(message: String?, illegal: Any?, actual: Any?)  
    fun assertSame(message: String?, expected: Any?, actual: Any?)  
    fun assertNotSame(message: String?, illegal: Any?, actual: Any?)  
    fun assertNull(message: String?, actual: Any?)  
    fun assertNotNull(message: String?, actual: Any?)  
}
```

Testing and Expected / Actual Declarations

Common code can require platform specific inputs

- You could be testing business rules in common code
- ...but where data comes from a sensor on the device

Your tests can use expected and actual declarations to work

- The test methods will still be implemented in common code
- ...but the methods to build the data will be platform specific

```
expect fun buildSampleData(): List<String>

class GrepTest {
    companion object {
        val sampleData = buildSampleData()
    }

    @Test
    fun shouldFindMatches() {
        val results = mutableListOf<String>()
        grep(sampleData, "[a-z]{3}[0-9]{3}") {
            println("Matched '$it'")
            results.add(it)
        }

        assertEquals(2, results.size)
    }
}
```

In commonTest

In androidUnitTest

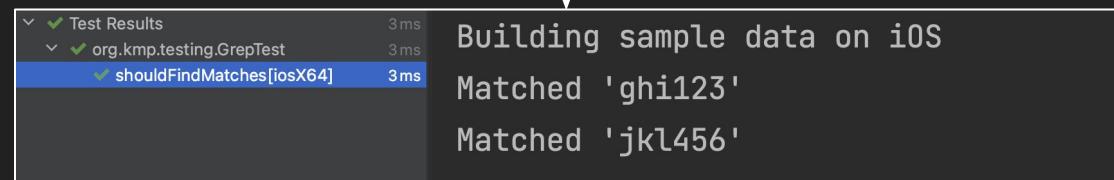
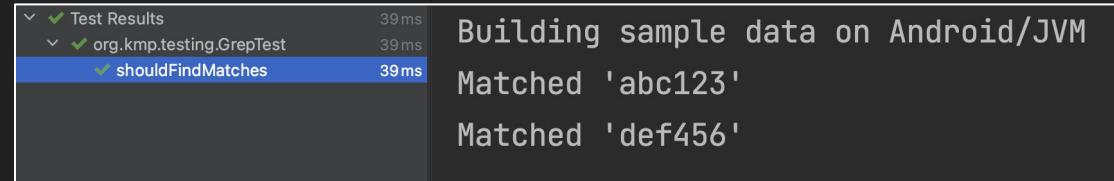
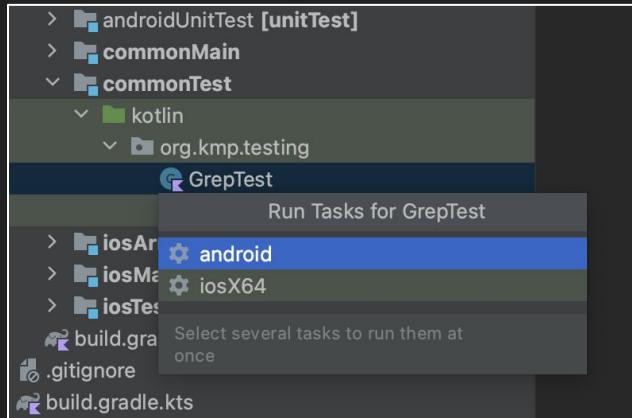
```
actual fun buildSampleData(): List<String> {
    println("Building sample data on Android/JVM")

    return listOf(
        "abc123",
        "123abc",
        "def456",
        "456def"
    )
}
```

In **iosTest**

```
actual fun buildSampleData(): List<String> {
    println("Building sample data on iOS")

    return listOf(
        "ghi123",
        "123ghi",
        "jkl456",
        "456jkl"
    )
}
```



Additional Options for Testing Common Code

The KMP Ecosystem is continually expanding

Existing libraries, like Kotest, now support KMP

With Kotest you can:

- Write tests in multiple styles (Cucumber, Jasmine, ScalaTest etc...)
- Execute the same test many times, but with different data sets
- Add delays, retries and timeouts for non-deterministic tests
- Perform Property Based Testing using generated values

Kotest is a flexible and elegant **multi-platform** test framework for **Kotlin** with extensive **assertions** and integrated **property testing**

[Get Started](#)

Star

4,053

KOTLINLANG KOTEST RELEASE V5.7.2 LATEST SNAPSHOT V5.6.0.1131-SNAPSHOT LICENSE APACHE2.0 STACKOVERFLOW KOTEST



Test Framework

The Kotest test framework enables test to be laid out in a fluid way and execute them on JVM, Javascript, or native platforms.

With built in coroutine support at every level, the ability to use functions as test lifecycle callbacks, extensive extension points, advanced conditional evaluation, powerful data driven testing, and more.



Assertions Library

The Kotest assertions library is a Kotlin-first multi-platform assertions library with over 300 rich assertions.

It comes equipped with collection inspectors, non-deterministic test helpers, soft assertions, modules for arrow, json, kotlinx-datetime and much more.

[Read more](#)

Property Testing

The Kotest property testing module is an advanced multi-platform property test library with over 50 built in generators.

It supports failure shrinking, the ability to easily create and compose new generators; both exhaustive and arbitrary checks, repeatable random seeds, coverage metrics, and more.

Practical 4: Creating Tests for Common Kotlin

Persistence

Persistence

- SQLDelight
- Realm
- KStore

SQLDelight

- Generates type-safe Kotlin APIs from SQL
- Compile-time verification
- IDE plugin
- Multiplatform scenario uses SQLite
- Platform-specific drivers

<https://cashapp.github.io/sqldelight/>

Database and queries - .sq file

```
CREATE TABLE Launch (
    flightNumber INTEGER NOT NULL,
    missionName TEXT NOT NULL,
    ...
);
```

```
selectAllLaunches:
SELECT Launch.*  
FROM Launch;  
...
```

Database and queries - .sq file

```
CREATE TABLE Launch (
    flightNumber INTEGER NOT NULL,
    missionName TEXT NOT NULL,
    ...
);
```

```
selectAllLaunches:
SELECT Launch.*  
FROM Launch;  
...
```

Database Helper

```
class Database(driver: SqlDriver) {  
    private val database = AppDatabase(driver)  
    private val dbQuery = database.appDatabaseQueries  
    fun getAllLaunches(): List<RocketLaunch> {  
        return dbQuery  
            .selectAllLaunches(::mapLaunchSelecting)  
            .executeAsList()  
    }  
    ...  
}
```

Why not use **SQLDelight** for
caching?

Need to create a table for
each non-primitive property

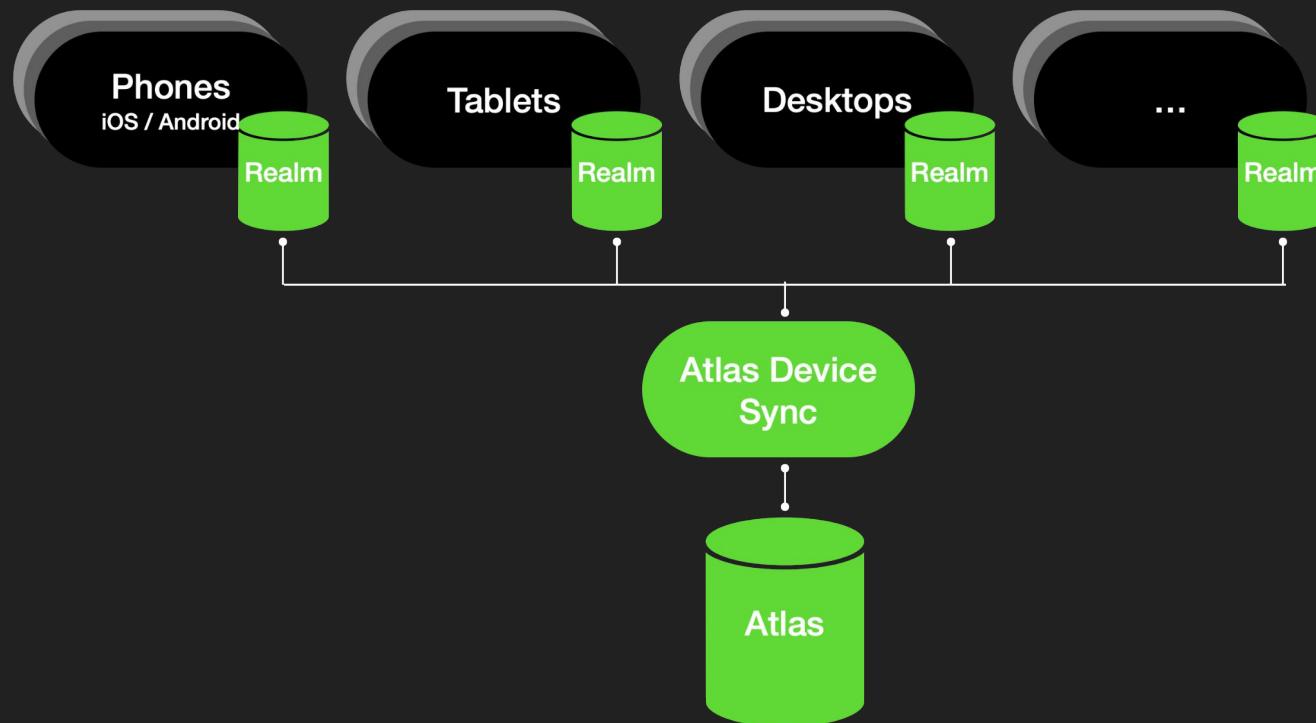


Realm

- Object-oriented database
- No mappings
- Simpler client code
- Faster complex queries
- Concurrency with coroutines and flows
- Encryption with key

<https://realm.io/realm-kotlin/>

Realm



Realm + Device Sync

- Offline-first approach
- Device Sync handles connectivity issues, conflict resolutions
- Small code changes needed for local to cloud

Why not use **Realm** for
caching?

Need to create a whole
parallel class hierarchy



KStore

- Stores objects in files
- Needs platform-specific path
- Uses coroutines, kotlinx-serialization, okio
- Read-write locks
- In-memory caching

<https://github.com/xxfast/KStore>

Perfect for caching 😊

Dependencies

```
kotlin {  
    sourceSets {  
        commonMain.dependencies {  
            implementation("io.github.xxfast:kstore:0.6.0")  
            implementation("io.github.xxfast:kstore-file:0.6.0")  
        }  
    }  
}
```

File Paths

- Android: "\${context.filesDir.path}/country_cache.json"
- iOS: "\${NSHomeDirectory()}/country_cache.json"
- Desktop:

```
val appDir = AppDirsFactory
    .getInstance()
    .getUserDataDir("weatherapp", "1.0.0", "workshop")
"$appDir/country_cache.json"
```

Create Cache

```
import io.github.xxfast.kstore.KStore
import io.github.xxfast.kstore.file.storeOf
...
private val cache: KStore<List<Country>> = storeOf(filePath =
pathToCountryCache())
```

Caching Behavior

1. Get everything in the cache
2. If cache contents is null:
 - a. Get the data from the webservice (API)
 - b. Fill the cache with the new data
3. Return the data

Practical 5: Client-Side Data Caching

[20 mins]

Much faster loading



Coroutines and Flows

Why Do We Need Coroutines?

- Concurrency design pattern that you can use to **simplify** code that executes **asynchronously**
- Help to manage **long-running tasks** that might otherwise **block the calling thread**
- For apps the calling thread is the **main thread**. Blocking the main thread can cause your app to become **unresponsive** (ANR)

Suspending Functions

- Function that can **suspend** its computation at some point and **resume** later on, without blocking the thread / freeing the thread to do other work
- Marked with **suspend** keyword in function signature
- Can only be called from **coroutine builders** or **other suspend functions**

Consuming Suspending Functions

```
var weather: Weather? by remember { mutableStateOf(null) }
```

```
LaunchedEffect(Unit) {  
    weather = WeatherApi().getWeather(lat, long)  
}
```



LaunchedEffect: run
suspend functions in the
scope of a composable

What is a Flow?

- Type that can **emit multiple values** sequentially, as opposed to suspend functions that return only a **single value**
- Example: changes in prices to the stock market
- Created using a **flow builder** `flow { ... }`
- You can `emit(...)` values inside a flow builder
- You can call **suspending functions** inside a flow builder

StateFlow

- Optimally emit state updates and emit values to multiple consumers
- StateFlow is a state-holder observable flow that emits the current and new state updates to its collectors
- The current state value can also be read through its **value** property
- To update state and send it to the flow, assign a new **value** to the **value** property of the **MutableStateFlow** class.

Creating a Flow

```
flow {
    // Call suspending functions and use their values
    WeatherApi().getWeather(0.0,0.0)

    // Emit a value
    emit(...)

    // Wait an approximate time
    delay(1.seconds)
}
```

Consuming a StateFlow in Compose Multiplatform

Remember across recomposition.
May not be necessary if the flow is saved in another way (like VM).

Collects values of Flow and transforms into Compose State

```
val myValue = remember { someFlow }.collectAsState()
```

```
Text(text = "My value: ${myValue.value}")
```

kotlinx-datetime

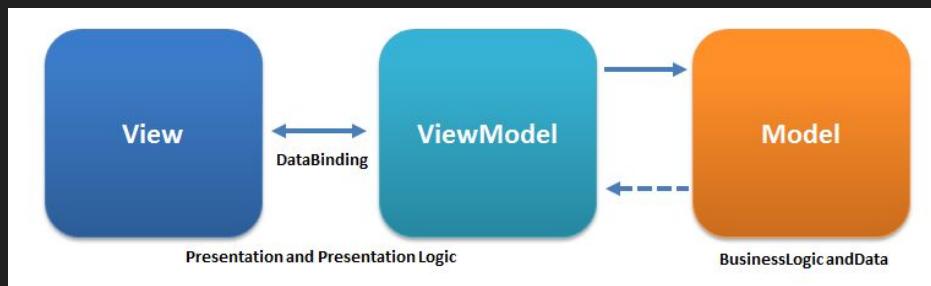
- Instant is a moment in time, independent of timezone
- LocalDateTime is a moment in time, related to timezone
- Clock.System.now() queries the OS for current time Instant
- Convert an Instant to your LocalDateTime with
`time.toLocalDateTime(TimeZone.currentSystemDefault())`
- Query a LocalDateTime with hour, minute, second etc.

Practical 6: Consuming Flows

Sharing View Models

ViewModel

- Integral part of the MVVM (Model-View-ViewModel) pattern
- Exposes state to the UI, encapsulates business logic
- On Android, ViewModel survives configuration changes



Source: Wikipedia

Fixes a code smell



Code Smell

```
class TimeApi {  
    private val clockTime = flow {  
        while(true) {  
            emit(formatTime(Clock.System.now()))  
            delay(1.seconds)  
        }  
    }  
  
    val actualTime = clockTime.stateIn(GlobalScope, SharingStarted.Eagerly, "N/A")  
    ...  
}
```

Delicate coroutines API.

- Lifetime is responsibility of programmer
- Easy to create memory leaks

Importing the Dependencies

```
commonMain.dependencies {  
    ...  
    implementation("cafe.adriel.voyager:voyager-navigator:1.0.0")  
    implementation("cafe.adriel.voyager:voyager-screenmodel:1.0.0")  
}
```

The ViewModel

```
class HomeScreenModel : ScreenModel {  
    val actualTime = TimeApi().clockTime.stateIn(screenModelScope, SharingStarted.Eagerly, "N/A")  
}
```

Creating the ViewModel

```
class HomeScreen : Screen {  
    @Composable  
    override fun Content() {  
        val screenModel = rememberScreenModel { HomeScreenModel() }  
  
        Surface {  
            ...  
  
            val time = screenModel.actualTime.collectAsState()  
  
            Column {  
                Text(  
                    modifier = Modifier.align(Alignment.CenterHorizontally).padding(8.dp),  
                    text = "Local time: ${time.value}",  
                    style = MaterialTheme.typography.h4  
                )  
  
            }  
        }  
    }  
}
```

Using the ViewModel

```
class HomeScreen : Screen {  
    @Composable  
    override fun Content() {  
        val screenModel = rememberScreenModel { HomeScreenModel() }  
  
        Surface {  
            ...  
  
            val time = screenModel.actualTime.collectAsState()  
  
            Column {  
                Text(  
                    modifier = Modifier.align(Alignment.CenterHorizontally).padding(8.dp),  
                    text = "Local time: ${time.value}",  
                    style = MaterialTheme.typography.h4  
                )  
                ...  
            }  
        }  
    }  
}
```

Flow is saved by VM, no need to remember it.

Dependency Injection with Koin

Apply the OO principle of Dependency Injection:

For each dependency we create an appropriate interface

These interfaces are then used in the constructors of the two View Models

Implementations of the interfaces are injected from outside

The View Models don't know which versions they are using

Why Dependency Injection?

Decouple the interface from the implementation

Easily substitute the implementations

Such as when you want to put in a test implementation

Summary of steps:

1. Import the dependencies in the `build.gradle.kts` file
2. Make the dependencies implement interfaces
3. Make constructors for the View Models
4. Create the module
5. Use `startKoin` for Android/iOS/Desktop

Import the dependencies

```
androidMain.dependencies {  
    ...  
    implementation("io.insert-koin:koin-android:3.2.0")  
}  
commonMain.dependencies {  
    ...  
    implementation("cafe.adriel.voyager:voyager-koin:1.0.0")  
    implementation("io.insert-koin:koin-core:3.2.0")  
}
```

Implement interfaces

```
interface CountryApi {  
    suspend fun getAllCountries(): List<Country>  
}  
  
class CountryApiImpl : CountryApi{  
    ...  
}  
  
// And so on
```

Create constructors for View Models

Create module

```
val appModule = module {
    single<CountryApi>{CountryApiImpl()}
    single<WeatherApi>{WeatherApiImpl()}
    single<TimeApi>{ TimeApiImpl()}
    single<CountrySDK>{ CountrySDKImpl(get()) }
    factory<HomeScreenModel> { HomeScreenModel(get(), get()) }
    factory<WeatherScreenModel> { WeatherScreenModel(get()) }
}
```

Call startKoin for Android

```
class WeatherApplication : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
  
        startKoin() {  
            modules(appModule)  
        }  
    }  
  
}
```

Call startKoin for iOS: in iosMain define helper function

```
fun initKoin(){  
    startKoin {  
        modules(appModule)  
    }  
}
```

Call helper function from iOS app init

```
@main
struct iOSApp: App {

    init() {
        MainViewControllerKt.doInitKoin()
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Recap of steps:

1. Import the dependencies in the `build.gradle.kts` file
2. Make the dependencies implement interfaces
3. Make constructors for the View Models
4. Create the module
5. Use `startKoin` for Android/iOS/Desktop

Discussion Time

What is your impressions
of the technology now that
you've given it a try?



What are your next steps ?

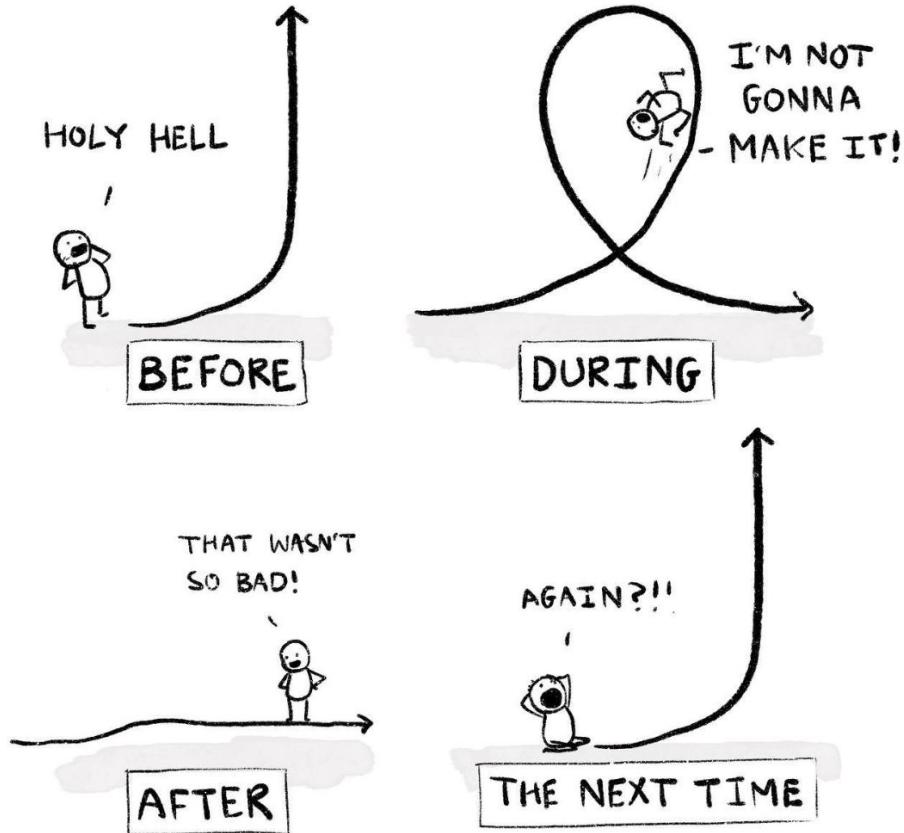


How would you convince
your team  to also try it?

What would you say to your
boss  to convince them?

Closing

HOW LEARNING CURVES FEEL...



How do you feel about
today's workshop?

What have you learnt?

What will you do next?

Group picture



(Let me know if you don't want to be photographed)

Thank you



Stay in touch:

𝕏 - @pamelaahill

✉️ - pamela.hill@jetbrains.com

💻 - pamelaahill.com