Hi! 👋

WiFi:

Pull repo: github.com/pahill/gdg-cpt-2024-practical-intro-kmp

`docs` directory:

- Installation instructions
- Practicals instructions
- Slides

A Practical Introduction to Developing for Android and iOS with Kotlin Multiplatform

GDG Cape Town Google I/O Extended 2024

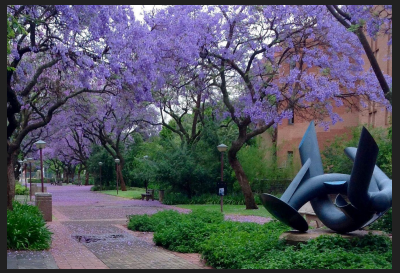
JETBRAINS

About Your Instructor - Pamela

- Kotlin Developer Advocate at JetBrains



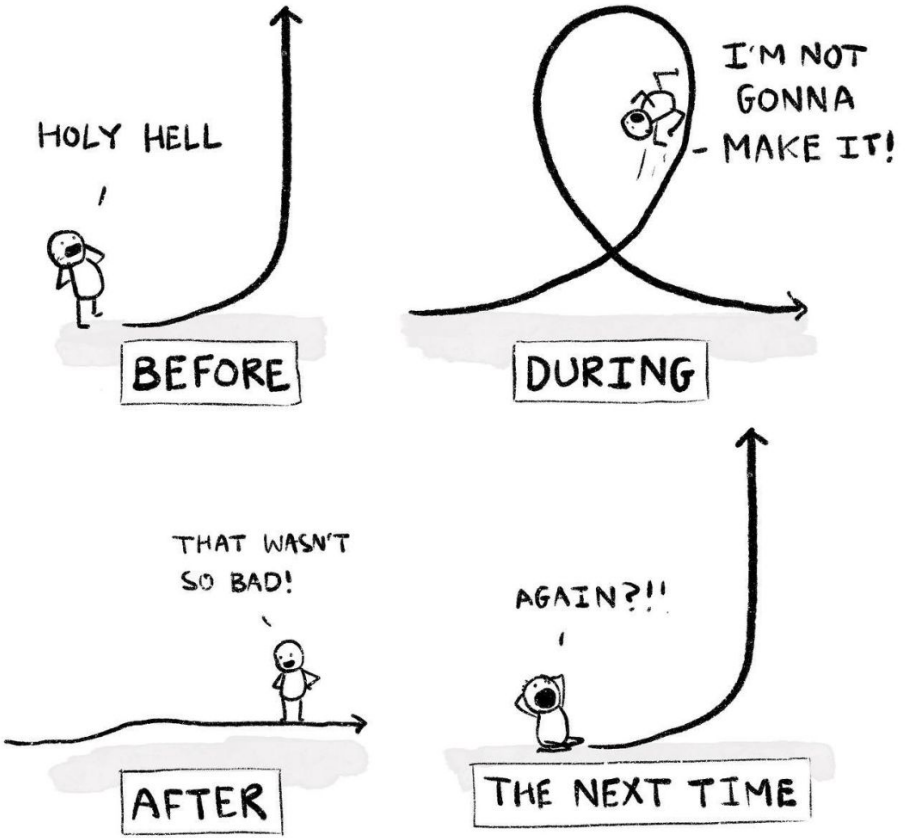
- From Pretoria




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

HOW LEARNING CURVES FEEL...



@STEINBERGDRAWSCARTOONS

What is Kotlin Multiplatform?

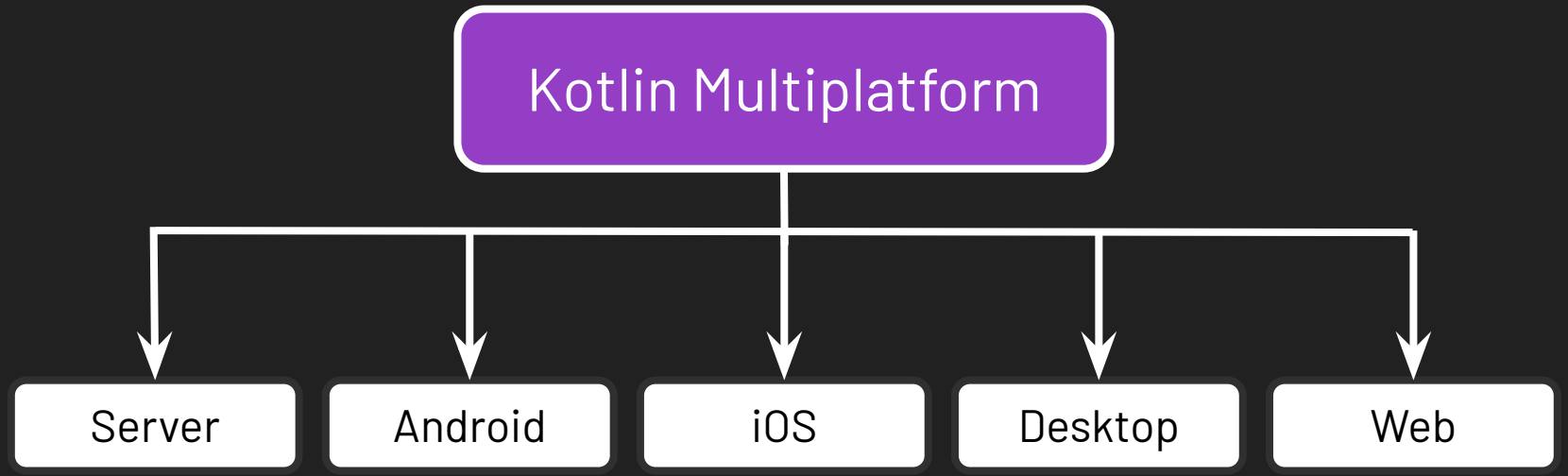
What is Kotlin Multiplatform?

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

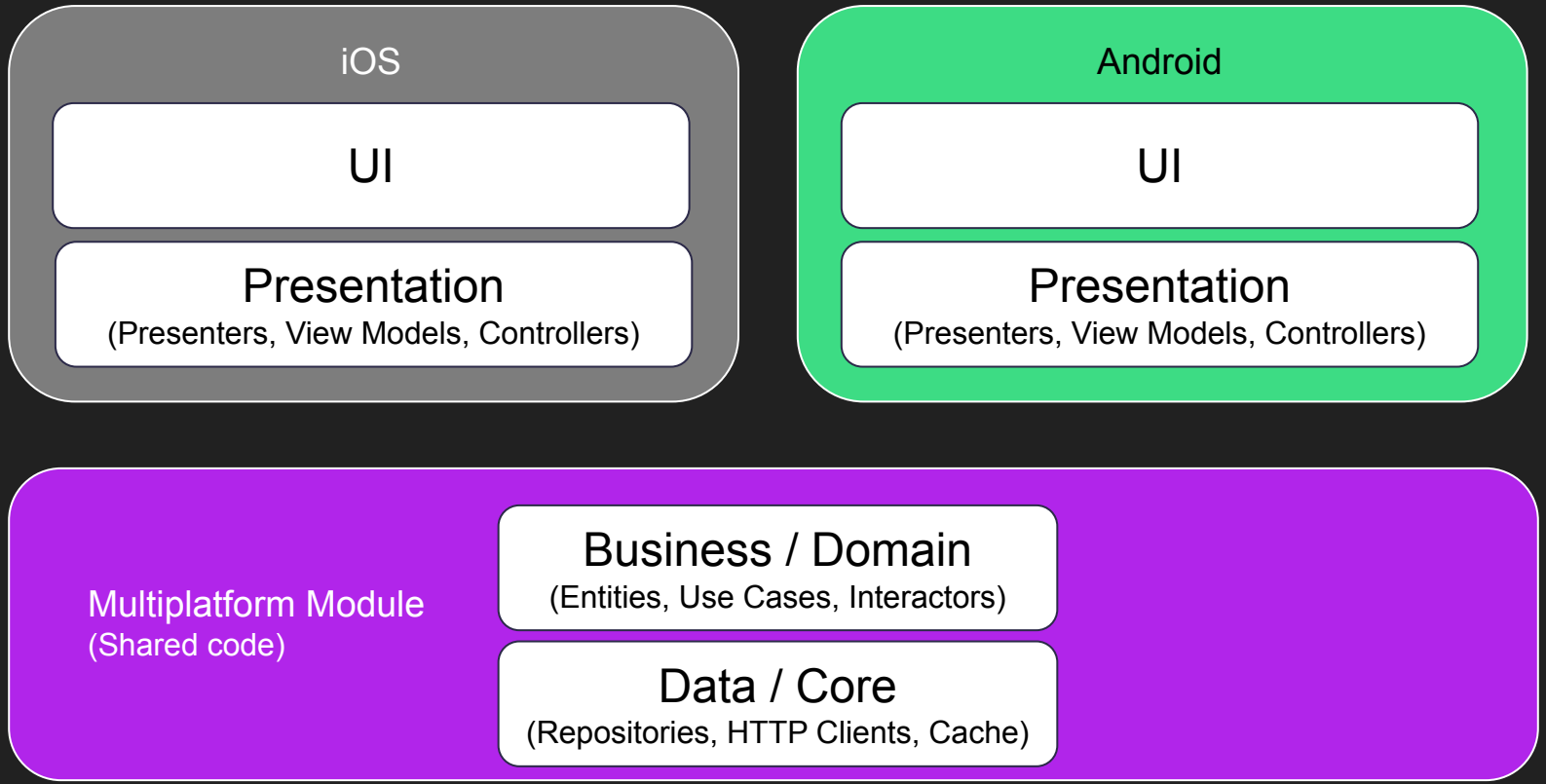
What is Kotlin Multiplatform?

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

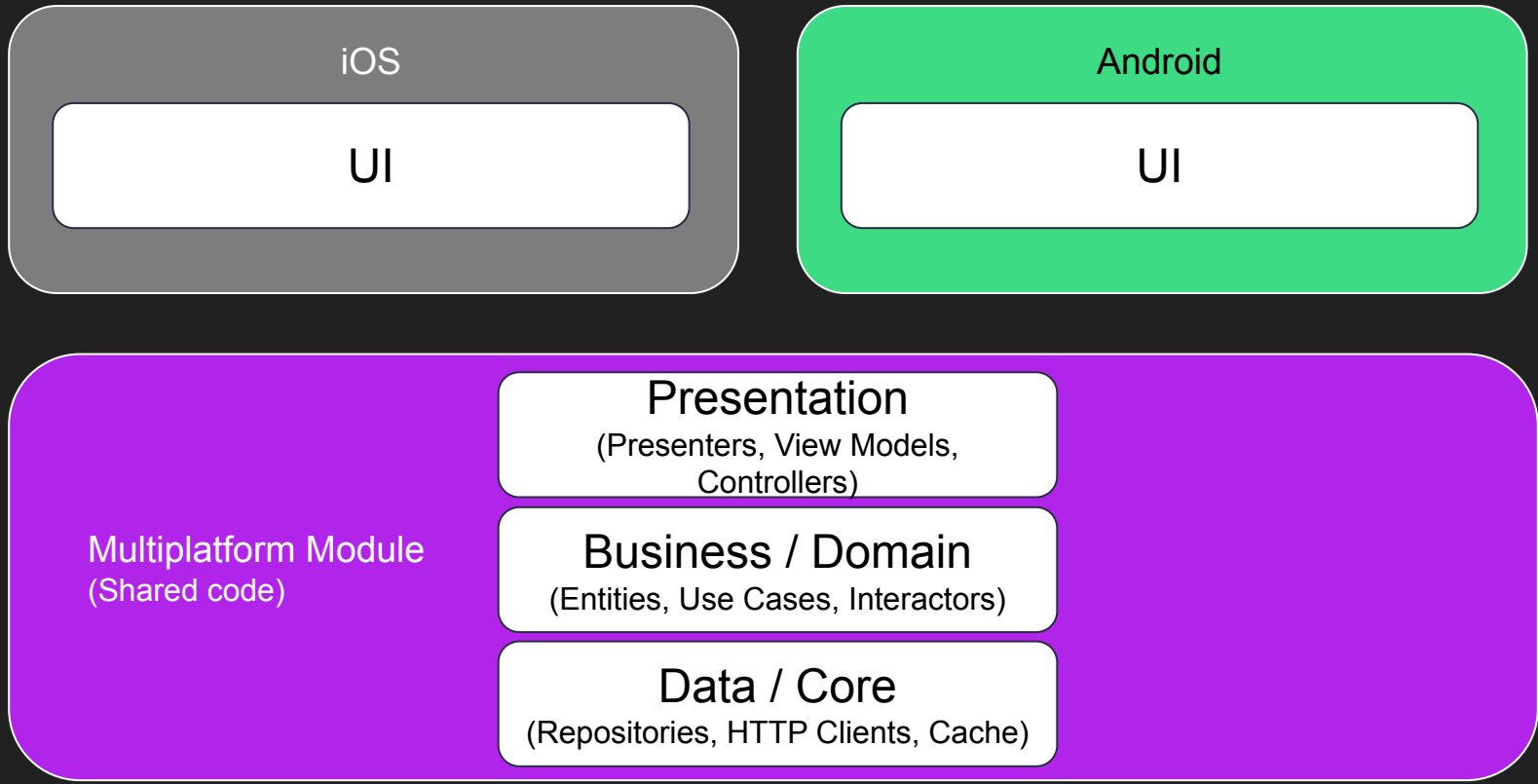
What kind of platforms?



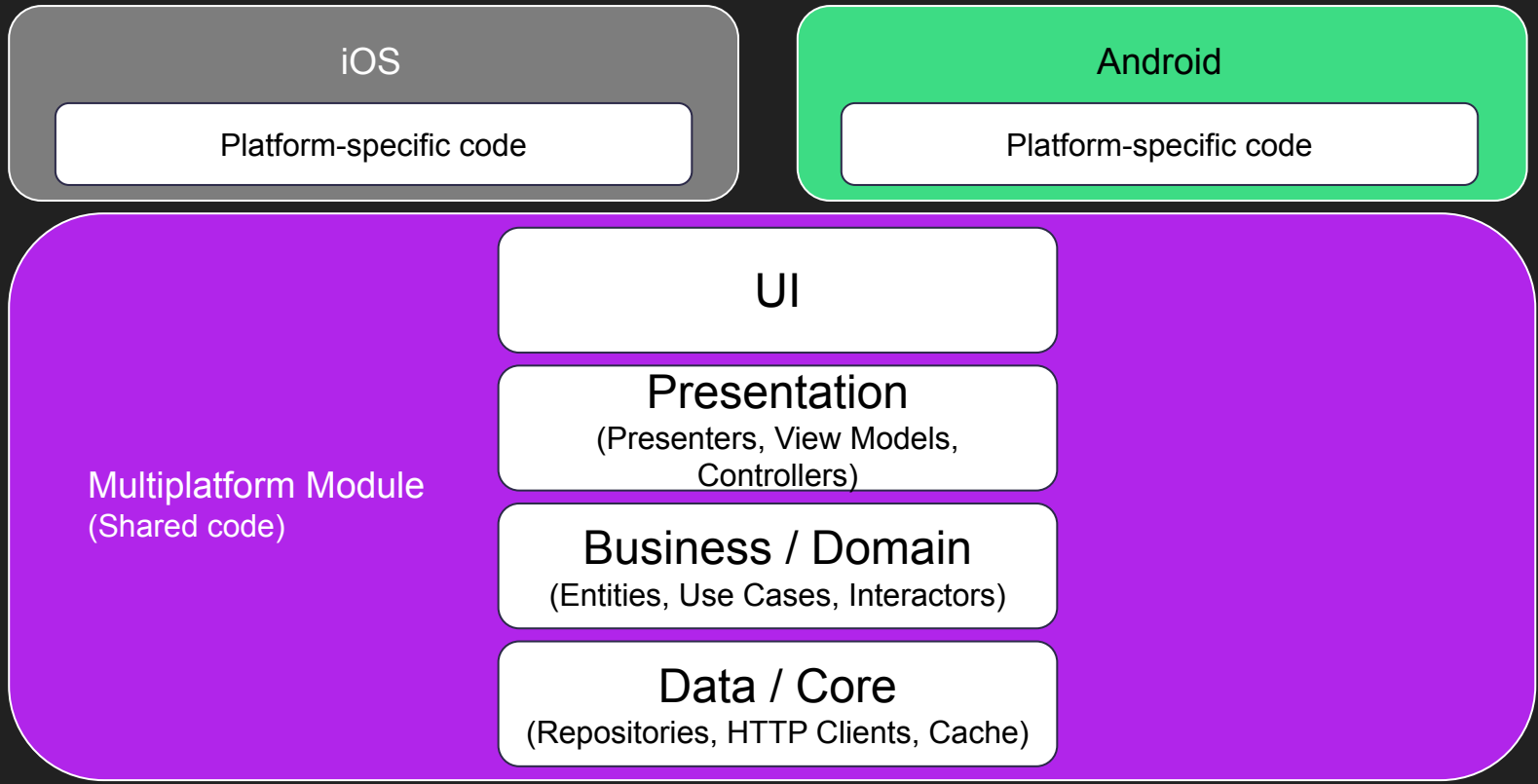
What kind of common code?



What kind of common code?



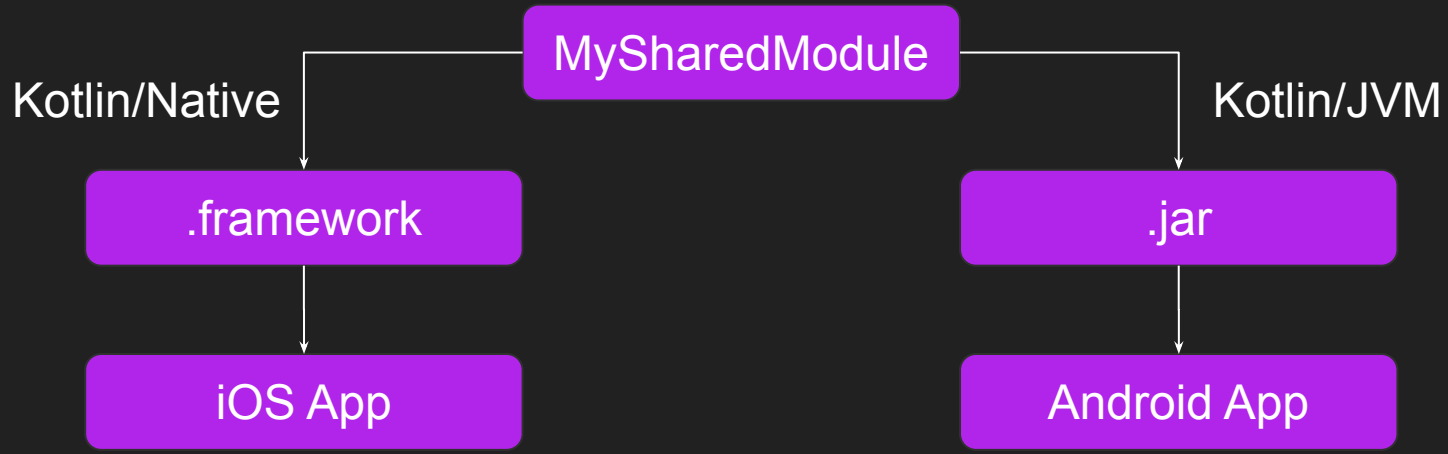
What about ✨*Compose Multiplatform*✨?



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**

Forbes

PHILIPS

vmware®

AUTODESK

https://www.jetbrains.com/help/kotlin-multiplatform-dev/case-studies.html

Demo: Using Project Wizards

Introducing Compose Multiplatform

The TL;DR on Compose

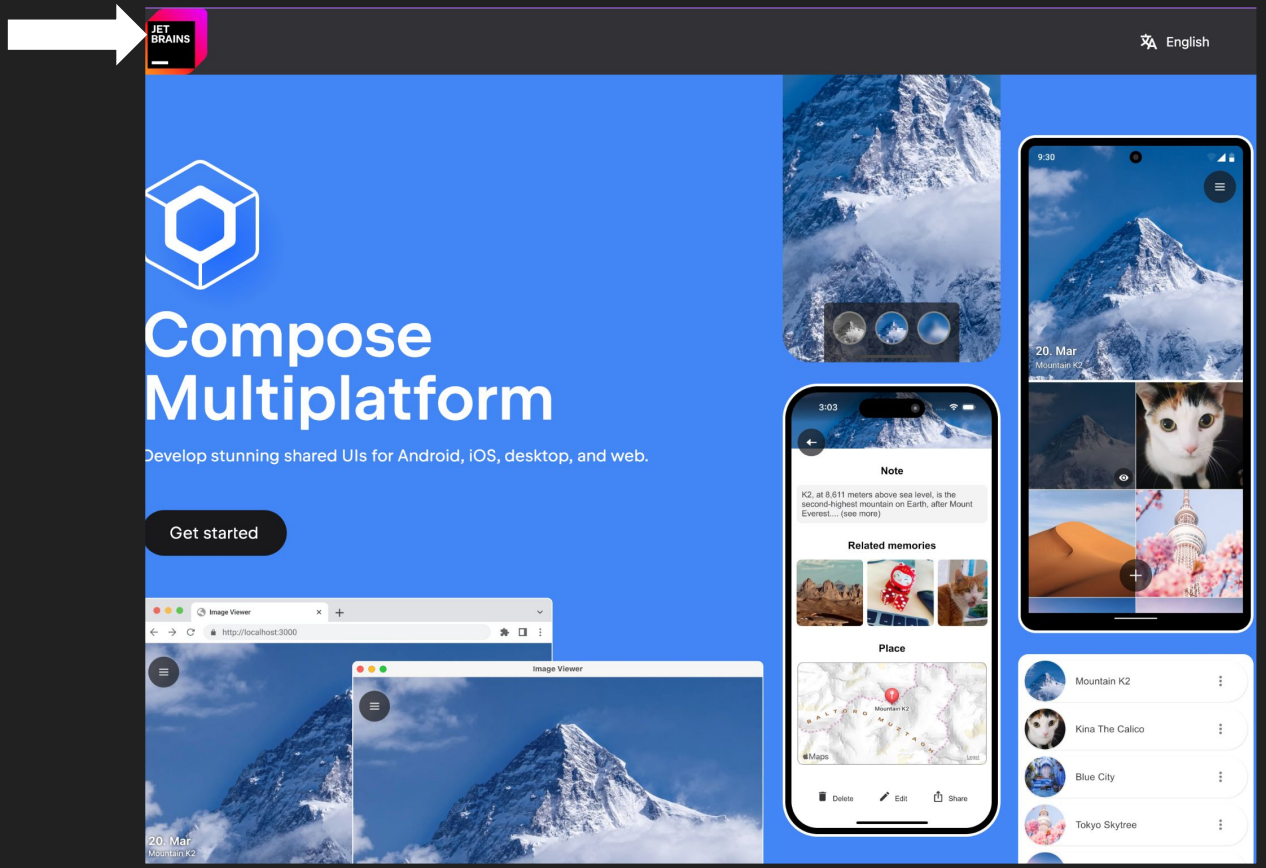
- Compose lets you create awesome UIs
- Compose Multiplatform is produced by **JetBrains**
- Built on Kotlin Multiplatform by **JetBrains**
- Powered by Jetpack Compose from **Google**

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.

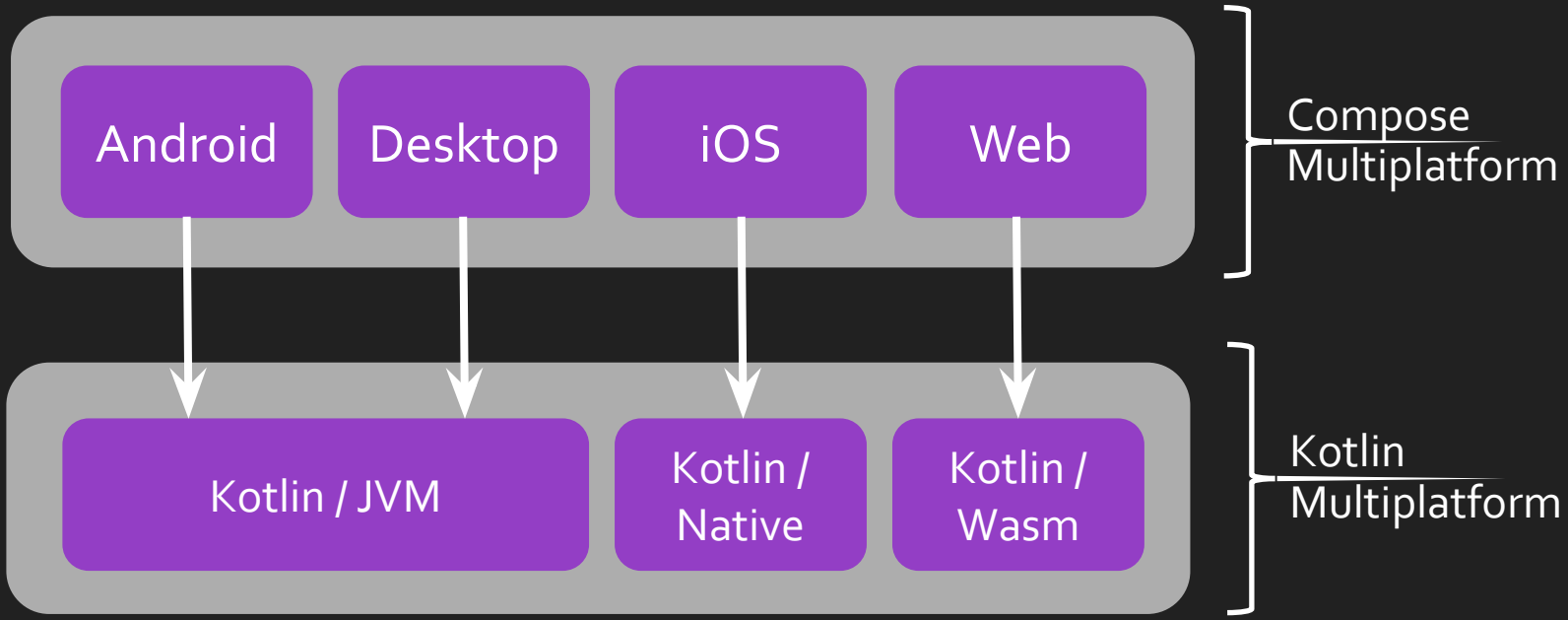
```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
 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

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



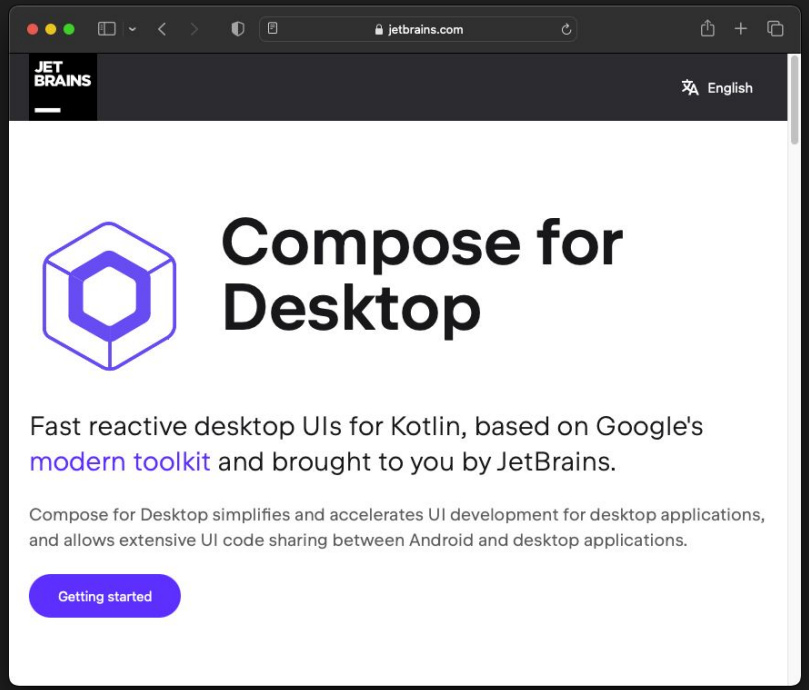
https://jb.gg/compose

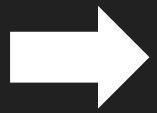
Compose & Kotlin Multiplatform

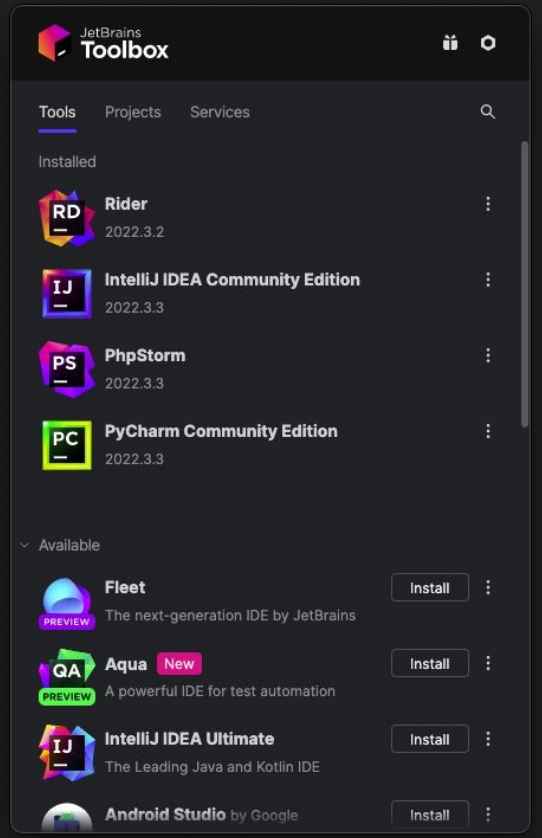


What is WebAssembly / Wasm?

- A binary instruction format for portable virtual machines
- Smaller and faster than JavaScript, but fully interoperable
- Write code in languages like Kotlin / Go / Rust, compile to Wasm
- Already supported in most modern browsers, Node.js

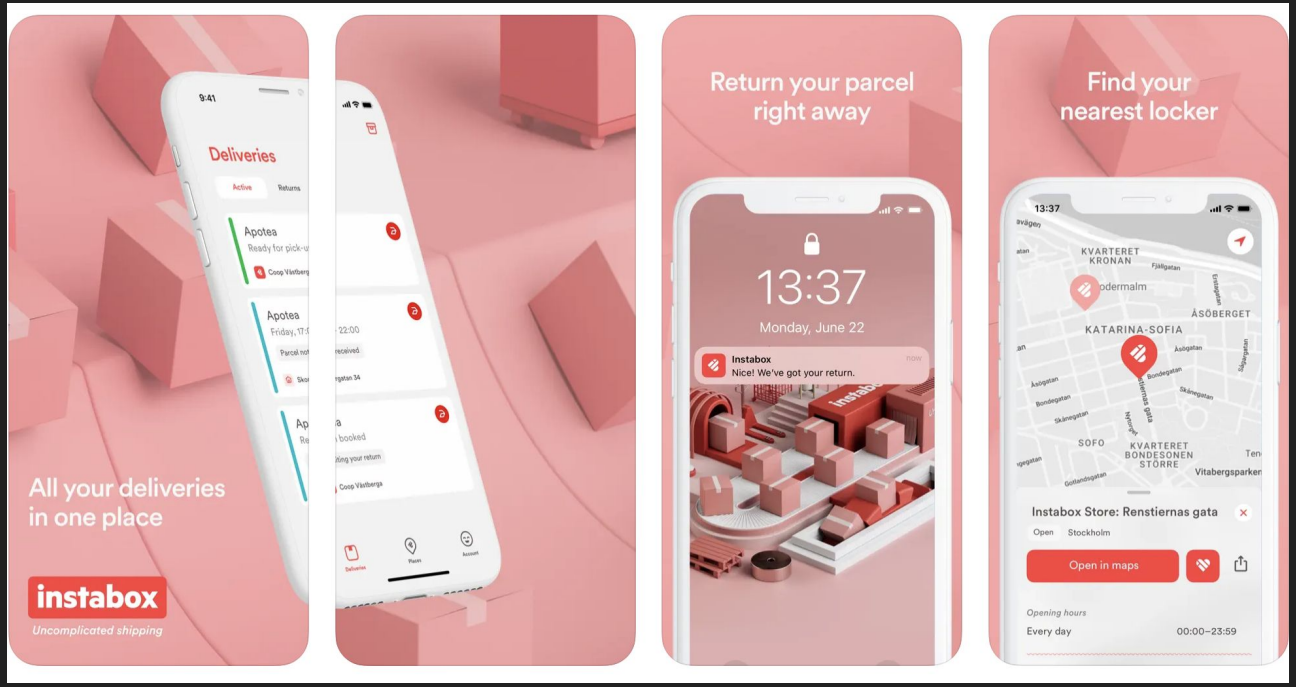

jetbrains.com
JET BRAINS
English
Compose for Desktop
Fast reactive desktop UIs for Kotlin, based on Google's modern toolkit and brought to you by JetBrains.
Compose for Desktop simplifies and accelerates UI development for desktop applications, and allows extensive UI code sharing between Android and desktop applications.
Getting started




JetBrains Toolbox
Tools
Projects
Services
Installed
Rider
2022.3.2
IntelliJ IDEA Community Edition
2022.3.3
PhpStorm
2022.3.3
PyCharm Community Edition
2022.3.3
Available
Fleet
The next-generation IDE by JetBrains
Install
Aqua
New
A powerful IDE for test automation
Install
IntelliJ IDEA Ultimate
The Leading Java and Kotlin IDE
Install
Android Studio by Google
Install

Instabee

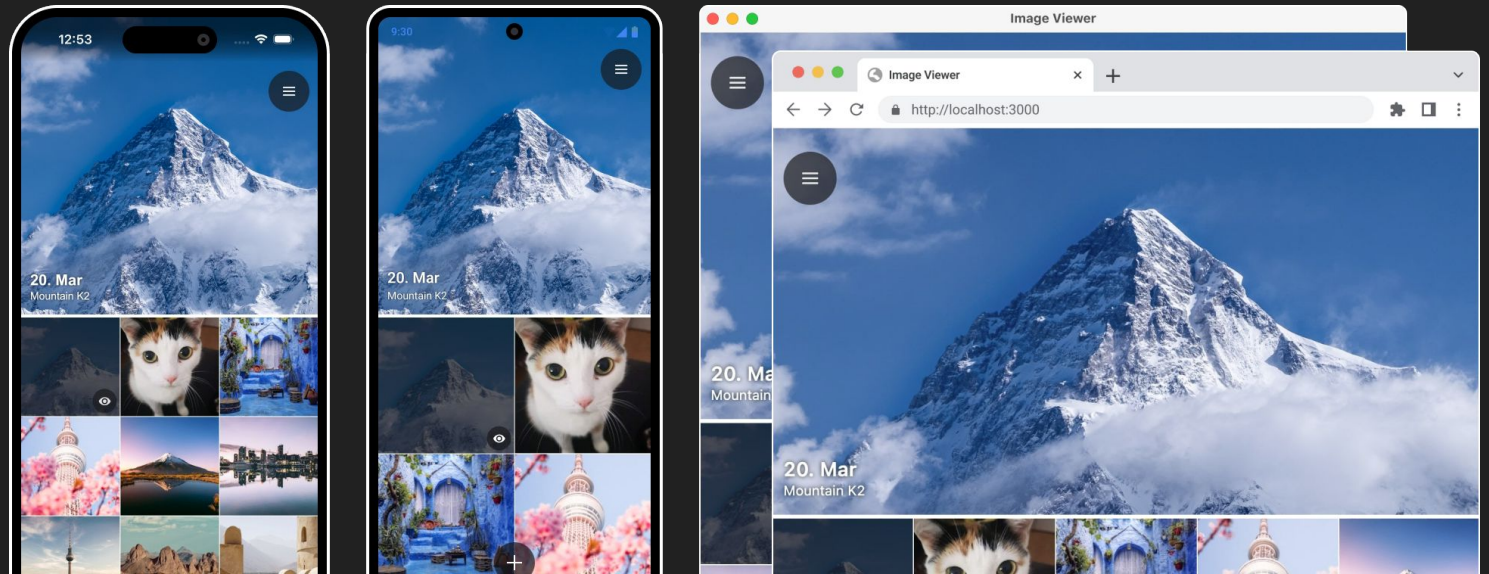
Built on Compose Multiplatform (Android and iOS)



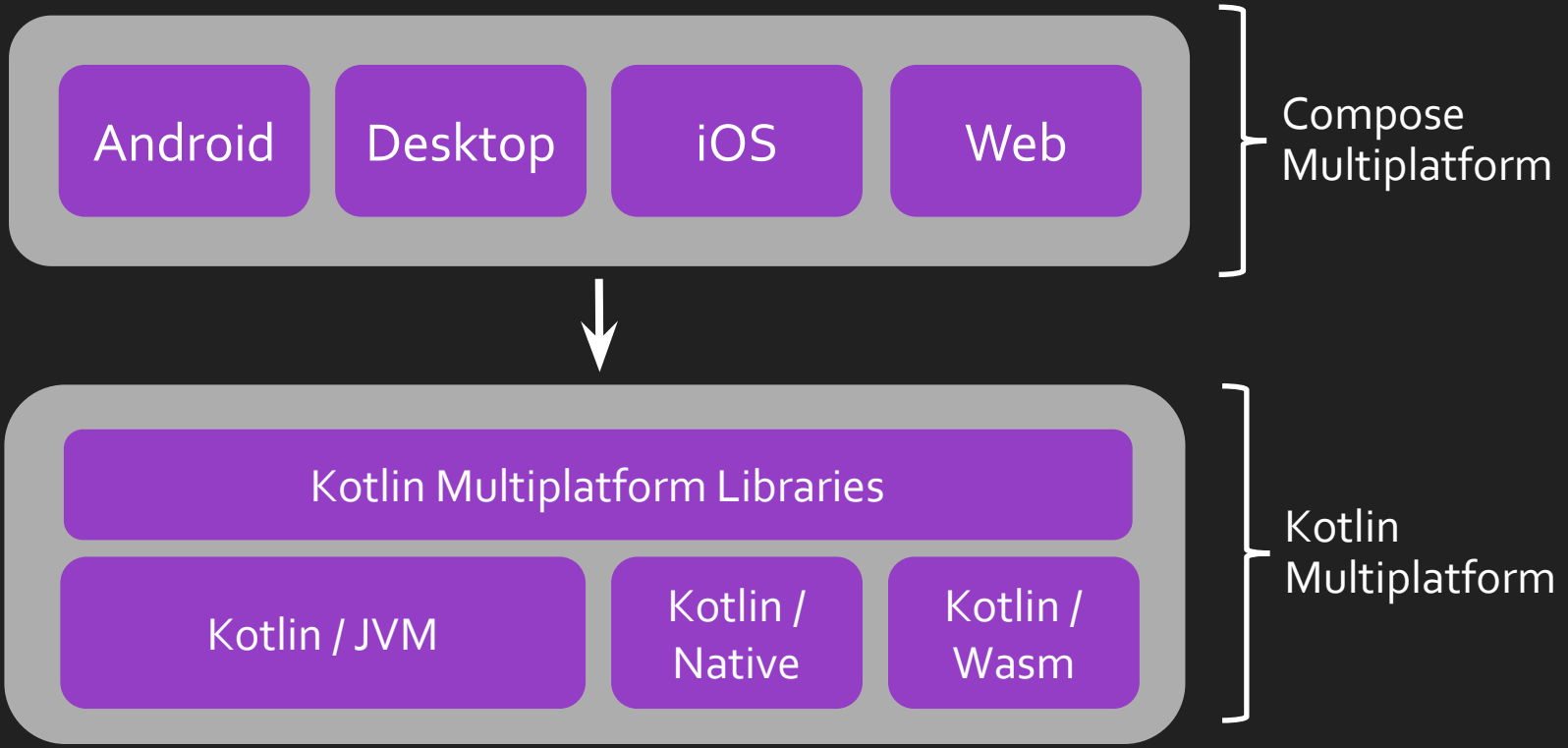
The ImageViewer Sample Application

An example image gallery with camera and map support

Built on Compose Multiplatform (Desktop, Android and iOS)



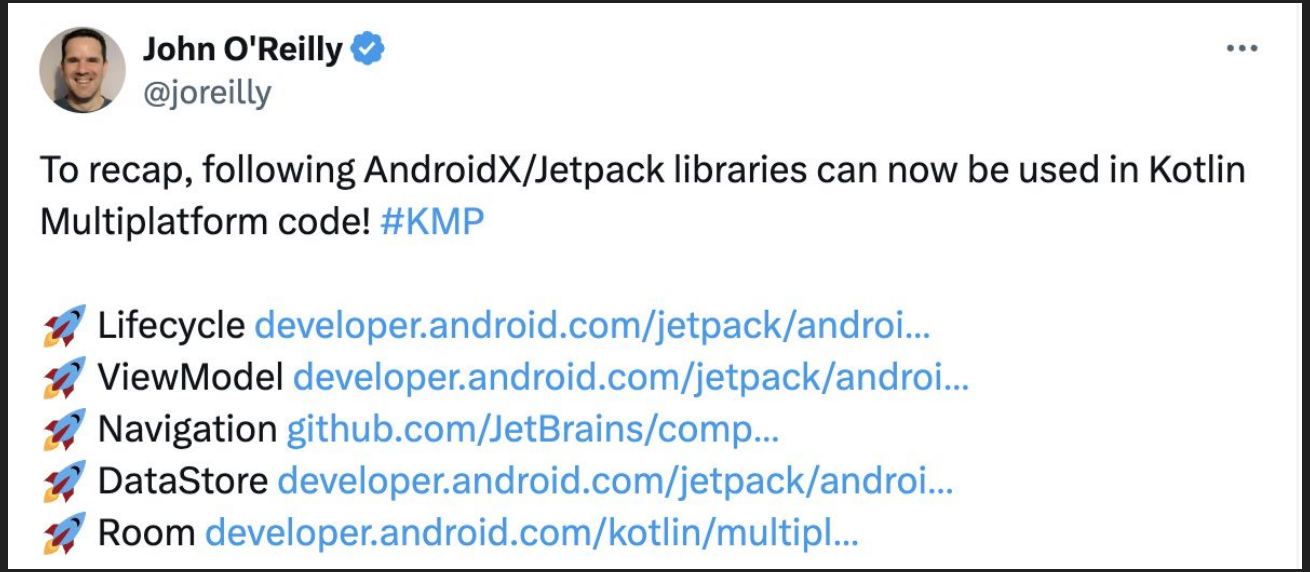
Compose & Kotlin Multiplatform & Libraries



Some Kotlin Multiplatform Libraries

Koin	Dependency Injection framework
KMP-ObservableViewModel	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

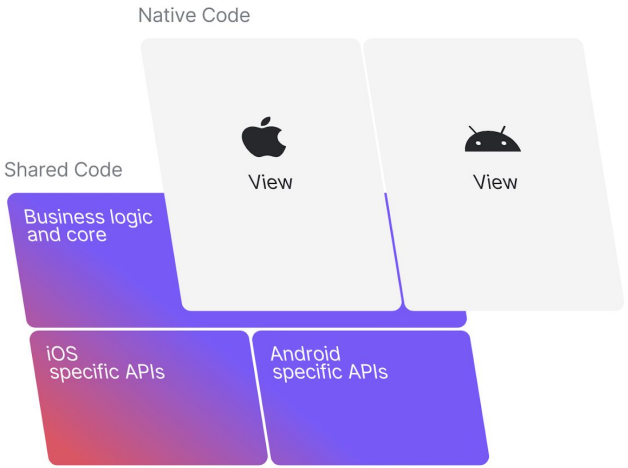
Great News!


John O'Reilly
@joreilly
To recap, following AndroidX/Jetpack libraries can now be used in Kotlin Multiplatform code! #KMP
🚀 Lifecycle developer.android.com/jetpack/androi...
🚀 ViewModel developer.android.com/jetpack/androi...
🚀 Navigation github.com/JetBrains/comp...
🚀 DataStore developer.android.com/jetpack/androi...
🚀 Room developer.android.com/kotlin/multipl...

github.com

README.MD

Awesome KMM



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

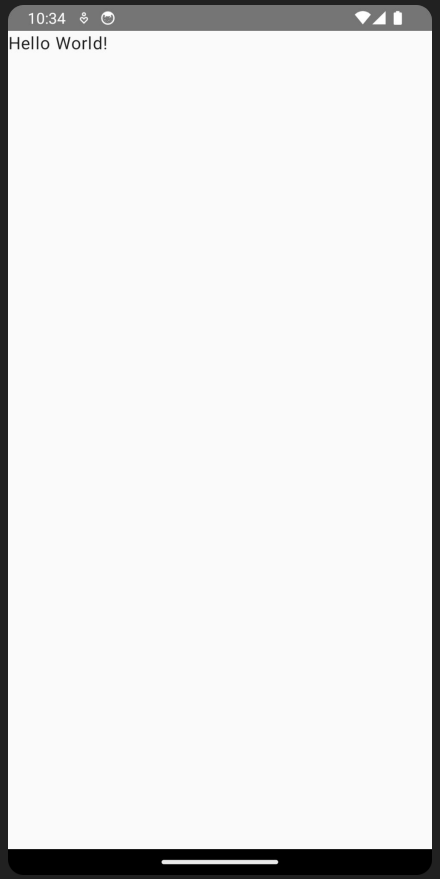
Hello Compose Multiplatform 👋

Concepts

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

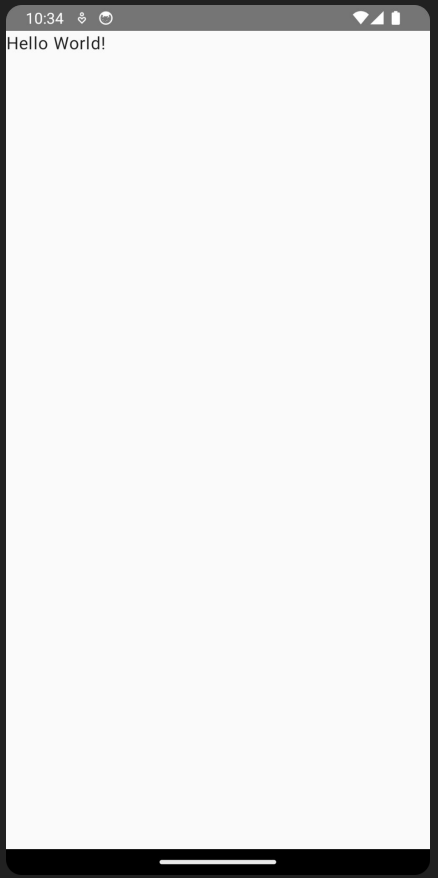
Adding a Text Composable

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



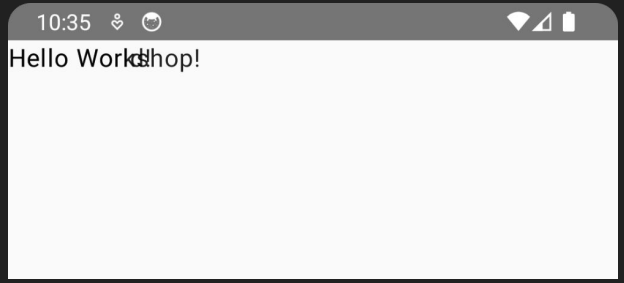
Defining Your Own Composable

```
@Composable
fun App() {
    MaterialTheme {
        Hello()
    }
}

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



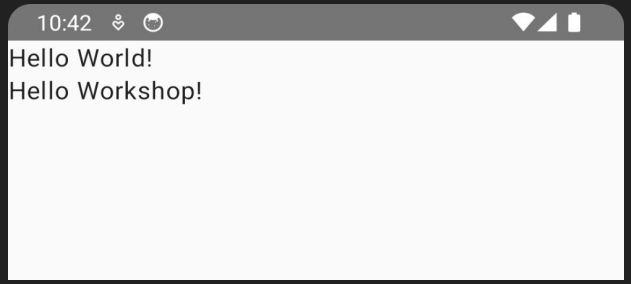
Adding Multiple Texts

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



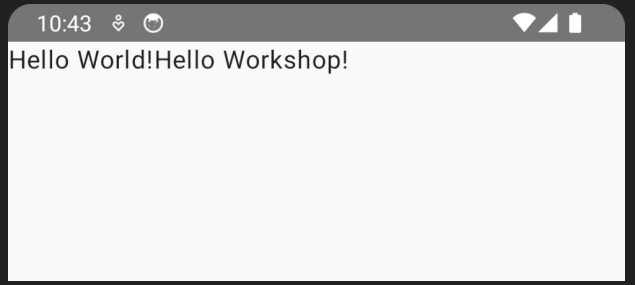
Adding Multiple Texts - Vertical

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



Adding Multiple Texts - Horizontal

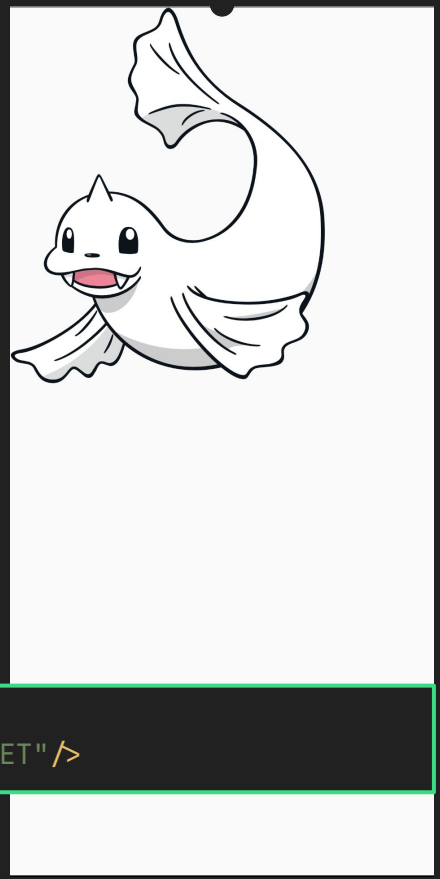
```
@Composable
fun Hello(){
    Row {
        Text("Hello World!")
        Text("Hello Workshop!")
    }
}
```

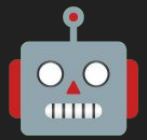


Adding an Image
Using Compose Image Loader

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

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





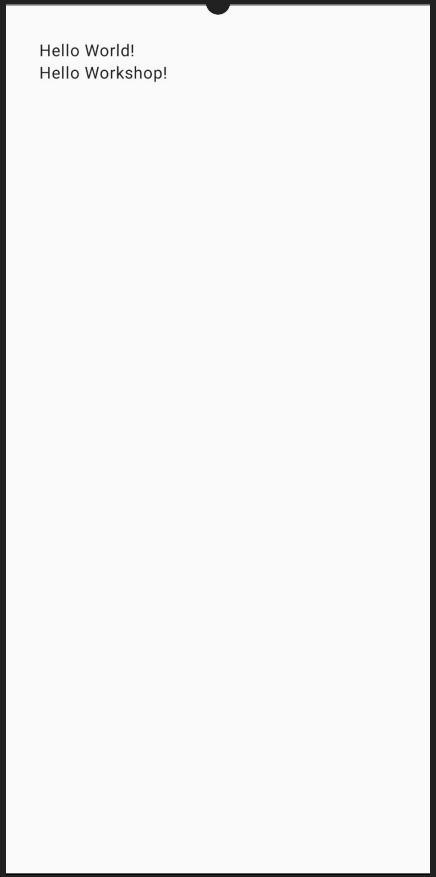
```
<uses-permission
android:name="android.permission.INTERNET"/>
```

Why not Coil?

- An image loading library for Android backed by Kotlin Coroutines
- From v3 onwards, it is also for Kotlin Multiplatform
- But it’s still in pre-release (alpha06)
- It’s also slightly more complicated to use than Compose Image Loader (need to setup an ImageLoader with a context)
- For the curious: https://proandroiddev.com/coil-for-compose-multiplatform-5745ea76356f

Configuring Padding

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

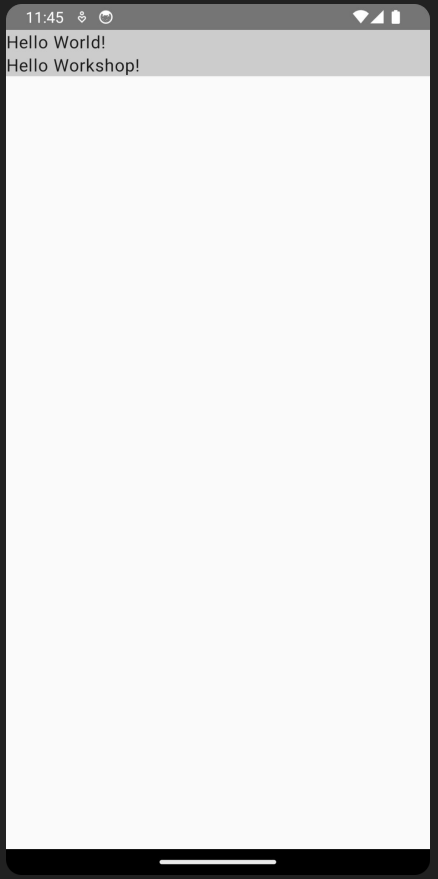


Configuring Sizing - Width

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

OR

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

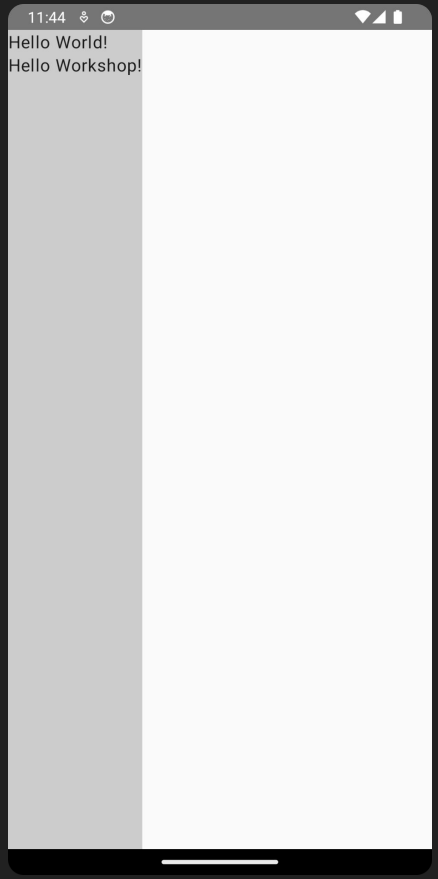


Configuring Sizing - Height

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

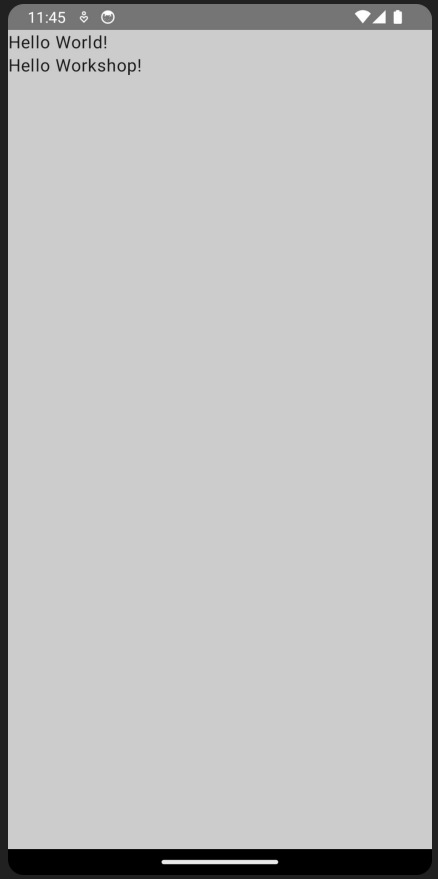
OR

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



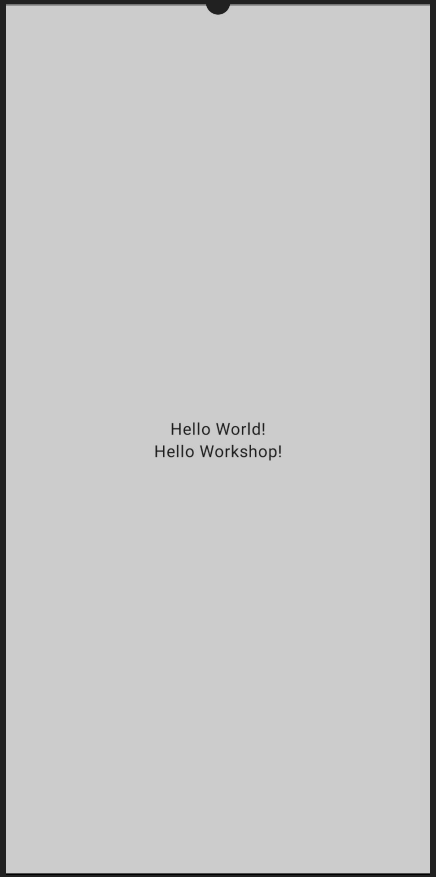
Configuring Sizing - Size

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



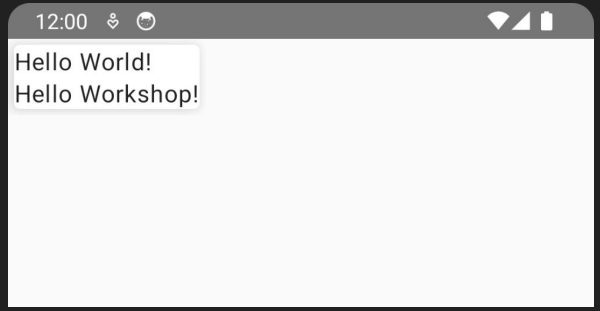
Centering

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



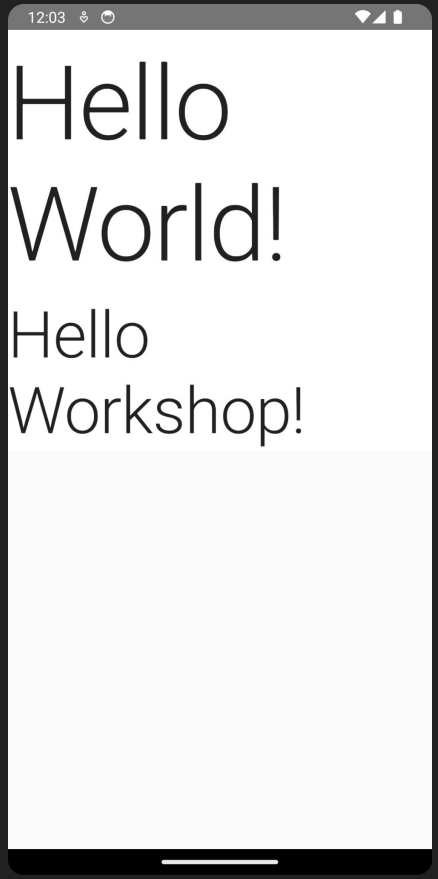
Creating a Card

```
@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
        )
    }
}
```

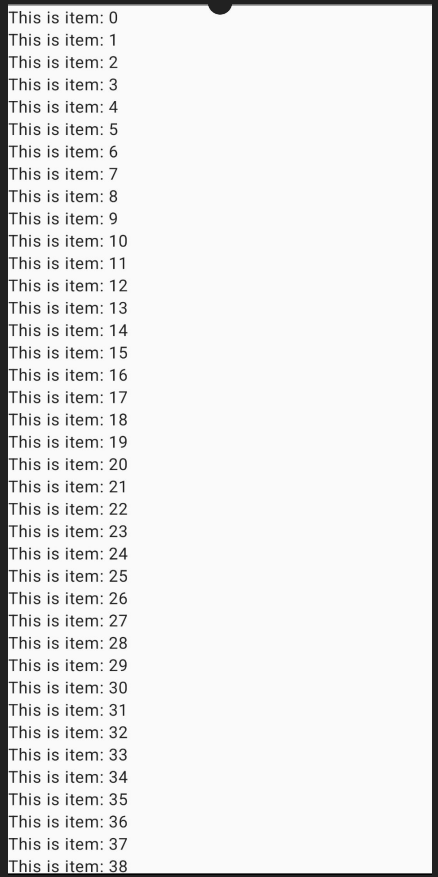


Formatting Text

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

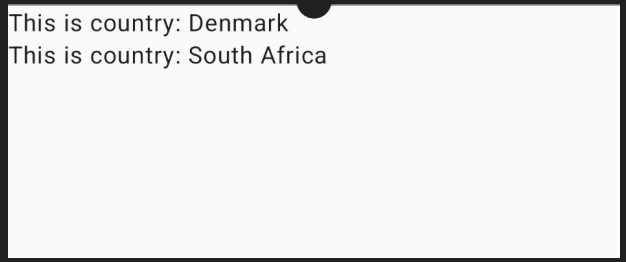
Lists - Number of Items

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



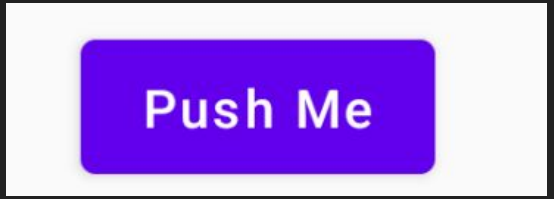
Lists - Collection

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



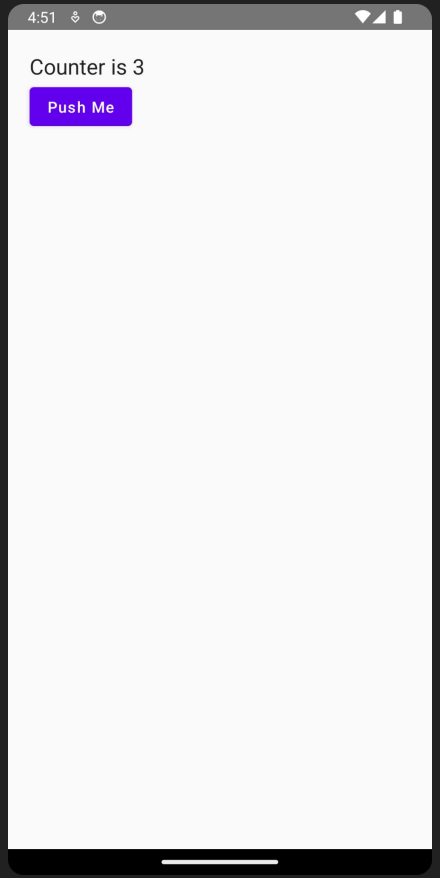
Buttons

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



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

```
@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")
        }
    }
}
```



`mutableStateOf` creates an observable.

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

`remember` stores values across recompositions.

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

Compose Multiplatform navigation is currently experimental and has its limitations* so we'll be looking at a stable alternative.

https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html*

Voyager - Multiplatform Navigation Library for Compose

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

Linear Navigation - Screens

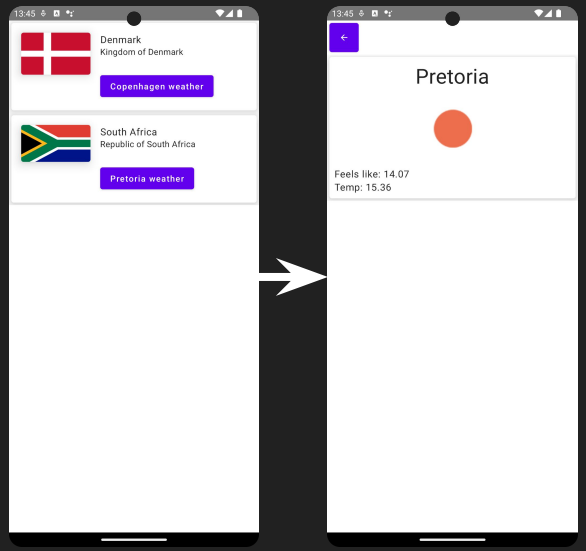
```
class HomeScreen : Screen {
    @Composable
    override fun Content() {
        // Contents of Screen
    }
}
```

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

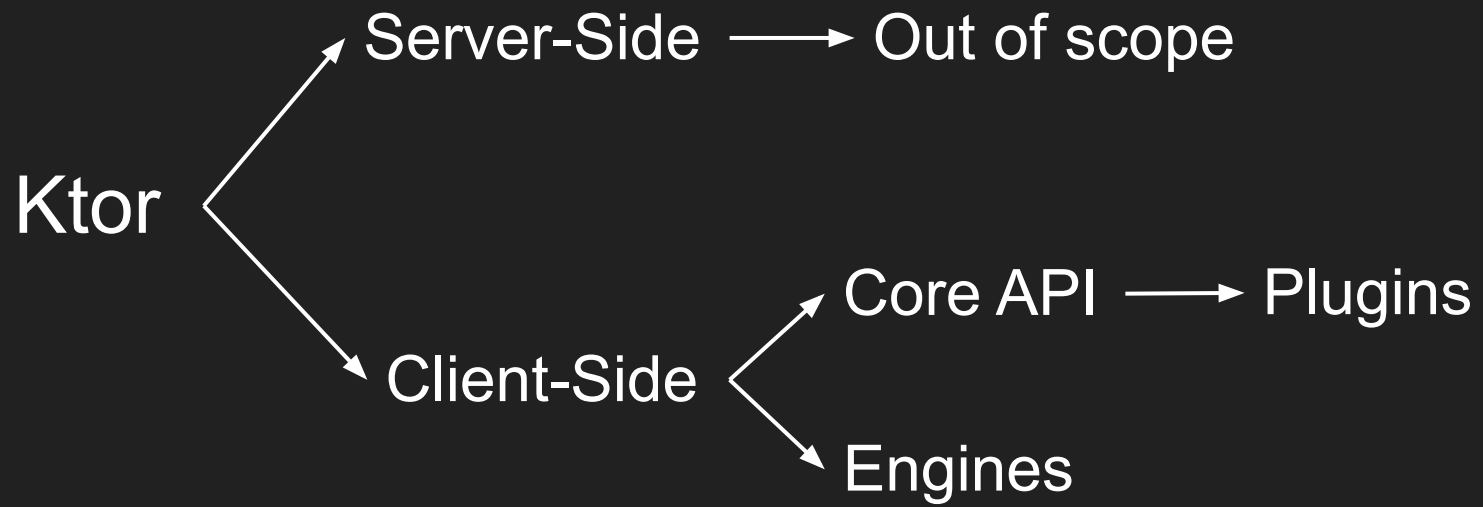
Summary

- 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 an alternative 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


Ktor
Server-Side
Out of scope
Client-Side
Core API
Plugins
Engines

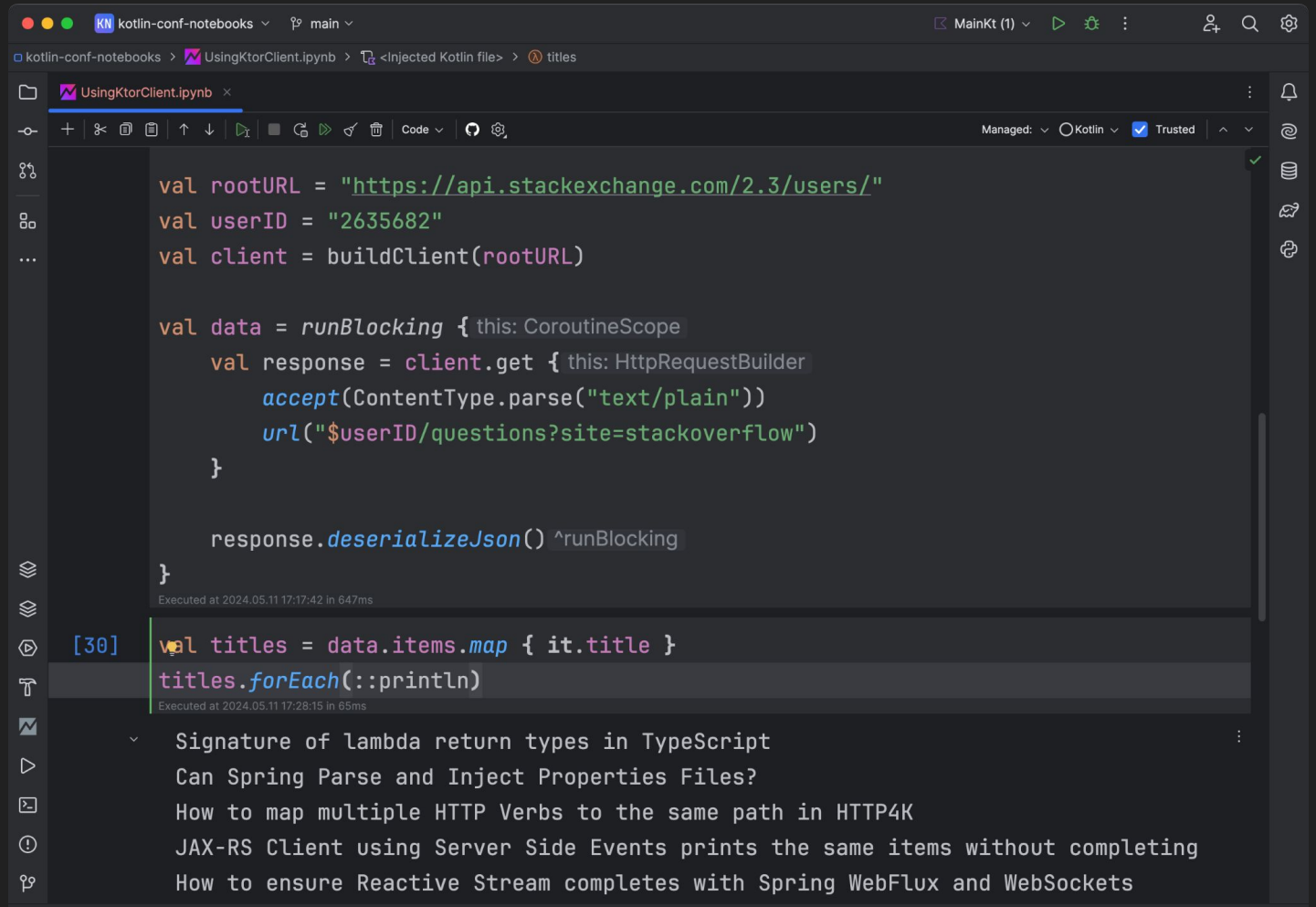
Configuring Ktor Client

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

NB Kotlin Notebook users have a shortcut

- ‘`%use ktor-client`’ provides immediate access to the Ktor Client
- Notebooks are a great way to prototype your networking code


kotlin-conf-notebooks
main
MainKt (1)
kotlin-conf-notebooks > UsingKtorClient.ipynb > <Injected Kotlin file> > titles
UsingKtorClient.ipynb
Code
Managed:
Kotlin
Trusted
val rootURL = "https://api.stackexchange.com/2.3/users/"
val userID = "2635682"
val client = buildClient(rootURL)
val data = runBlocking { this: CoroutineScope
val response = client.get { this: HttpRequestBuilder
accept(ContentType.parse("text/plain"))
url("$userID/questions?site=stackoverflow")
}
response.deserializeJson() ^runBlocking
}
Executed at 2024.05.11 17:17:42 in 647ms
[30]
val titles = data.items.map { it.title }
titles.forEach(::println)
Executed at 2024.05.11 17:28:15 in 65ms
Signature of lambda return types in TypeScript
Can Spring Parse and Inject Properties Files?
How to map multiple HTTP Verbs to the same path in HTTP4K
JAX-RS Client using Server Side Events prints the same items without completing
How to ensure Reactive Stream completes with Spring WebFlux and WebSockets
40:26
LF
UTF-8
4 spaces
main

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

The Ktor philosophy is ‘no magic annotations’

- Plugins on the classpath do not register themselves
- You must explicitly configure them by calling `install`
- This may differ from other frameworks you have used

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")
  }
}
```

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`

Configuring Kotlinx.serialization - Compiler Plugin

```
plugins {
    ...

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

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

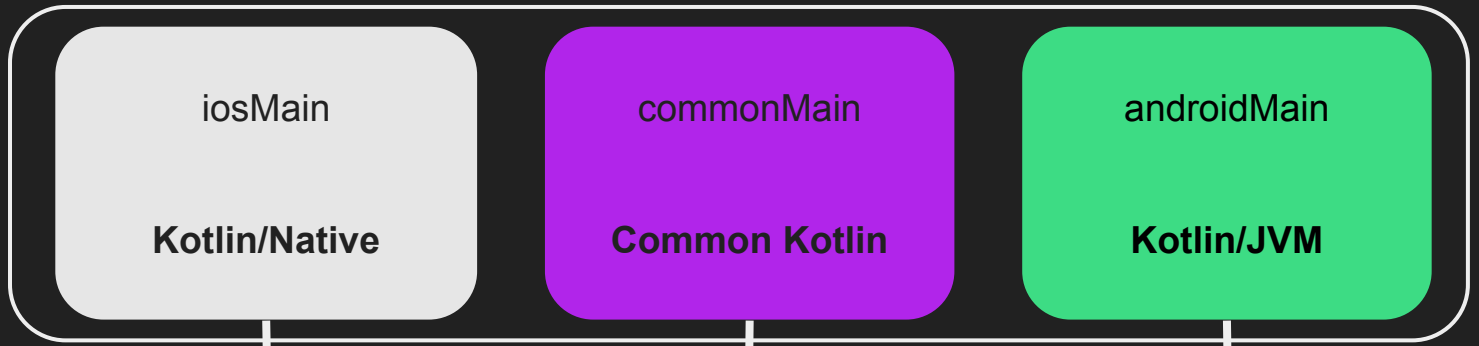
Demo: Wizard `expect`/`actual` Usage

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(s) 😄

Expect/actual functions




```
expect fun randomUUID(): String
```

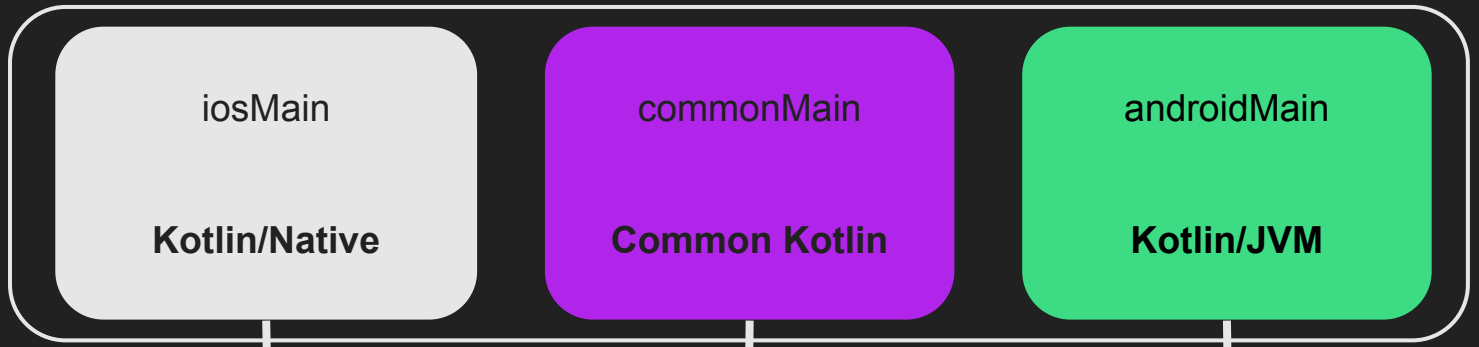
```
import platform.Foundation.NSUUID
actual fun randomUUID(): String =
NSUUID().UUIDString()
```

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

Use `expect/actual` functions for **simple** cases

Use `interfaces` for more **complex** cases

interfaces



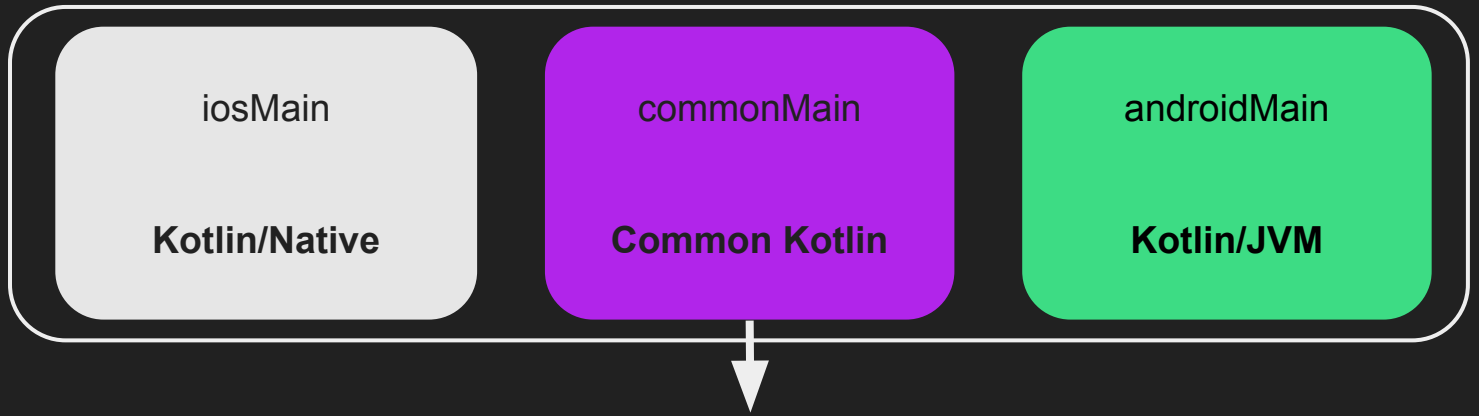

```
interface Platform {
    val name: String
}
```

```
class iOSPlatform: Platform
```

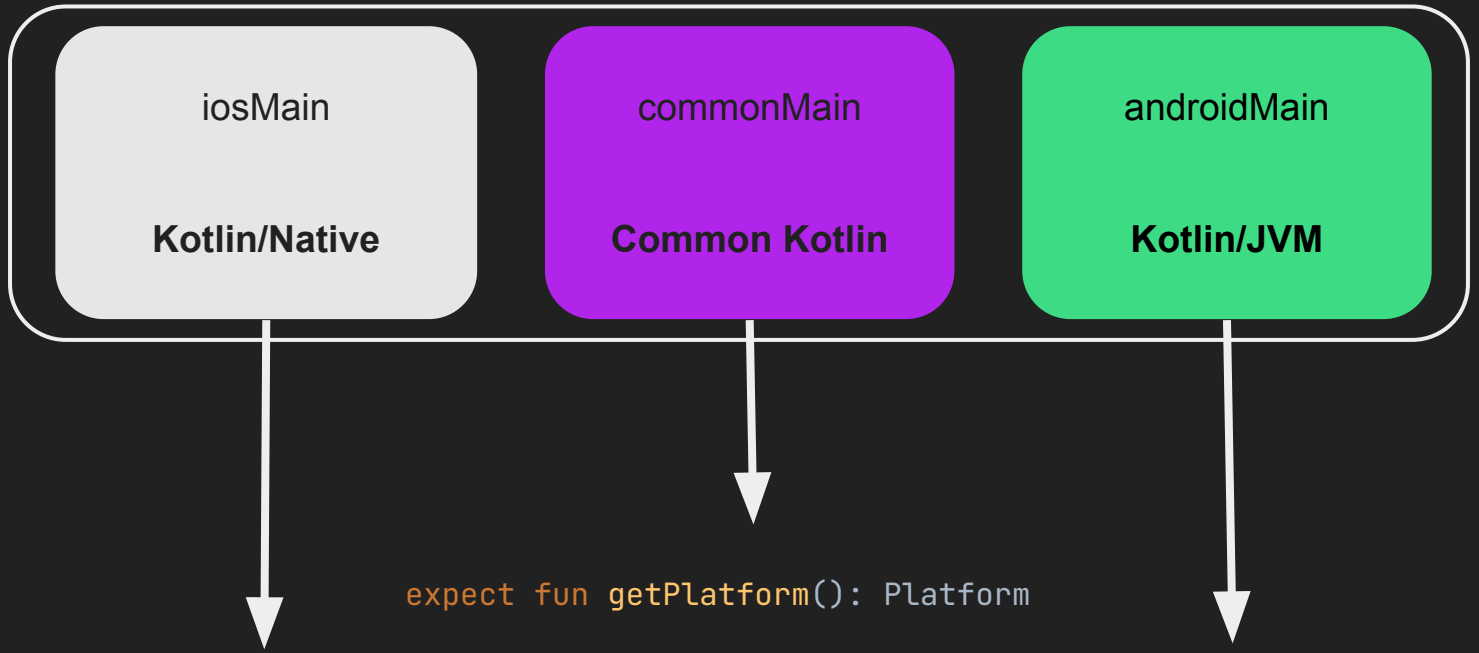
```
class AndroidPlatform: Platform
```

But how do we actually provide the implementation?

Factory Functions




```
interface Platform {
    val name: String
}

expect fun getPlatform(): Platform
```

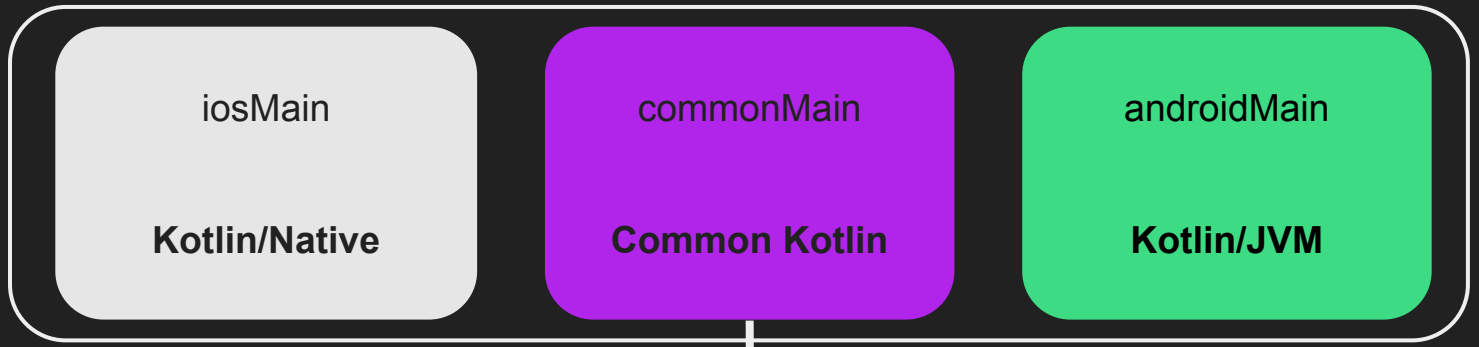



```
expect fun getPlatform(): Platform
```

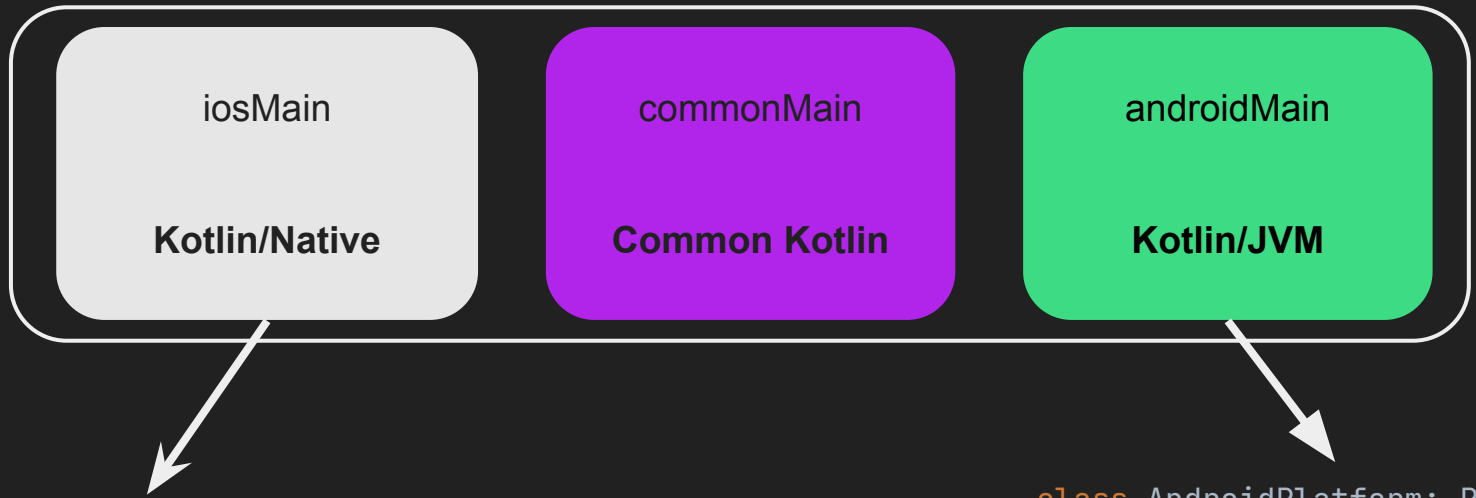
```
class iOSPlatform: Platform
…

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
}
```

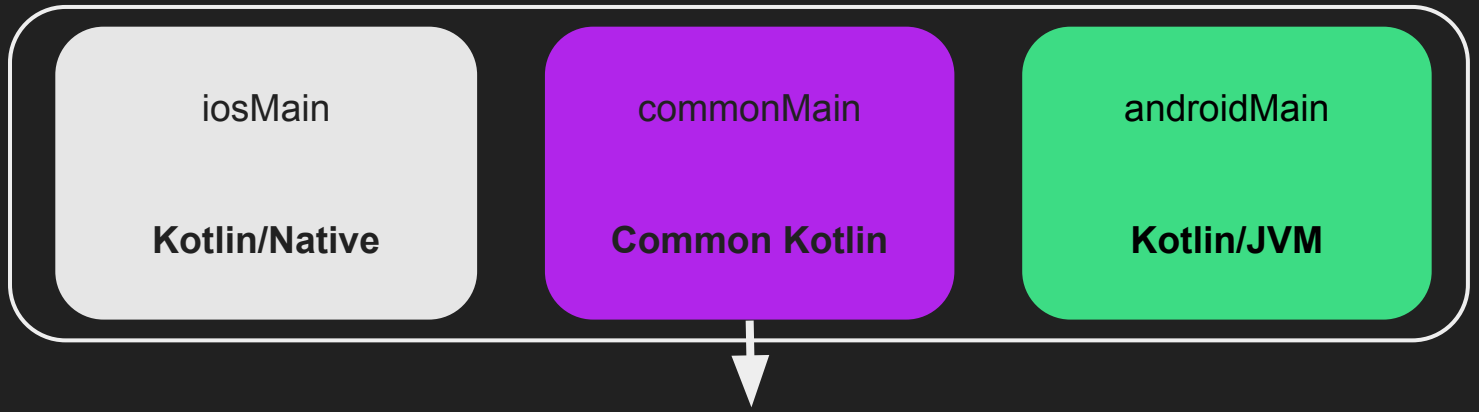



```
class iOSPlatform: Platform
…

@main
struct iOSApp : App {
   init() {
       app(IOSPlatform())
   }
}
```

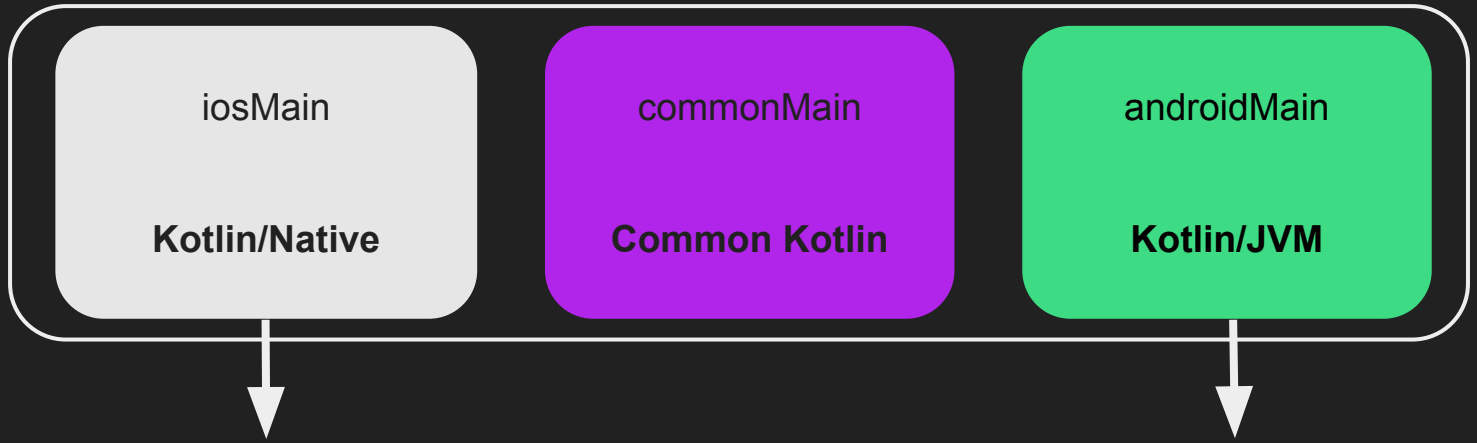
```
class AndroidPlatform: Platform
…

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()
   }
}
```

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`

Discussion Time

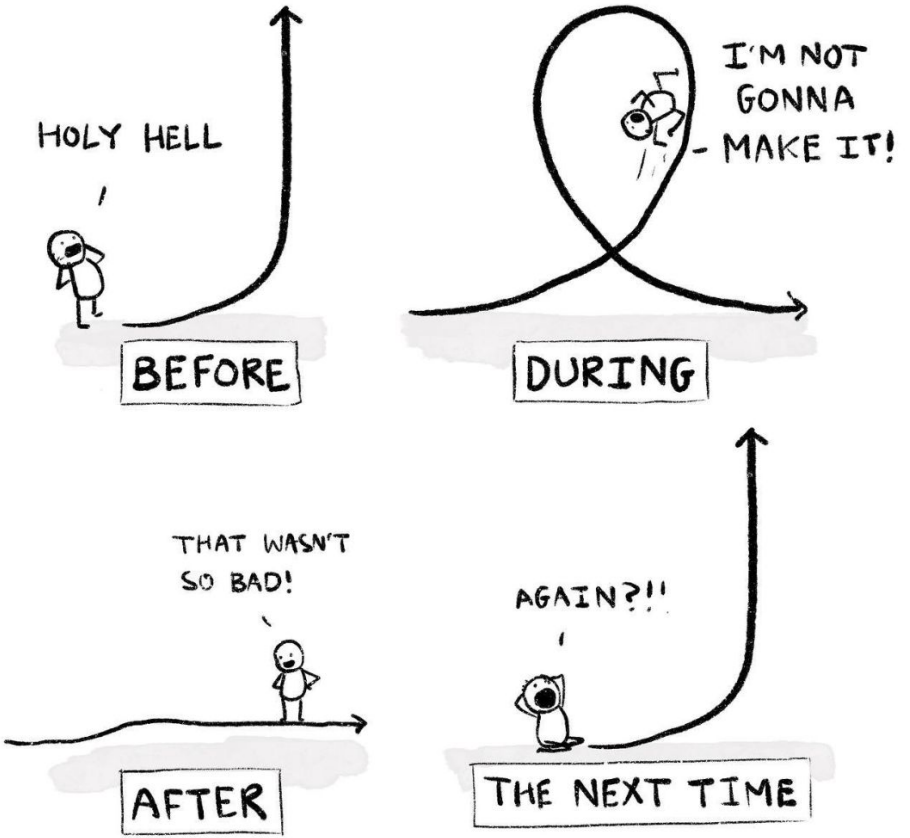
What is your impressions 👁️ of the technology now that you've given it a try?

What are your next steps  ?

How would you convince your team 🤝 to also try it?

Closing

HOW LEARNING CURVES FEEL...



@STEINBERGDRAWSCARTOONS

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

JETBRAINS