Getting the most out of shared code in Kotlin Multiplatform

Maciej Procyk

Kotlin Multiplatform Tooling

JetBrains

The background story

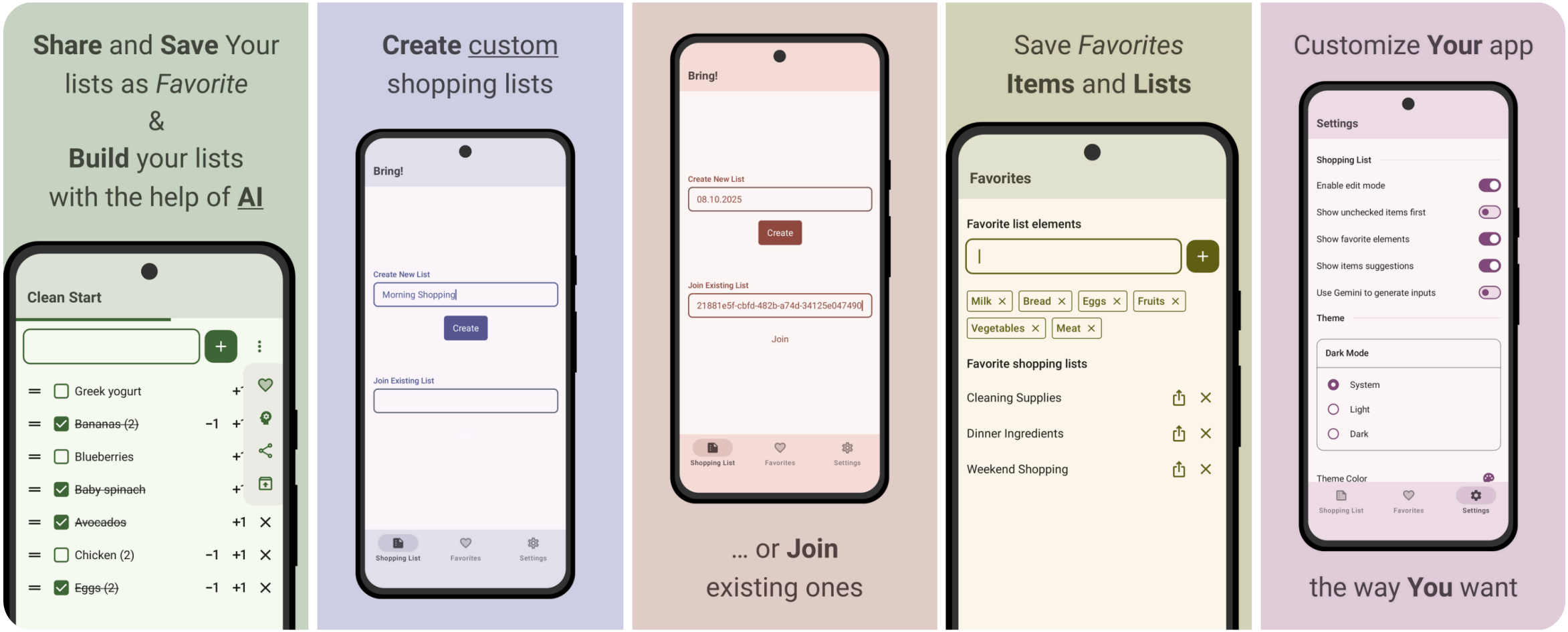
The idea

- everyone needs to do the shopping
- but not everyone likes commercial solutions
- let's write a usable app for shopping with my family
- write code once, run everywhere
- deploy to the world

The result

- a usable app published on Google Play and available as a web page
- a deployed app server with an independent database
- an app capable of full shopping list management

The result


Share and **Save** Your lists as *Favorite* & **Build** your lists with the help of **AI**
Clean Start
Greek yogurt
Bananas (2)
Blueberries
Baby spinach
Avocados
Chicken (2)
Eggs (2)
Create custom shopping lists
Bring!
Create New List
Morning Shopping
Create
Join Existing List
Shopping List
Favorites
Settings
Bring!
Create New List
08.10.2025
Create
Join Existing List
21881e5f-cbfd-482b-a74d-34125e047490
Join
... or **Join** existing ones
Save *Favorites* **Items** and **Lists**
Favorites
Favorite list elements
Milk
Bread
Eggs
Fruits
Vegetables
Meat
Favorite shopping lists
Cleaning Supplies
Dinner Ingredients
Weekend Shopping
Customize **Your** app
Settings
Shopping List
Enable edit mode
Show unchecked items first
Show favorite elements
Show items suggestions
Use Gemini to generate inputs
Theme
Dark Mode
System
Light
Dark
Theme Color
the way **You** want

The technical part

The goals

- increase awareness of Kotlin Multiplatform (KMP)
- show how to start your journey with KMP
- share crucial and tricky parts of KMP development
- showcase a nice combination of KMP libraries as a generic development stack
- share a battle-tested reference of a multiplatform app

Why use Kotlin Multiplatform?

- make your project easy to fix and improve
- it runs everywhere (where reasonable)
- integrates at various levels — you decide where
- lets you use platform-native APIs
- you end up with platform-native binaries

Kotlin Multiplatform — the general idea

- you define which targets you want your app to compile for
- look for KMP libraries you need that expose functionality in the `common` source set
- use `expect` and `actual` keywords to define platform-specific code
- build the app for each of the targets separately

Where to start?

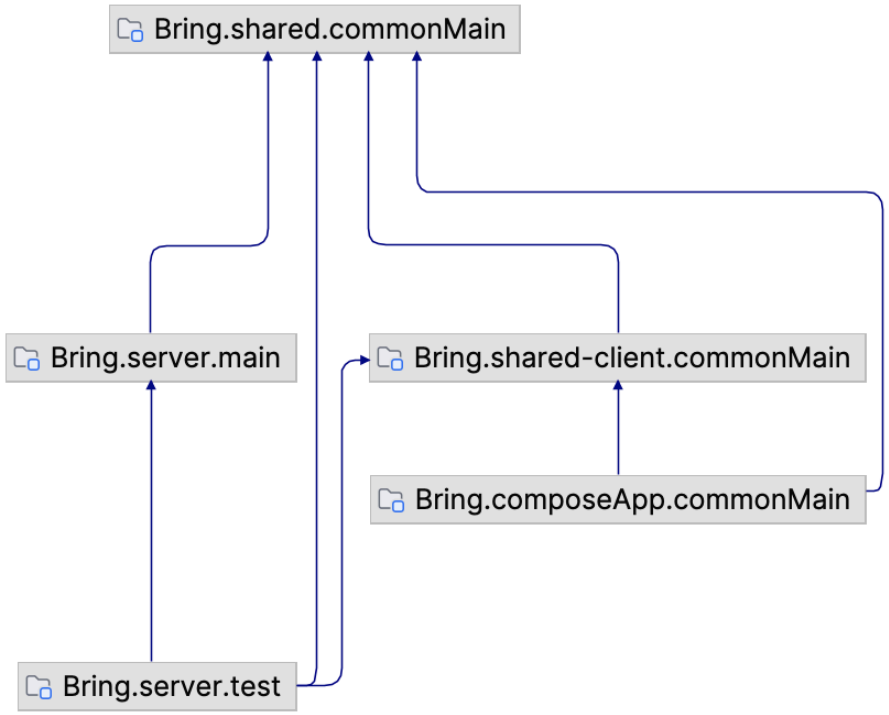
- keep the official KMP docs open
- check out the nice collection of libraries at klibs.io
- project creation is as simple as going to kmp.new
- see `README.md` for first-time instructions; the IDE will help you a lot
  - the story begins with a functional Compose Multiplatform (CMP) app
  - which you can easily iterate on
  - where the desktop JVM target is your friend, since you get Compose Hot Reload (CHR)

How you can start — in-IDE demo

1. Use the latest IntelliJ IDEA or Android Studio and install the Kotlin Multiplatform plugin
2. Go to the `Kotlin Multiplatform` wizard to create a new project
3. Select your targets and start coding!
4. Optional: Try live coding with CHR

Insights and tips

`Bring!` project structure



- `shared` module contains data models, RPC APIs, common logic, utilities, etc.
- `shared-client` module contains the client (used by `composeApp` and `server` integration tests)
- `composeApp` module contains the client app
- `server` module contains the server-side implementation

`Bring!` project structure

```
bring
├─ composeApp
│  └─ src
│     ├─ androidMain
│     ├─ commonMain
│     ├─ iosMain
│     ├─ jvmMain
│     └─ webMain
├─ server
│  └─ src
│     ├─ main
│     └─ test
├─ shared
│  └─ src
│     └─ commonMain
└─ shared-client
   └─ src
      ├─ commonMain
      └─ ...
```

- `shared` module contains data models, RPC APIs, common logic, utilities, etc.
- `shared-client` module contains the client (used by `composeApp` and `server` integration tests)
- `composeApp` module contains the client app
- `server` module contains the server-side implementation

Gradle configuration files — `shared` targets

```
kotlin {
    androidTarget {
        compilerOptions {
            jvmTarget = JvmTarget.JVM_11
        }
    }

    iosArm64()
    iosSimulatorArm64()

    jvm()

    @OptIn(ExperimentalWasmDsl::class)
    wasmJs { browser() }

    js { browser() }
```

- `kotlin { ... }` block defines targets in `build.gradle.kts`
- define multiple targets in a single module
- configure each target separately with IDE completion inside the `{ ... }` block
- add them step by step, each time trying to build the project for that specific target

Gradle configuration files — `composeApp` targets

```
kotlin {
    androidTarget {
        compilerOptions {
            jvmTarget = JvmTarget.JVM_11
        }
    }
    listOf(
        iosArm64(),
        iosSimulatorArm64()
    ).forEach { iosTarget ->
        iosTarget.binaries.framework {
            baseName = "ComposeApp"
            isStatic = true
        }
    }
    jvm()
    js {
        browser()
        binaries.executable()
    }
    @OptIn(ExperimentalWasmDsl::class)
    wasmJs {
        browser()
        binaries.executable()
    }
```

- explicitly define final build executables
- JVM is handled by `compose.desktop { ... }`, while Android is handled via `android { ... }` and the `AndroidManifest.xml`
- the defaults for JVM, JS, and WASM are `main()` entry points
- use the `webMain` common source set so both web targets can be built from a single source set for simplicity (search in docs for Gradle task `composeCompatibilityBrowserDistribution`)

Gradle configuration files — `shared` source sets and dependencies

```
kotlin {
    // targets...

    sourceSets {
        commonMain.dependencies {
            // Multiplatform dependencies
        }
        commonTest.dependencies {
            implementation(libs.kotlin.test)
        }
    }
```

- source sets define the set of files compiled against a specific group of targets
- some predefined ones (`commonMain`, `commonTest`, `jvmMain`, `jvmTest`, ...)
- you can define your own that combine some of them
- each source set is commonized to the APIs available on the targets it's used with

Gradle configuration files — `composeApp` source sets and dependencies

```
kotlin {
    // targets...

    sourceSets {
        androidMain.dependencies {
            // ...
            implementation(libs.androidx.activity.compose)
        }
        commonMain.dependencies {
            implementation(compose.runtime)
            // ...
            implementation(libs.androidx.lifecycle.viewmodelCompose)
            implementation(libs.androidx.lifecycle.runtimeCompose)
            implementation(projects.shared)
        }
        jvmMain.dependencies {
            // ...
            implementation(libs.kotlinx.coroutinesSwing)
        }
    }
```

- target-specific source sets can have target-specific dependencies
- each is resolved in the relevant source set
- it's enough to depend on the `projects.shared` project to get its `commonMain` APIs

Different behavior on each platform — `expect`/`actual` in action

```
interface Platform {
    val name: String
}

expect fun getCurrentPlatform(): Platform

class WasmPlatform : Platform {
    override val name: String =
        "Web with Kotlin/Wasm"
}

actual fun getCurrentPlatform(): Platform =
    WasmPlatform()

class AndroidPlatform : Platform {
    override val name: String =
        "Android ${VERSION.SDK_INT}"
}

actual fun getCurrentPlatform(): Platform =
    AndroidPlatform()
```

- common code with `expect`
- platform-specific code with platform-specific APIs
- WASM-specific behavior
- Android-specific behavior
- ...
- differences in behavior on each platform

Different APIs on each platform — `expect`/`actual` in action

```
internal expect suspend fun onShareList(listId: String, context: Context)

internal actual suspend fun onShareList(listId: String, context: Context) {
    val clipEntry = withPlainText("${window.location.origin}/#${listId}")
    context.clipboard.setClipEntry(clipEntry)
    context.showSnackbar(Res.string.copied_list_url_to_clipboard)
}

internal actual suspend fun onShareList(listId: String, context: Context) {
    val shareIntent = Intent(ACTION_SEND).apply {
        type = "text/plain"
        putExtra(Intent.EXTRA_TEXT, listId)
    }
    val intent = createChooser(shareIntent, null).apply {
        addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
    }
    context.platformContext.context.startActivity(intent)

    context.clipboard.setClipEntry(ClipEntry(newPlainText(null, listId)))
    context.showSnackbar(Res.string.copied_list_id_to_clipboard)
}
```

- common code with `expect`
- platform-specific code with platform-specific APIs
- web implementation using a library-provided `ClipEntry` and a platform-specific API
- Android implementation configures a custom Intent to enable sharing the list

`shared` part — data models, core logic, utilities

```
@Serializable
data class ShoppingListData(
    @CborLabel(0) val id: Uuid,
    @CborLabel(1) val name: String,
    @CborLabel(2) val byUserId: Uuid,
    @Serializable(InstantSerializer::class)
    @CborLabel(3) val createdAt: Instant,
    @CborLabel(4) val items: List<ShoppingListItemData>,
)
```

- located in the `shared` module
- available to the `server` and `composeApp` modules
- define the data models being transferred
- `kotlinx-serialization` is used to serialize/deserialize them
- supports many formats, including JSON, CBOR, Protobuf, etc.
- control how data is serialized/deserialized with custom serializers

`shared` part — data models, core logic, utilities

```
object InstantSerializer : KSerializer<Instant> {
    override val descriptor: SerialDescriptor
        get() = SerialDescriptor(
            "in.procyk.bring.InstantSerializer",
            Long.serializer().descriptor,
        )

    override fun serialize(encoder: Encoder, value: Instant) {
        encoder.encodeLong(value.toEpochMilliseconds())
    }

    override fun deserialize(decoder: Decoder): Instant {
        return Instant.fromEpochMilliseconds(decoder.decodeLong())
    }
}
```

- located in the `shared` module
- available to the `server` and `composeApp` modules
- define the data models being transferred
- `kotlinx-serialization` is used to serialize/deserialize them
- supports many formats, including JSON, CBOR, Protobuf, etc.
- control how data is serialized/deserialized with custom serializers

`shared` part — client/server communication

```
@Rpc
interface ShoppingListService {

    @Serializable
    enum class CreateNewShoppingListError {
        Internal,
        ExtractionError,
        InvalidName
    }

    suspend fun createNewShoppingList(
        userId: Uuid,
        input: String,
    ): Either<Uuid, CreateNewShoppingListError>

    // ...
}
```

- `kotlinx-rpc` is an (experimental) high-level abstraction for communication
- allows customization of the actual transport layer
- define the API for the client and server
- implement the API on the server side as plain functions

`server` part — client/server communication

```
internal class ShoppingListServiceImpl : ShoppingListService {

    override suspend fun createNewShoppingList(
        userId: Uuid,
        input: String,
    ): Either<Uuid, CreateNewShoppingListError> {
        if (input.isBlank()) {
            return CreateNewShoppingListError.InvalidName.right()
        }
        val (title, ingredients) = when {
            !extractor.supports(input) -> input to emptyList()
            else -> extractListEntries(input, onError = {
                return CreateNewShoppingListError.ExtractionError.right()
            })
        }
        return txn(CreateNewShoppingListError.Internal) {
            createNewShoppingList(userId, title) { listId ->
                addEntriesToShoppingListInTransaction(userId.toJavaUuid(), listId, ingredients).left()
            }
        }
    }

    // ...
```

- simply override defined methods on the server side
- connections from clients are established automatically
- uses Ktor transports under the hood (WebSockets and HTTP), but gRPC is also supported
- each used type must be `@Serializable`
- to get continuous updates of `T` from the server, use `Flow<T>`

`server` part — database access

```
internal object ShoppingListsTable
    : UUIDTable(name = "shopping_lists") {
    val name = text("name", eagerLoading = true)
    val byUserId = uuid("by_user_id")
    val createdAt = timestamp("created_at")
}

internal class ShoppingListEntity(id: EntityID<UUID>)
    : UUIDEntity(id) {
    companion object :
        UUIDEntityClass<ShoppingListEntity>(ShoppingListsTable)

    var name by ShoppingListsTable.name
    var byUserId by ShoppingListsTable.byUserId
    var createdAt by ShoppingListsTable.createdAt

    val items by ShoppingListItemEntity referrersOn
            ShoppingListItemsTable.listId
}
```

- Exposed SQL library for database access
- lets you operate at various levels of abstraction:
  - DSL for typed SQL queries
  - DAO for ORM-like queries

`server` part — database access

```
return txn(GetUserShoppingListSuggestionsError.Internal) {
    val count = ShoppingListItemsTable.name.count().alias("count")
    ShoppingListItemsTable.select(
        ShoppingListItemsTable.name, count,
    ).where {
        ShoppingListItemsTable.byUserId eq userId.toJavaUuid()
    }
        .groupBy(ShoppingListItemsTable.name)
        .orderBy(count, SortOrder.DESC)
        .limit(maxOf(limit, 0))
        .mapLazy { it[ShoppingListItemsTable.name] }
        .toSet()
        .let(::UserShoppingListSuggestionsData)
        .left()
}
```

- Exposed SQL library for database access
- lets you operate at various levels of abstraction:
  - DSL for typed SQL queries
  - DAO for ORM-like queries

`server` part — database access

```
private fun <E> createNewShoppingList(
    userId: Uuid,
    name: String,
    addEntries: (listId: EntityID<UUID>) -> Either<Unit, E>
): Either<Uuid, E> {
    val userUUID = userId.toJavaUuid()
    val listId = ShoppingListEntity.new {
        this.name = name
        this.byUserId = userUUID
        this.createdAt = Clock.System.now().toDeprecatedInstant()
    }.id
    addEntries(listId).onRight { return it.right() }
    return listId.value.toKotlinUuid().left()
}
```

- Exposed SQL library for database access
- lets you operate at various levels of abstraction:
  - DSL for typed SQL queries
  - DAO for ORM-like queries

`server` part — server implementation

```
fun main() {
    val dotenv = dotenv {
        ignoreIfMissing = true
        directory = "../"
    }
    Database.init(dotenv)
    val appModule = module {
        single<Dotenv> { dotenv }
    }
    embeddedServer(
        factory = CIO,
        host = dotenv.env("HOST"),
        port = dotenv.env("PORT")
    ) {
        plugins(dotenv, appModule)
        routes()
    }
        .start(wait = true)
}
```

- Ktor framework for building the server

`server` part — server implementation

```
internal fun Application.plugins(
    dotenv: Dotenv,
    appModule: Module,
) {
    install(Krpc) {
        serialization {
            cbor(DefaultCbor)
        }
    }
    install(AutoHeadResponse)
    install(Resources)
    install(ContentNegotiation) {
        cbor(DefaultCbor)
    }
    installCors(dotenv)
    install(Koin) {
        modules(appModule)
    }
}
```

- Ktor framework for building the server
- a huge collection of plugins
- nice integration with `kotlinx-rpc`

`server` part — server implementation

```
internal fun Application.routes() = routing {
    get("/") {
        call.respondRedirect(
            url = "https://bring.procyk.in",
            permanent = true,
        )
    }
    get("/version") {
        call.respond(BuildConfig.VERSION)
    }
    get("/health") {
        call.respond(HttpStatusCode.OK)
    }
    favoriteElementRpc()
    shoppingListRpc()
}
```

- Ktor framework for building the server
- a huge collection of plugins
- nice integration with `kotlinx-rpc`
- handy DSL for configuring routes
- option to use GraalVM Native Image for native builds with the `CIO` engine (see `graalvmNative { ... }` block in `build.gradle.kts`)

`composeApp` part — shared UI

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Column {
        Text("Clicked: $count")
        Button(onClick = { count++ }) {
            Text("+")
        }
        Button(onClick = { count = 0 }) {
            Text("Reset")
        }
    }
}
```

- Compose Multiplatform for building UI
- building a consistent UI between platforms
- based on Android's Jetpack Compose, Compose Multiplatform brings it to many platforms
- a declarative UI — everything in one place

`composeApp` part — shared view models

```
class AppViewModel : ViewModel() {
    private val _showContent = MutableStateFlow(false)
    val showContent = _showContent.asStateFlow()

    private val updateJob = viewModelScope.launch {
        while (currentCoroutineContext().isActive) {
            delay(1.seconds)
            _showContent.update { !it }
        }
    }

    val greeting: String by lazy {
        Greeting().greet()
    }

    fun onToggleShowContent() {
        updateJob.cancel()
        _showContent.update { !it }
    }
}
```

- shared view models
- consistent logic between platforms
- well integrated with `kotlinx-coroutines` via `viewModelScope`
- when you need to use them from native code, you might explore KMP-NativeCoroutines, Decompose, the experimental Swift export

Summary

- Building KMP apps is easy to start and offers a lot of flexibility
- No code duplication thanks to modules and libraries like `shared`
- Well-scaling results — `composeApp` can be compiled to multiple client targets
- The pleasure of building UI with CMP and CHR
- A consistent UI across platforms, with controllable platform-specific behavior
- Building the server side with Ktor gives you full control and can be extended with plugins

Explore more...

- AI model integration with the Koog KMP framework
- Configuration of a Docker image with a natively compiled server
- GitHub Actions for building and publishing the app
- Screenshot tests that automatically render the app UI and export it as `.png` files
- Server integration tests with Testcontainers

Thank you for your attention!

Slides

GitHub

bring.procyk.in

Google Play