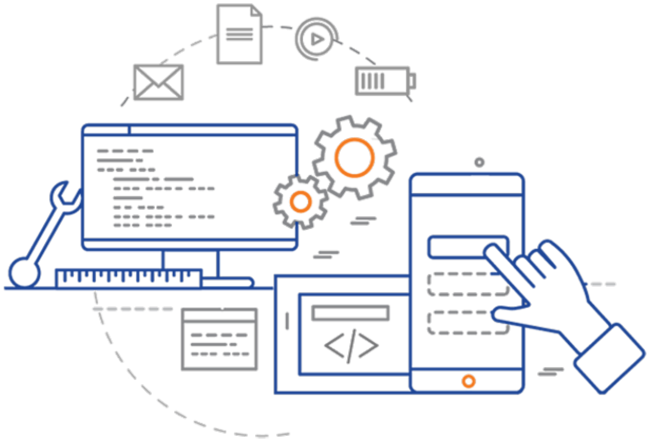
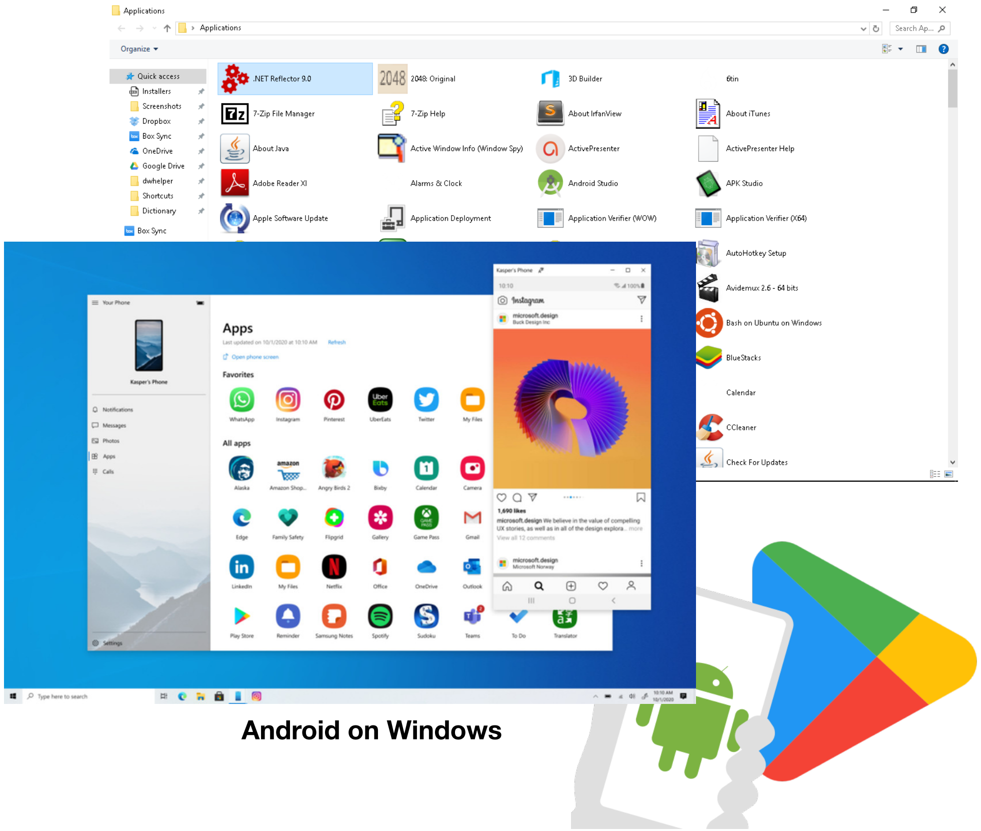


CS 398: Application Development

Kotlin Multiplatform

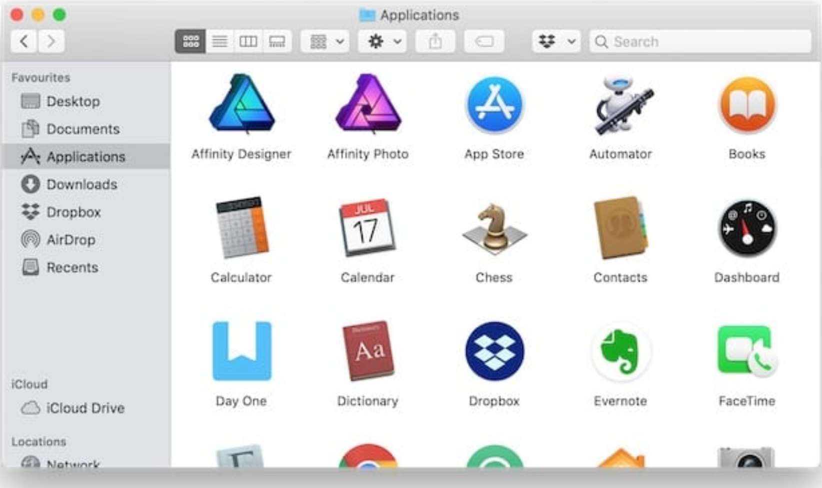
Cross-Platform Development; KMP; Kotlin/Native

What’s our operating environment?



Android on Windows

Android



Mac

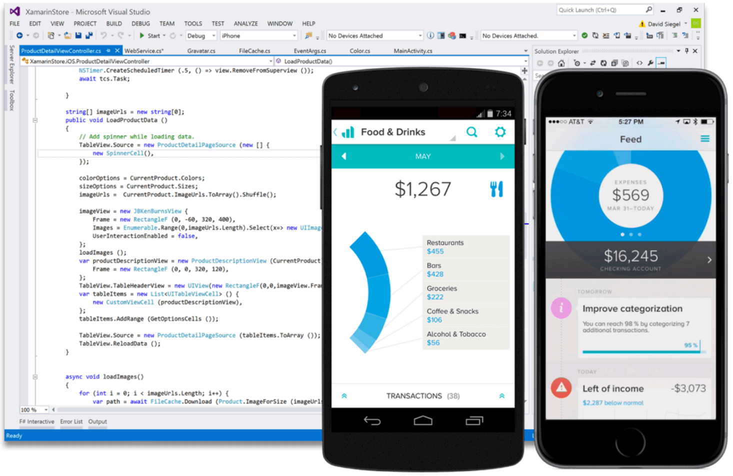
iOS

watchOS

The case for cross-platform development

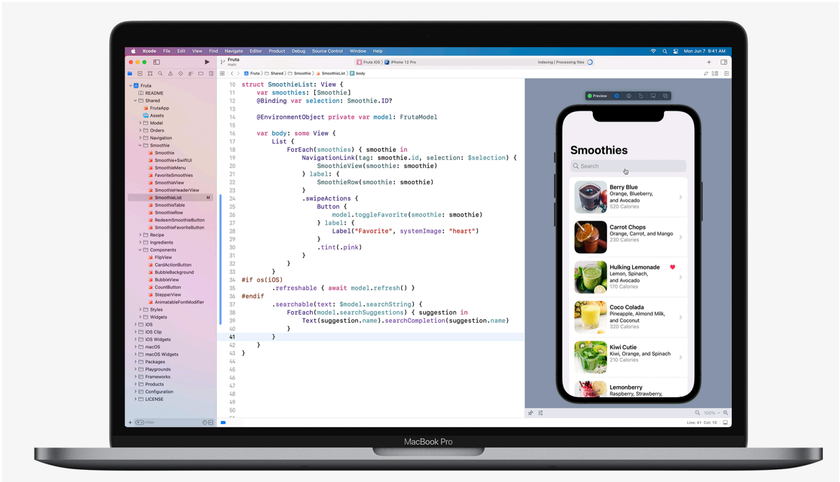
Developers have always been challenged to build for different platforms

- Differences in CPU architectures, operating systems limits software portability.
- Vendors produce libraries and tools for their platform only. Competitive not collaborative.
- Native tools and functionality will always be “the best available”.


XamarinStore - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Start Debug iPhone No Devices Attached
ProductDetailViewController.cs WebService.cs* Gravatar.cs FileCache.cs EventArgs.cs Color.cs MainActivity.cs
XamarinStore.iOS.ProductDetailViewController LoadProductData()
Food & Drinks
MAY
$1,267
Restaurants $455
Bars $428
Groceries $222
Coffee & Snacks $106
Alcohol & Tobacco $56
TRANSACTIONS (38)
Feed
EXPENSES $569
$16,245 CHECKING ACCOUNT
Improve categorization
Left of income -$3,073
Ready

```
            NSTimer.CreateScheduledTimer (.5, () => view.RemoveFromSuperview ());
            await tcs.Task;

        }

        string[] imageUrls = new string[0];
        public void LoadProductData ()
        {
            // Add spinner while loading data.
            TableView.Source = new ProductDetailPageSource (new [] {
                new SpinnerCell(),
            });

            colorOptions = CurrentProduct.Colors;
            sizeOptions = CurrentProduct.Sizes;
            imageUrls =  CurrentProduct.ImageUrls.ToArray().Shuffle();

            imageView = new JBKenBurnsView {
                Frame = new RectangleF (0, -60, 320, 400),
                Images = Enumerable.Range(0,imageUrls.Length).Select(x=> new UIImag
                UserInteractionEnabled = false,
            };
            loadImages ();
            var productDescriptionView = new ProductDescriptionView (CurrentProduct
                Frame = new RectangleF (0, 0, 320, 120),
            };
            TableView.TableHeaderView = new UIView(new RectangleF(0,0,imageView.Fra
            var tableItems = new List<UITableViewCell> () {
                new CustomViewCell (productDescriptionView),
            };
            tableItems.AddRange (GetOptionsCells ());

            TableView.Source = new ProductDetailPageSource (tableItems.ToArray ());
            TableView.ReloadData ();
        }


        async void loadImages()
        {
            for (int i = 0; i < imageUrls.Length; i++) {
                var path = await FileCache.Download (Product.ImageForSize (imageUrl
```

C# running in Visual Studio (Windows)



```
struct SmoothieList: View {
    var smoothies: [Smoothie]
    @Binding var selection: Smoothie.ID?

    @EnvironmentObject private var model: FrutaModel

    var body: some View {
        List {
            ForEach(smoothies) { smoothie in
                NavigationLink(tag: smoothie.id, selection: $selection) {
                    SmoothieView(smoothie: smoothie)
                } label: {
                    SmoothieRow(smoothie: smoothie)
                }
                .swipeActions {
                    Button {
                        model.toggleFavorite(smoothie: smoothie)
                    } label: {
                        Label("Favorite", systemImage: "heart")
                    }
                    .tint(.pink)
                }
            }
        }
#if os(iOS)
        .refreshable { await model.refresh() }
#endif
        .searchable(text: $model.searchString) {
            ForEach(model.searchSuggestions) { suggestion in
                Text(suggestion.name).searchCompletion(suggestion.name)
            }
        }
    }
}
```

Swift running in XCode (macOS)

What options do we have?

What are our choices as application developers?

1. **Limit ourselves to one platform**. e.g. Windows desktop, or iOS. This is bad for everyone.
2. **Target multiple platforms**, with separate projects for each. Cost and time prohibitive.
3. **Find ways to reuse our code**. This is ideal if we can do it!
    - Favour programming languages and libraries that are explicitly cross-platform: C++ not C#.
    - Recognize that some things are platform specific: UI, graphics especially.

Cross-platform toolkits!

Desktop toolkits

- **Java JDK**: Swing, JavaFX for user interfaces. Write against JVM using Java, Kotlin, Scala. ★
- **GTK, Qt, wxWidgets**. Write in C++, produce native code. Non-standard. ★

Mobile toolkits

- **PhoneGap**: Enabled writing mobile apps using HTML5, CSS3 and JavaScript.
- **Apache Cordova**: Open source fork of PhoneGap.
- **Appcelerator Titanium**: JavaScript-based SDK that supported iOS, Android, Windows and Blackberry.
- **Xamarin**: Microsoft-owned C#-based development framework that includes the .NET runtime.
- **React Native**: Based on the popular React web framework. Slow bridge between native and web. ★
- **Flutter**: Write your UI once and it will work on all platforms using native widgets. Relies on Dart. ★

The failure of cross-platform toolkits

- Vendors will always favour their own tools/libraries
  - Platform innovation is introduced by vendors in their native tools first.
  - Cross-platform is always playing catch-up.
- Cross-platform toolkits cannot achieve feature parity
  - Time + effort for cross-platform toolkits to fully support new platforms e.g. WatchOS.
  - Developers are restricted to capabilities of that platform. e.g. no hardware acceleration.
- Native will provide a “better experience” for users.
- I’m not convinced that this is the best *overall* strategy.

Cross-platform toolkits can still be useful! JavaFX is great for desktop, but it took 15 years to get here. It also won’t run on mobile.

Kotlin Multiplatform (KMP)

How is KMP different?

Kotlin Multiplatform (KMP) offers a unique solution to this problem, which provides the best of native + cross-platform.

- Produce Kotlin code which can be compiled across each platform.
- Recognize that sometimes you want native code instead, so provide interoperability with native libraries and native code.
- Generate native binaries that are a combination of Kotlin code and native code.

Linux (x86_64, arm32, arm64)	macOS (x86_64)
Windows (mingw x86_64, x86)	tvOS (arm64, x86_64)
Android (arm32, arm64, x86, x86_64)	watchOS (arm32, arm64, x86)
iOS (arm32, arm64, simulator x86_64)	WebAssembly (wasm32)

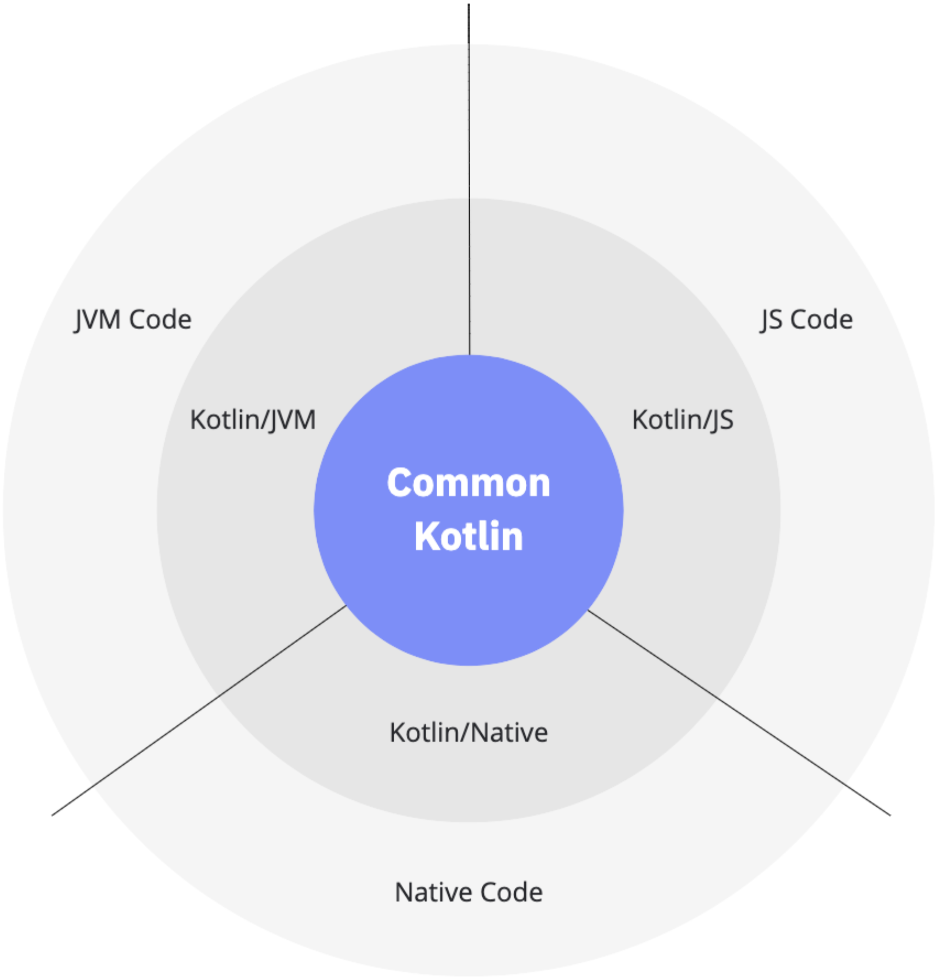
Supported native targets for KMP

Common Kotlin includes the language, core libraries, and basic tools.

- Code written in common Kotlin works on all supported platforms. Libraries cover everyday tasks such as HTTP, serialization, and managing coroutines.

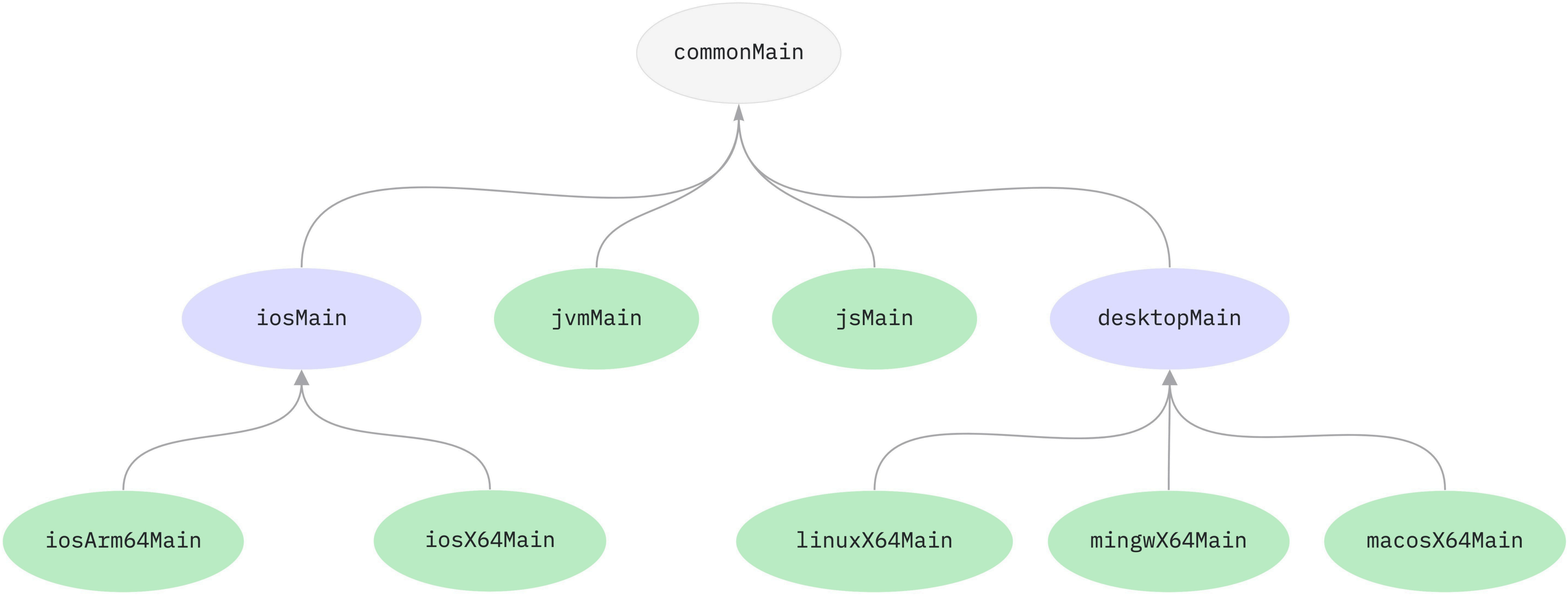
Kotlin also includes platform-specific versions of Kotlin libraries and tools (Kotlin/JVM, Kotlin/JS, Kotlin/Native).

- Access the platform native code (JVM, JS, and Native) and leverage all native capabilities.



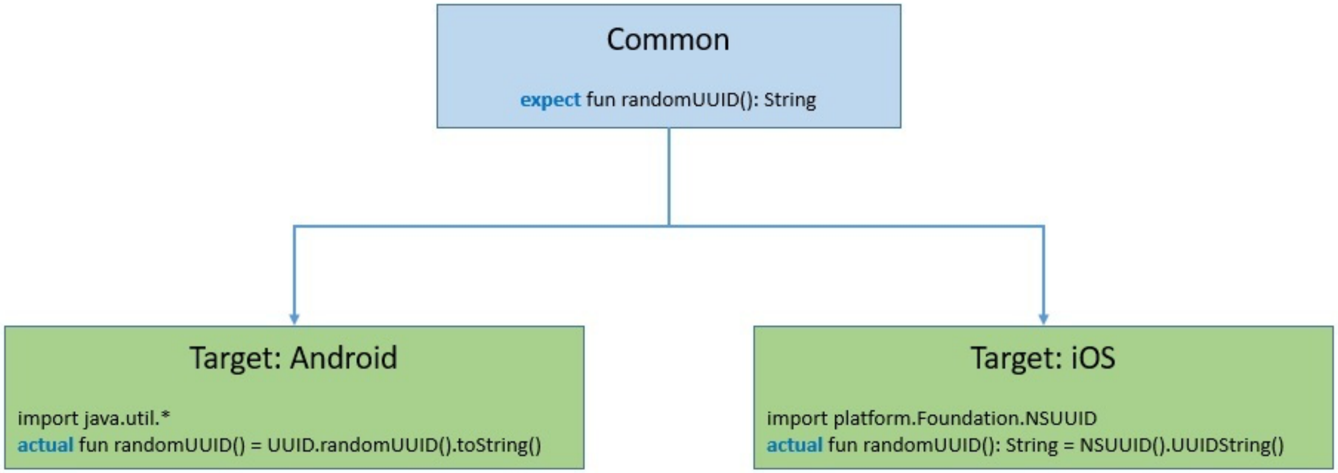
Multiplatform Projects

Kotlin multi-platform organizes the source code in hierarchies, with common-code at the base, and branches representing platform specific modules. All platform-specific source sets depend upon the common source set by default.



In some cases, it may be desirable to define and access platform-specific APIs in common. This is particularly useful for areas where certain common and reusable tasks are specialized for leveraging platform-specific capabilities.

Kotlin multi-platform uses **expected** to indicate a required function in common modules, and **actual** declarations in the platform specific modules.



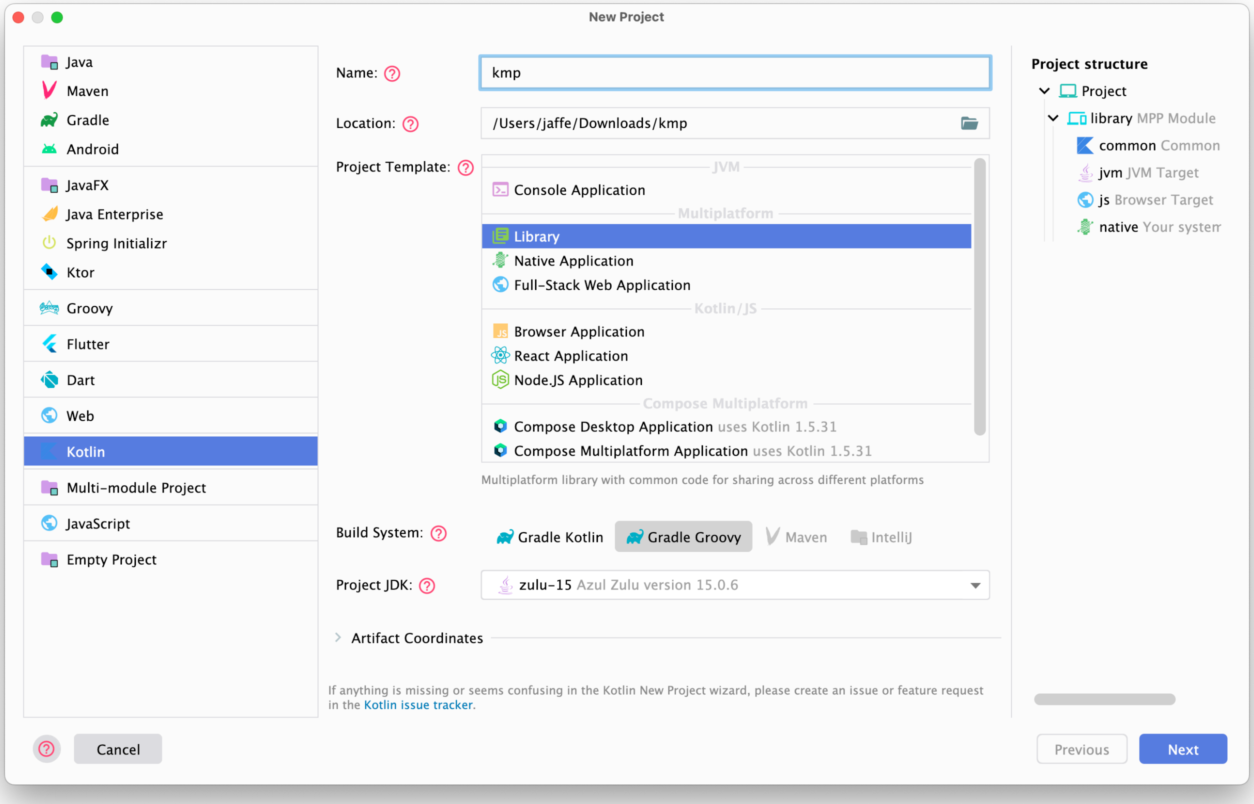
```
expect fun randomUUID(): String
```

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

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

Creating a project

In IntelliJ IDEA, select Kotlin - Multiplatform - Library.



New Project

Java
Maven
Gradle
Android
JavaFX
Java Enterprise
Spring Initializr
Ktor
Groovy
Flutter
Dart
Web
Kotlin
Multi-module Project
JavaScript
Empty Project

Name: kmp

Location: /Users/jaffe/Downloads/kmp

Project Template:

JVM
Console Application
Multiplatform
Library
Native Application
Full-Stack Web Application
Kotlin/JS
Browser Application
React Application
Node.JS Application
Compose Multiplatform
Compose Desktop Application uses Kotlin 1.5.31
Compose Multiplatform Application uses Kotlin 1.5.31

Multiplatform library with common code for sharing across different platforms

Build System: Gradle Kotlin | Gradle Groovy | Maven | IntelliJ

Project JDK: zulu-15 Azul Zulu version 15.0.6

Artifact Coordinates

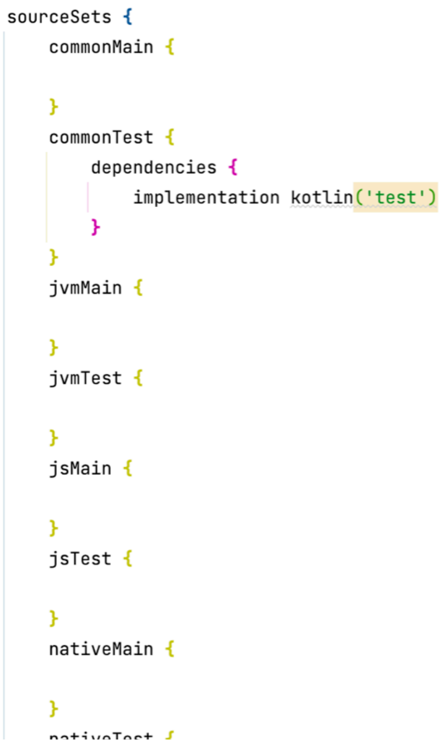
If anything is missing or seems confusing in the Kotlin New Project wizard, please create an issue or feature request in the Kotlin issue tracker.

Project structure
Project
library MPP Module
common Common
jvm JVM Target
js Browser Target
native Your system

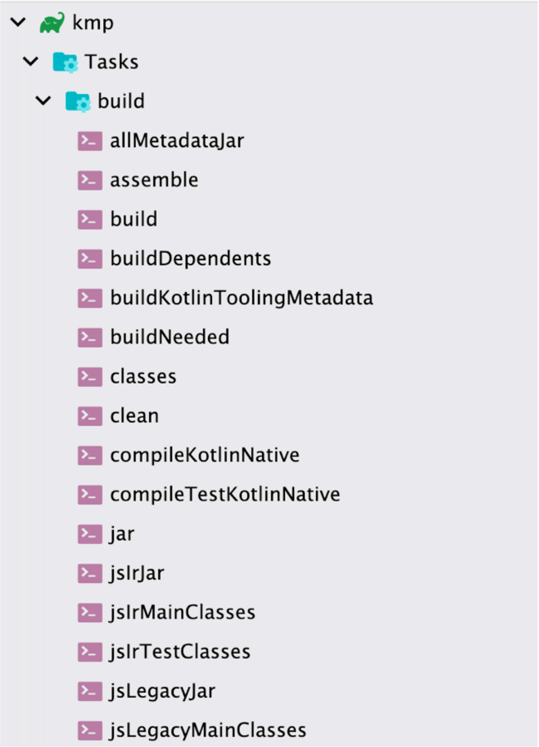
Cancel
Previous
Next

This generates a project with the kotlin-multiplatform Gradle plugin. This plugin is added to our `build.gradle` file.

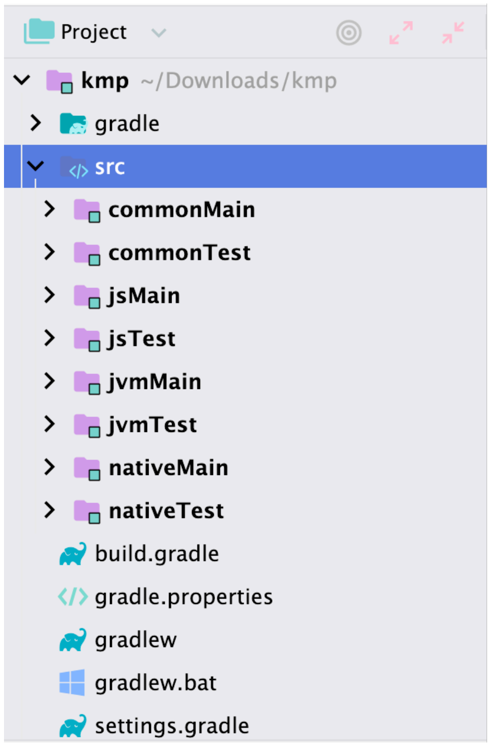
```
plugins {
    kotlin("multiplatform") version "1.4.0"
}
```



```
sourceSets {
    commonMain {

    }
    commonTest {
        dependencies {
            implementation kotlin('test')
        }
    }
    jvmMain {

    }
    jvmTest {

    }
    jsMain {

    }
    jsTest {

    }
    nativeMain {

    }
    nativeTest {
```



- kmp
  - Tasks
    - build
      - allMetadataJar
      - assemble
      - build
      - buildDependents
      - buildKotlinToolingMetadata
      - buildNeeded
      - classes
      - clean
      - compileKotlinNative
      - compileTestKotlinNative
      - jar
      - jsIrJar
      - jsIrMainClasses
      - jsIrTestClasses
      - jsLegacyJar
      - jsLegacyMainClasses



- kmp ~/Downloads/kmp
  - gradle
  - src
    - commonMain
    - commonTest
    - jsMain
    - jsTest
    - jvmMain
    - jvmTest
    - nativeMain
    - nativeTest
  - build.gradle
  - gradle.properties
  - gradlew
  - gradlew.bat
  - settings.gradle

Writing common code

```
fun add(num1: Double, num2: Double): Double {
    val sum = num1 + num2
    writeLogMessage("The sum of $num1 & $num2 is $sum", LogLevel.DEBUG)
    return sum
}

fun subtract(num1: Double, num2: Double): Double {
    val diff = num1 - num2
    writeLogMessage("The difference of $num1 & $num2 is $diff", LogLevel.DEBUG)
    return diff
}

fun multiply(num1: Double, num2: Double): Double {
    val product = num1 * num2
    writeLogMessage("The product of $num1 & $num2 is $product", LogLevel.DEBUG)
    return product
}

fun divide(num1: Double, num2: Double): Double {
    val division = num1 / num2
    writeLogMessage("The division of $num1 & $num2 is $division", LogLevel.DEBUG)
    return division
}
```

/commonMain

Note that it’s platform neutral code, that only uses Kotlin libraries.

Writing platform code

The `writeLogMessage()` function should be platform specific, since each OS will handle this differently. We will add a top-level declaration to our common code defining how that function should look:

```
enum class LogLevel {
    DEBUG, WARN, ERROR
}
internal expect fun writeLogMessage(message: String, logLevel: LogLevel)
```

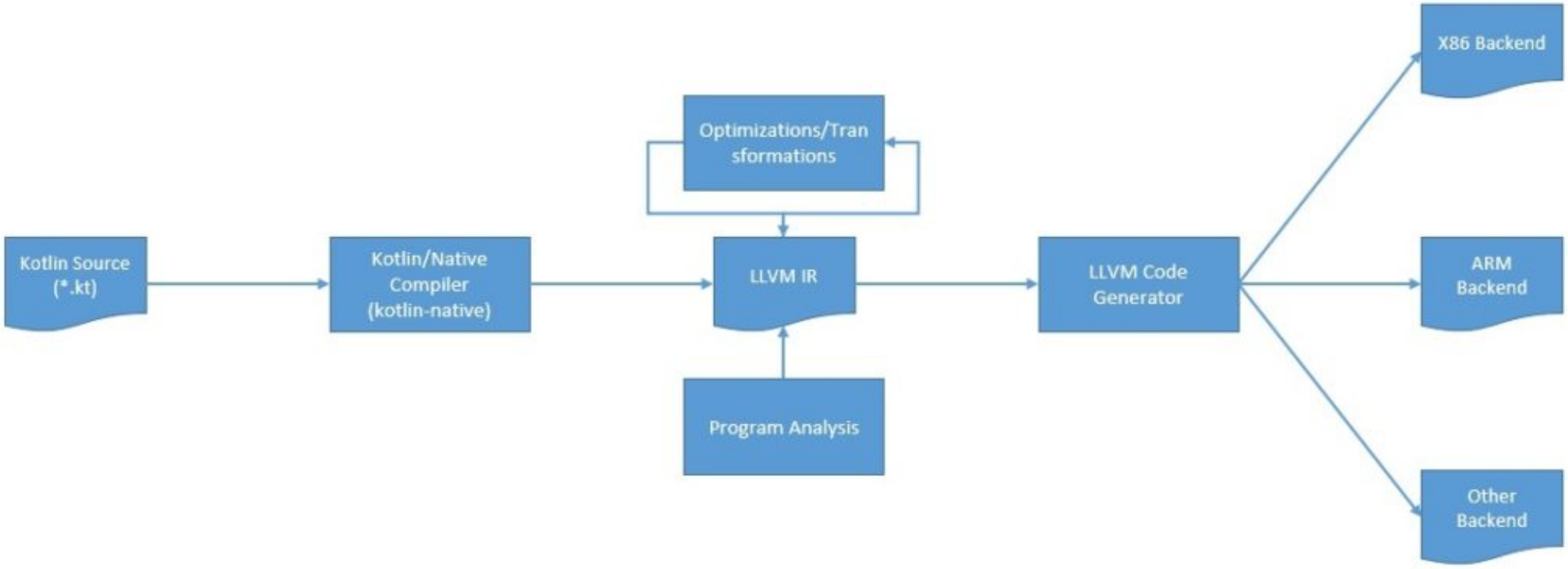
The `expect` keyword tells the compiler that the definition will be handled at the platform level, in another module. For example, we can flesh this out in the `jvmMain` module for Kotlin/JVM platform. The build for that platform will use the platform-specific version of this function.

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("Running in JVM: [$logLevel]: $message")
}
```

Kotlin/Native

Kotlin/Native is a subproject of KMP that is responsible for compiling the Kotlin source to native binaries specific to the target platform. Kotlin/Native provides an LLVM based backend for the Kotlin/Native compiler and native implementations of the Kotlin standard library. The Kotlin/Native compiler itself is known as `Konan`.

LLVM is a compiler infrastructure that we can use to develop a front end for any programming language and a back end for any instruction set architecture.



Kotlin/Native supports a number of platforms:

- Linux (x86_64, arm32, arm64, MIPS, MIPS little-endian)
- Windows (mingw x86_64, x86)
- Android (arm32, arm64, x86, x86_64)
- iOS (arm32, arm64, simulator x86_64)
- macOS (x86_64)4
- tvOS (arm64, x86_64)
- watchOS (arm32, arm64, x86)
- WebAssembly (wasm32)

In our Gradle configuration, there is a check for the host operating system. This determines what is built.

```
kotlin {
    val hostOs = System.getProperty("os.name")
    val isMingwX64 = hostOs.startsWith("Windows")
    val nativeTarget = when {
        hostOs == "Mac OS X" -> macosX64("native")
        hostOs == "Linux" -> linuxX64("native")
        isMingwX64 -> mingwX64("native")
        else -> throw GradleException("Host OS is not supported in Kotlin/Native.")
    }
}
```

Interoperability

Kotlin/Native supports two-way interoperability with native programming languages for different operating systems. The compiler creates:

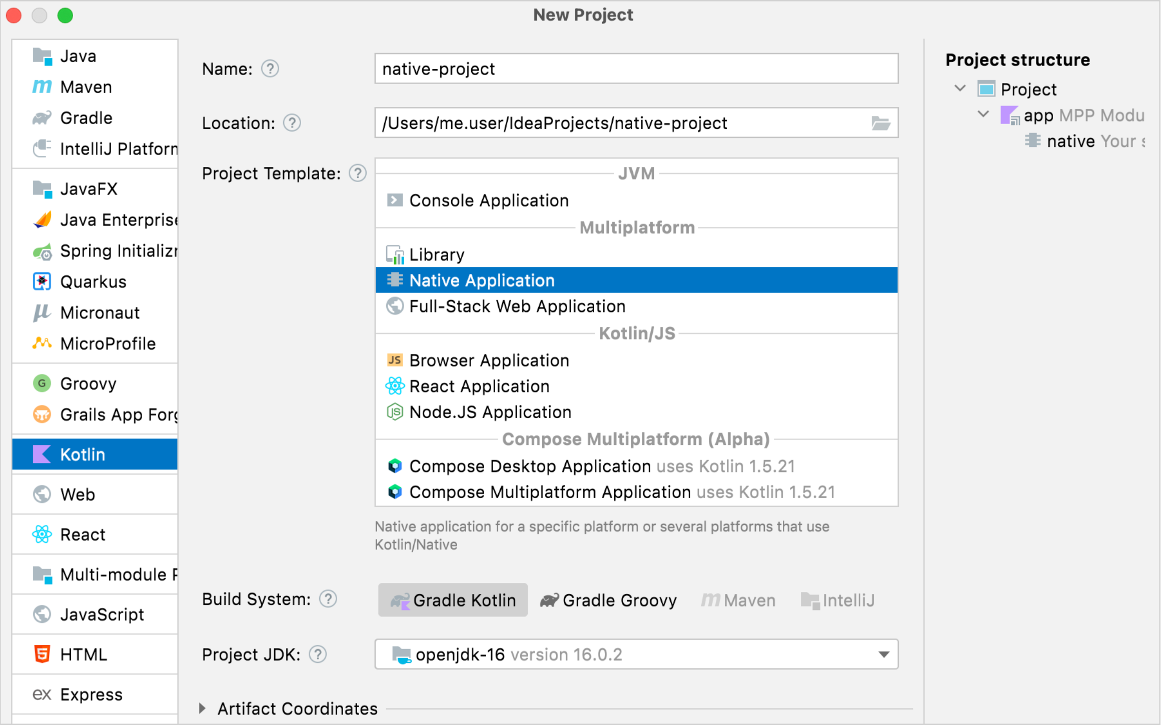
- an executable for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Kotlin/Native supports interoperability to use existing libraries directly from Kotlin/Native:

- static or dynamic C libraries
- C, Swift, and Objective-C frameworks

Platform libraries and framework typically support C interop.

Creating a native project



New Project

Name: native-project

Location: /Users/me.user/IdeaProjects/native-project

Project Template: Native Application

Native application for a specific platform or several platforms that use Kotlin/Native

Build System: Gradle Kotlin

Project JDK: openjdk-16 version 16.0.2

To create Kotlin/Native applications, you need the Kotlin Multiplatform plugin in your build.gradle file.

```
plugins {
    kotlin("multiplatform") version "1.6.10"
}
```

Build it. The native executable will be placed under:

```
build/bin/native/debugExecutable/
<your_app_name>.kexe
```

Yes that's all of it.

Example: Native/Interop

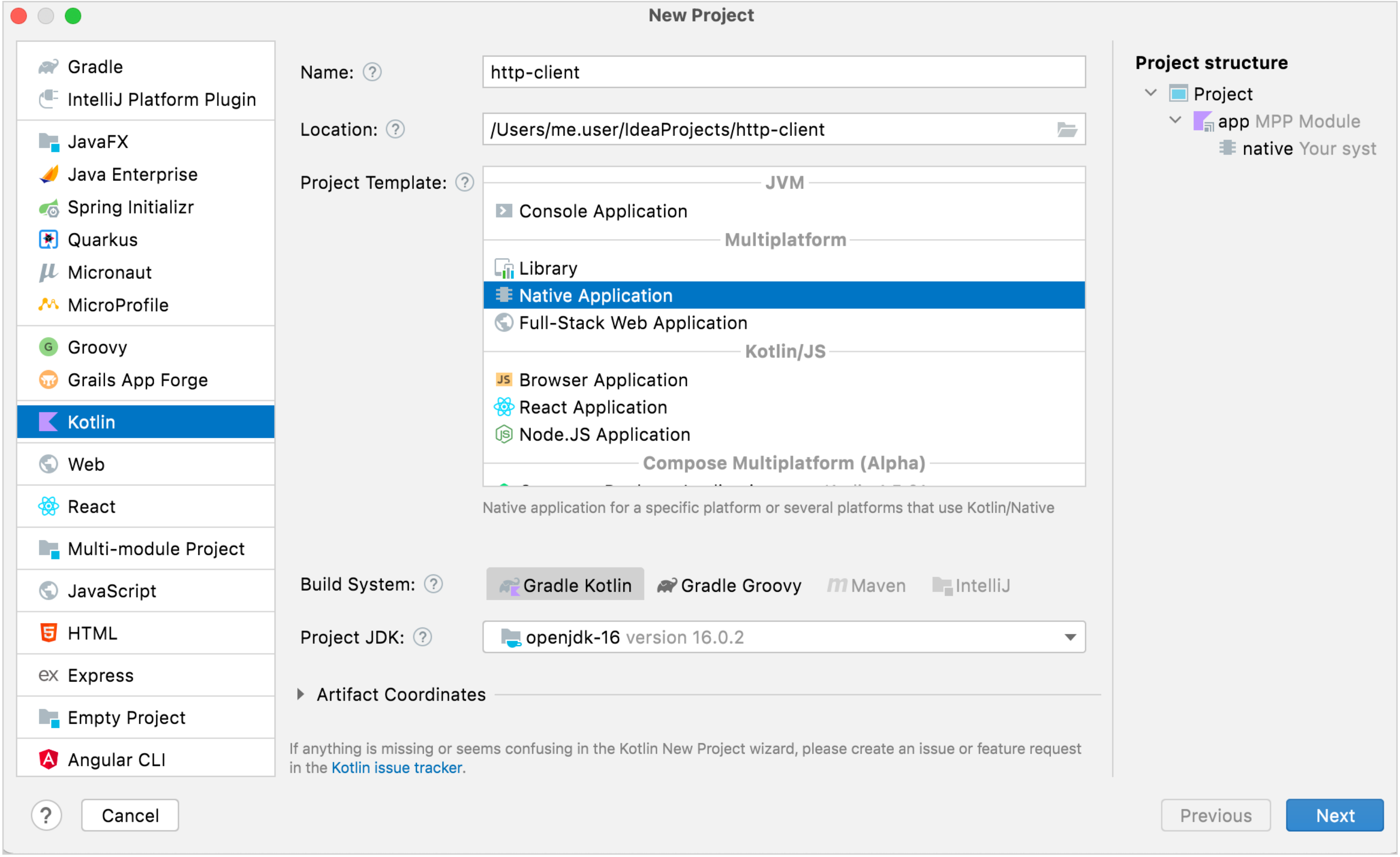
Building native apps is awesome, but the really interesting situation is when you start interoperating with native libraries.

This tutorial demonstrates how to use IntelliJ IDEA to create a simple HTTP client that can run natively on specified platforms using Kotlin/Native and the `libcurl` library.

This is taken from the Kotlin/Native samples:

- https://kotlinlang.org/docs/native-app-with-c-and-libcurl.html
- https://github.com/Kotlin/kotlin-hands-on-intro-kotlin-native

1. Create the project



New Project

Name: http-client

Location: /Users/me.user/IdeaProjects/http-client

Project Template: Native Application

Native application for a specific platform or several platforms that use Kotlin/Native

Build System: Gradle Kotlin

Project JDK: openjdk-16 version 16.0.2

If anything is missing or seems confusing in the Kotlin New Project wizard, please create an issue or feature request in the Kotlin issue tracker.

2. Update the build.gradle

```
kotlin {
    def hostOs = System.getProperty("os.name")
    def isMingwX64 = hostOs.startsWith("Windows")
    def nativeTarget
        if (hostOs == "Mac OS X") nativeTarget = macosX64('native')
        else if (hostOs == "Linux") nativeTarget = linuxX64("native")
        else if (isMingwX64) nativeTarget = mingwX64("native")
        else throw new FileNotFoundException("Host OS is not supported in Kotlin/Native.")

    nativeTarget.with {
        binaries {
            executable {
                entryPoint = 'main'
            }
        }
    }
}
```

3. Create a definition file

Kotlin/Native helps consume standard C libraries. We can link in a standard C library by describing the header and library location.

- Create a directory named `src/nativeInterop/cinterop`.
- Create a file `libcurl.def` with the following contents.

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

4. Add interoperability to your build

Add this to your build.gradle file.

```
nativeTarget.with {
    compilations.main { // NL
        cinterops {     // NL
            libcurl     // NL
        }               // NL
    }                   // NL
    binaries {
        executable {
            entryPoint = 'main'
        }
    }
```

5. Write application code

Update the source file `Main.kt` with the following source.

```
import kotlinx.cinterop.*
import libcurl.*

fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed ${curl_easy_strerror(res)?.toKString()}")
        }
        curl_easy_cleanup(curl)
    }
}
```

6. Compile and run it

```
$ ./httpclient.kexe
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
   ....
```

curl returns the contents of a URL i.e. a website.

Kotlin Multiplatform Mobile (KMM)

What is KMM?

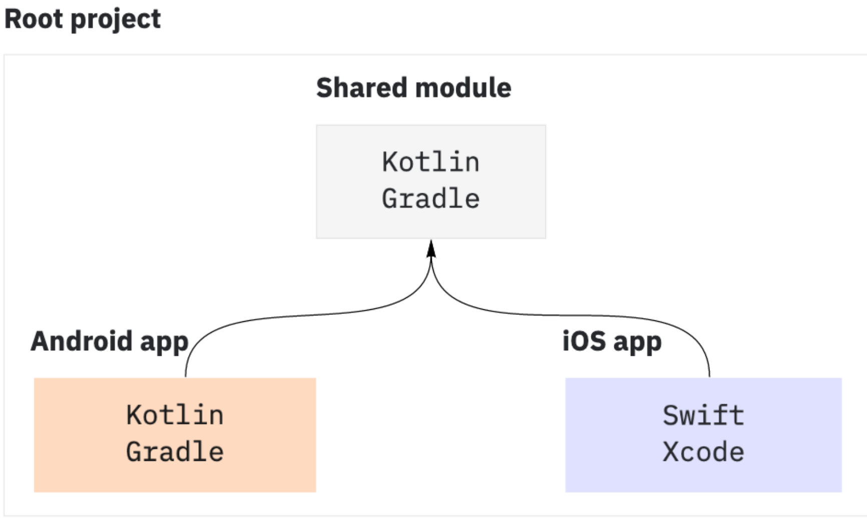
Kotlin Multiplatform Mobile (KMM) designed to build Android and iOS from the same project.

A basic KMM project consists of three components:

- **Shared module** – a Kotlin module that contains common logic for both Android and iOS applications. Builds into an Android library and an iOS framework.

- **Android application** – a Kotlin module that builds into the Android application. Uses Gradle as a build system.

- **iOS application** – an Xcode project that builds into the iOS application. Uses CocoaPods for builds.



Kotlin supports two-way interop with iOS: Kotlin can call into iOS libraries, and vice-versa using the Objective-C bindings. (Swift bindings are being developed). In order to use KMM you need to be using a Mac, and have the Xcode toolchain installed.

KMM is exciting because we can use Kotlin for both targets, and share probably 50-75% of the code between platforms.

A KMM application could potentially offer identical functionality on Android and iOS, while delivering a completely native UI experience with Jetpack Compose on Android, and SwiftUI on iOS.

See the list of KMM Samples. https://kotlinlang.org/docs/multiplatform-mobile-samples.html

Reference

Kumar Chandrakant. 2021. Introduction to Multiplatform Programming in Kotlin. https://www.baeldung.com/kotlin/multiplatform-programming

JetBrains. 2022. Kotlin Native. https://kotlinlang.org/docs/native-overview.html

JetBrains. 2022. Kotlin Multiplatform. https://kotlinlang.org/docs/multiplatform-get-started.html

JetBrains. 2022. Kotlin Multiplatform Mobile. https://kotlinlang.org/docs/multiplatform-mobile-getting-started.html

Carlos Mota, Saeed Taheri and Kevin D Moore. 2022. Kotlin Multiplatform by Tutorials. https://www.raywenderlich.com/books/kotlin-multiplatform-by-tutorials