

WYKŁAD 1

Wprowadzenie do Programowania Aplikacji Mobilnych

Przedmiot: Programowanie Aplikacji Mobilnych

Studia Inżynierskie / Informatyka

Android · iOS · Flutter · React Native · Kotlin · Swift

dr inż. Mateusz Pomianek

D.103 | m.pomianek@prz.edu.pl

mpomianek.v.prz.edu.pl

Ekosystem mobilny

Android Studio + Kotlin

Xcode + Swift

Flutter / React Native

Pierwsza aplikacja → Hello World

Plan wykładów

- | | |
|---|-------------------------------------|
| 1. Wykład – Wprowadzenie do programowania aplikacji mobilnych | 1. 2026-03-04 08:45 : 10:15 |
| 2. Wykład – Projektowanie aplikacji mobilnych | 2. 2026-03-11 08:45 : 10:15 |
| 3. Wykład – Architektura i budowa urządzeń mobilnych | 3. 2026-03-18 08:45 : 10:15 |
| 4. Wykład – Metody interakcji i projektowanie UI/UX | 4. 2026-03-25 08:45 : 10:15 |
| 5. Wykład – Programowanie natywne (Android Studio i Xcode) | 5. 2026-04-08 08:45 : 10:15 |
| 6. Wykład – Programowanie cross-platformowe i PWA | 6. 2026-04-15 08:45 : 10:15 |
| 7. Wykład – Obsługa sensorów urządzenia mobilnego | 7. 2026-04-22 08:45 : 10:15 |
| 8. Wykład – Programowanie aplikacji mobilnych IoT | 8. 2026-04-29 08:45 : 10:15 |
| 9. Wykład – Informatyka Afektywna w aplikacjach mobilnych | 9. 2026-05-06 08:45 : 10:15 |
| 10. Wykład – Programowanie aplikacji mobilnych XR | 10. 2026-05-13 08:45 : 10:15 |
| 11. Wykład – Programowanie gier mobilnych | 11. 2026-05-20 08:45 : 10:15 |
| 12. Wykład – Programowanie autonomicznych robotów | 12. 2026-05-27 08:45 : 10:15 |
| 13. Egzamin | 13. 2026-06-03 08:45 : 09:30 |

Plan wykładu

Sl. 1-3

Ekosystem mobilny — rynek, platformy, liczby

Sl. 6-7

Narzędzia — Android Studio, Xcode, emulatory

Sl. 10-11

Swift — podstawy języka dla iOS

Sl. 14-15

Pierwsza aplikacja iOS — Hello World

Sl. 18-19

Układy UI — XML Layouts i SwiftUI

Sl. 22-23

Wzorce architektoniczne — MVC, MVVM, MVP

Sl. 26-27

Testowanie aplikacji mobilnych

Sl. 30

Projekt semestralny i wymagania

Sl. 4-5

Android vs iOS — architektura i filozofia

Sl. 8-9

Kotlin — podstawy języka dla Android

Sl. 12-13

Pierwsza aplikacja Android — Hello World

Sl. 16-17

Cykl życia Activity i ViewController

Sl. 20-21

Cross-platform — Flutter i React Native

Sl. 24-25

Dystrybucja — Google Play i App Store

Sl. 28-29

Kariera i rynek pracy — developer mobilny

Ekosystem mobilny: liczby, platformy i trendy 2025

Aplikacje mobilne to największy kanał dystrybucji oprogramowania na świecie
—> ponad 9 miliardów aktywnych urządzeń, 250 miliardów pobrań rocznie.

9.1 mld

Aktywnych
urządzeń mobilnych

257 mld

Pobrań aplikacji
rocznie (2024)

\$935 mld

Przychody
App Economy 2025

72%

Ruchu internetowego
pochodzi z mobile

Podział rynku i kategorie aplikacji

Android (Google)

71.5%

Open platform. Google Play: 3.5M aplikacji. AOSP + OEM skinning (Samsung One UI, Xiaomi MIUI). Fragmentacja urządzeń — wyzwanie.

iOS (Apple)

27.5%

Zamknięty ekosystem. App Store: 1.8M aplikacji. Wyższy ARPU (Average Revenue Per User). Polityka prywatności ATT. Silniejsza monetyzacja.

Gry mobilne

52% rev

Największa kategoria przychodów. Free-to-play + in-app purchases. Unity / Unreal Engine. Rynek \$100+ mld/rok.

Social / Komunikatory

40% DL

Najwięcej pobrań. TikTok, Instagram, WhatsApp. Retencja: 90-dniowy retention rate < 5%. Short-form video dominuje.

FinTech / mBanking

\$850 m lrd

Najszybciej rosnący sektor. BLIK, Revolut, Klarna. Wymogi regulacyjne PCI-DSS, DORA (EU), PSD2. Biometria jako standard.

Health & Fitness

+25% YoY

Post-COVID boom. Apple Health, Google Fit integration. HealthKit / Health Connect API. Regulatory: FDA SaMD (Software as Medical Device).

Android vs iOS: filozofia, architektura i wybór platformy

Dwie dominujące platformy — różna filozofia projektowania, inne narzędzia, inne języki. Programista mobilny musi znać obie lub świadomie wybrać jedną.

Kryterium	Android	iOS
Twórca	Google (AOSP) + OEM	Apple Inc.
Język (główny)	Kotlin (+ Java legacy)	Swift (+ Objective-C legacy)
IDE	Android Studio (JetBrains)	Xcode (macOS only)
UI Framework	Jetpack Compose / XML Views	SwiftUI / UIKit
Dystrybucja	Google Play + sideloading	App Store (tylko)
Review czas	kilka godzin–1 dzień	1–3 dni (review team)
Opłata	\$25 jednorazowo	\$99/rok
Fragmentacja	Ogromna (14 000+ urządzeń)	Minimalna (~20 modeli)
Rynkowy udział	71.5% globalnie	27.5% (60%+ USA/Zachodnia EU)
ARPU	Niższy (\$0.43 avg purchase)	Wyższy (\$1.08 avg purchase)

Dla nowych: zacznij od Androida (Android Studio wolne, ale Windows-friendly). iOS wymaga Maca i płatnego konta deweloperskiego Apple.

Ekosystem narzędzi — IDE, SDK, Emulatory i Debugowanie

Narzędzia deweloperskie to fundament produktywności. Dobry setup środowiska to 30% sukcesu projektu. Skonfiguruj raz — pracuj efektywnie.

Android — stos narzędzi:

Android Studio Meerkat (2025)

IntelliJ IDEA-based. Gemini AI integration. Live Edit (Compose). Device Streaming (fizyczne zdalne). Flamegraph profiler. Darmowy, open-source IDE.

Android SDK + Build Tools

Gradle 8.x jako build system. AGP (Android Gradle Plugin) 8.x. compileSdk 35 (Android 15). minSdk 24 (Android 7.0 = 95% urządzeń).

AVD Manager (Emulator)

Android Virtual Device. HAXM/WHPX akceleracja sprzętowa. API Level 35 emulacja. Obsługa Pixel 9 Pro skórek. Apple Silicon: ARM images natywnie.

ADB (Android Debug Bridge)

Komunikacja PC ↔ urządzenie. adb install/uninstall APK. adb logcat (logi). adb shell (terminal). adb forward (port tunneling). WiFi debugging.

Profiler + Lint

Memory profiler (heap dump, allocation). CPU profiler (tracing, sampling). Network profiler. Lint: ~400 reguł statycznej analizy kodu.

iOS — stos narzędzi:

Xcode 16 (macOS only)

Apple IDE zintegrowany z SDKiem. Swift Package Manager wbudowany. Simulator wieloplatformowy. Preview (SwiftUI live). Instruments profiler. Wymaga macOS 14+.

iOS SDK + Simulator

Simulator wieloplatformowy: iPhone 16, iPad, Watch. Drag&drop instalacja .app. Accessibility Inspector. Network Link Conditioner (throttling).

Swift Package Manager (SPM)

Oficjalny menedżer pakietów Swift. Package.swift manifest. Zastępuje CocoaPods i Carthage. Xcode 15+ integracja natywna w project settings.

Instruments

Time Profiler (CPU). Allocations (memory). Leaks detector. Core Data Fetches. Core Animation (60fps). Analogi do Android Profiler. Niezbędny.

TestFlight + Xcode Cloud

TestFlight: beta distribution (10 000 testerów). Xcode Cloud: CI/CD natywnie w Xcode. Test automatyczny przed review Apple. Darmowe 25h/mies.

Kotlin: Oficjalny język Androida (od 2017)

Kotlin to statycznie typowany, nowoczesny język JVM. Google ogłosiło 'Kotlin-first' w 2019.
Dziś 95%+ nowego kodu Android pisze się w Kotlinie.

Najważniejsze cechy

```
// 1. NULL SAFETY — eliminacja NullPointerException
val surname: String? = null
val len = surname?.length ?: 0

// 2. DATA CLASSES — boilerplate gone
data class User(val id: Int, val name: String)
val u2 = User(1, "Anna").copy(name = "Bartek")

// 3. EXTENSION & LAMBIDAS
fun String.isPal() = this == reversed()
val doubledEvens = listOf(1,2,3,4,5).filter { it%2==0 }.map { it*2 }

// 4. COROUTINES — asynchroniczność bez callback hell
suspend fun fetchUser(id: Int) =
    withContext(Dispatchers.IO) { api.getUser(id) }

// 5. SMART CASTS
fun describe(x: Any) = when (x) {
    is Int -> "Liczba: $x"
    is String -> "Tekst: ${x.length}"
    else -> "Nieznany"
}
```

Kotlin

Kotlin vs Java — dlaczego Kotlin wygrał

Boilerplate

Kotlin: data class, val/var

Java: POJO + getters/setters (10× więcej kodu)

Null safety

Kotlin: Nullable typy (?), Elvis (?:)

Java: NullPointerException — 'Billion Dollar Mistake'

Coroutines

Kotlin: suspend fun, launch, async

Java: Threads, ExecutorService, CompletableFuture

Interop

Kotlin: 100% kompatybilny z Java

Java: Brak Kotlin interop (odwrotna strona)

Expressiveness

Kotlin: Lambdy, extension, DSL

Java: Verbose, ceremonialny

Nowoczesna asynchroniczność: Kotlin Coroutines i Flow

Coroutines to lekkie kooperatywne wątki Kotlinia. Flow to reaktywne strumienie danych. Razem zastępują RxJava/callbacks i są podstawą współczesnego Androida.

```
// ViewModel + StateFlow + IO Kotlin
class NewsVM(private val repo: Repo) : ViewModel() {
    private val _a = MutableStateFlow<List<Article>>(emptyList())
    val a: StateFlow<List<Article>> = _a
    fun load() = viewModelScope.launch {
        _a.value = withContext(Dispatchers.IO) { repo.fetchNews() }
    }
}
```

```
// Flow search (debounce + cancel) Kotlin
fun search(q: StateFlow<String>) =
    q.debounce(300).filter { it.length > 2 }.flatMapLatest(repo::searchUsers)

// Collect w UI
lifecycleScope.launch {
    viewModel.users.collect(adapter::submitList)
}
```

Dispatchers i structured concurrency

Dispatchers.Main

Wątek UI Androida. Aktualizacja widoków, animacje. Blokowanie $\geq 16\text{ms}$ = dropped frame ($< 60\text{fps}$). Użyj tylko do UI operations.

Dispatchers.IO

Puła 64 wątków (lub więcej). Operacje blokujące: sieć, baza danych Room, zapis pliku. Nie używaj do CPU-intensive pracy.

Dispatchers.Default

Puła = liczba CPU cores. CPU-intensive: sortowanie, parsing JSON, kompresja. Domyślny dla launch i async bez parametru.

Dispatchers.Unconfined

Nie zalecany w produkcji. Wykonanie na bieżącym wątku \rightarrow po suspend: kontynuacja gdzie coroutine wznowiona. Debug/test.

Zasada: withContext(IO) dla IO, withContext(Default) dla CPU. Nigdy nie blokuj Dispatchers.Main (Thread.sleep \rightarrow ANR po 5 sekundach).

Swift — Język Apple dla iOS, macOS, watchOS i visionOS

Swift to szybki, bezpieczny język open-source (2014). Zastąpił Objective-C. W 2023 Apple ogłosiło Swift jako język przyszłości dla iOS, macOS, server-side, embedded.

Swift — kluczowe cechy języka

```
// 1. OPTIONALS — bezpieczeństwo nil
let upper = username?.uppercased()
let name = username ?? "Gość"
guard let user = username else { return }
```

Swift

```
// 3. PROTOCOLS — duck typing
protocol Drawable { func draw() }
struct Circle: Drawable {
    func draw() { print(" ") }
}
```

```
// 4. GENERICS + where clauses
func largest<T: Comparable>(_ arr: [T]) -> T? {
    arr.max()
}
```

```
// 5. ASYNC/AWAIT (Swift 5.5+)
func fetchProfile(id: UUID) async throws -> Profile {
    let url = URL(string: "https://api.example.com/users/^(id)")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(Profile.self, from: data)
}
```

```
// 6. CLOSURES (odpowiednik lambda)
let sorted = ["banana", "apple", "cherry"].sorted { $0 < $1 }
let mapped = [1,2,3].map { $0 * $0 } // [1, 4, 9]
```

Swift vs Objective-C

Składnia

Swift: Nowoczesna, czytelna (jak Python/Kotlin)

ObjC: Square bracket hell: [obj message:arg]

Safety

Swift: Optionals, generics, ARC nowoczesny

ObjC: nil sends messages silently — ukryte błędy

Performance

Swift: Porównywalne z C++ (Swift benchmark 2.6x)

ObjC: Dynamiczny dispatch — overhead

Interop

Swift: Bridging header dla ObjC bibliotek

ObjC: Używany w starszych bibliotekach (UIKit core)

Ecosystem

Swift: SPM, open-source (Linux, Windows, WASM)

ObjC: Tylko Apple platforms

Pierwsza aplikacja Android: Struktura projektu i Hello Mobile World

Struktura projektu Android

```
MyApp/
├── app/
│   ├── src/main/
│   │   ├── java/com/example/myapp/
│   │   │   └── MainActivity.kt ← główna Activity
│   │   ├── res/
│   │   │   ├── layout/
│   │   │   │   └── activity_main.xml ← układ UI (XML)
│   │   │   ├── values/
│   │   │   │   ├── strings.xml ← teksty (i18n)
│   │   │   │   └── themes.xml ← style Material
│   │   │   └── drawable/ ← obrazy, ikony
│   │   └── AndroidManifest.xml ← deklaracja app
│   ├── build.gradle.kts ← zależności modułu
├── gradle/libs.versions.toml ← Version Catalog
└── settings.gradle.kts
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(b: Bundle?) {
        super.onCreate(b)
        enableEdgeToEdge()
        setContent { MyAppTheme { Greeting("Programisto!") } }
    }
}
```

Tworzenie projektu: **File** → **New** → **New Project** → **Empty Activity**
Gradle synchronizuje zależności. Pierwsza kompilacja: 2-5 minut (cierpliwości!).

AndroidManifest.xml: deklaracja aplikacji

```
<!-- AndroidManifest.xml -->
```

build.gradle.kts (Module)

```
dependencies { implementation libs.androidx.core.ktx; implementation
libs.compose.ui; implementation libs.lifecycle.viewmodel.ktx } — Gradle
Version Catalog (.toml) to nowy standard.
```

Uruchamianie aplikacji

Run → Run 'app' (Shift+F10). Wybierz AVD lub fizyczne urządzenie (USB).
First build: ~2min. Instant Run / Live Edit dla Compose: sekundy.

Pierwsza aplikacja iOS: Xcode, SwiftUI i Hello World

Struktura projektu Xcode

```
MyApp.xcodeproj (lub .xcworkspace) Shell
├─ MyApp/
│  └─ MyApp.swift ← App entry point (@main)
│  └─ ContentView.swift ← główny widok SwiftUI
│  └─ Assets.xcassets/ ← obrazy, ikony, kolory
│  └─ Info.plist ← konfiguracja aplikacji
│  └─ Preview Content/ ← dane podglądu Preview
```

```
# Swift Package Dependencies (SPM):
# Xcode → File → Add Package Dependencies
# Wymaga URL repozytorium (GitHub / GitLab)
# Package.resolved = lock file wersji
```

```
// ContentView.swift — Hello World SwiftUI Swift
struct ContentView: View {
    @State private var name = "Programisto"
    var body: some View {
        VStack {
            Text("Witaj, \(name)!").font(.largeTitle).bold()
            TextField("Twoje imię", text:
$name).textFieldStyle(.roundedBorder)
        }.padding()
    }
}
```

File → New → Project → App (SwiftUI + Swift)

Symulator uruchamia się automatycznie.

Xcode Preview pozwala widzieć zmiany w czasie rzeczywistym bez pełnej kompilacji.

App entry point i App Delegate

```
@main Swift
struct MyApp: App {
    var body: some Scene { WindowGroup { ContentView() } }
}
```

SwiftUI Preview

#Preview automatycznie renderuje widok w Xcode Canvas. Można testować różne stany, urządzenia, dark/light mode. Nie wymaga Simulatora. Wymaga macOS 14+ i Xcode 15+.

Wymagania Apple Developer

Darmowe: budowanie na własny iPhone. \$99/rok: dystrybucja App Store, TestFlight external, podpisywanie certyfikatami. Team: 5 wolnych slotów na urządzenie.

Android Lifecycle: cykl życia Activity, metody i kiedy je wywołać

onCreate()

Inicjalizacja: inflate UI, setup ViewModel, bind data. Wywoływana raz (lub po procesowym restorowaniu). Bundle? = zapisany stan.

onStart()

Activity widoczna, ale nie w centrum uwagi. Rare use case. Poprzedza onResume.

onResume()

Activity w foreground — user może wchodzić w interakcje. Start: animacje, kamera, sensory, pauza muzyki.

onPause()

Inna Activity częściowo przykrywa. Stop: animacje, sensory (bateria!). Krótki callback — nie rób IO tutaj.

onStop()

Activity niewidoczna. Stop: ciężkie zasoby, sieć. SaveState dla UI. Room może zapisać dane tutaj.

onDestroy()

Activity niszczone (Back lub finish()). Wyczyść: ViewBinding null, unregister. ViewModel NIE jest niszczone przy rotate!

onSaveInstanceState()

Zapisz UI state: tekst, scroll position. Wywoływane przed onStop(). Bundle → Parcelable lub Serializable.

Cykl życia to jeden z najtrudniejszych konceptów Androida. Błędne zarządzanie stanem → memory leaks, niepotrzebne requesty sieciowe, utrata danych użytkownika.

```
Kotlin
// Android lifecycle + repeatOnLifecycle
class MainActivity : AppCompatActivity() {
    private val vm: MainVM by viewModels()
    private var _b: ActivityMainBinding? = null
    private val b get() = _b!!
    override fun onCreate(s: Bundle?) {
        super.onCreate(s); _b =
        ActivityMainBinding.inflate(layoutInflater)
        setContentView(b.root)
        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) { vm.uiState.collect
            { b.textView.text = it.message } }
        }
    }
    override fun onDestroy() { _b = null; super.onDestroy() }
}
```

repeatOnLifecycle(STARTED) automatycznie anuluje kolekcję gdy app idzie w tło.
Stara metoda lifecycleScope.launch + collect powoduje zbędne aktualizacje w tle.

iOS App Lifecycle: Cykl życia ViewController i Scene

iOS używa UIViewController lifecycle (UIKit) lub View onAppear/onDisappear (SwiftUI).

Zrozumienie kiedy ładować dane i zwalniać zasoby jest kluczowe dla performance.

UIViewController lifecycle:

viewDidLoad()

Raz po załadowaniu widoku z nib/storyboard/code. Setup UI, bind data, configure. NIE tutaj: dane z sieci (może być zbyt wcześnie).

viewWillAppear(_:)

Przed każdym pojawieniem. Odśwież UI, start sensory, update navigation bar.

viewDidAppear(_:)

Widok widoczny. Start animacje, odtwarzanie video, network requests (user widzi).

viewWillDisappear(_:)

Przed zniknięciem. Zatrzymaj animacje, zapisz state.

viewDidDisappear(_:)

Widok zniknął. Stop sensory, heavy resources, invalidate timers.

deinit

VC z ARC dealokowany. Usuń obserwatorów (NotificationCenter). Anuluj Task (async).

SwiftUI — onAppear, onDisappear, task

```
// UIKit (minimum) Swift
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    Task { await vm.load() }
}
deinit { task?.cancel() }
```

```
// esencja: task anulowany
List(vm.articles) { ArticleRow(article: $0) }
.task { await vm.loadArticles() }
.refreshable { await vm.reload() }
```

UIKit vs SwiftUI (kiedy co)

UIKit: legacy apps, skomplikowane custom views, UICollectionView z DiffableDataSource. SwiftUI: nowe projekty iOS 16+, cross-platform (Mac, Watch, TV, Vision). Mix: UIHostingController / UIViewRepresentable.

Scene lifecycle (iOS 13+)

SceneDelegate.sceneDidBecomeActive / sceneWillResignActive — wielookienkowe iPad apps. SwiftUI: @Environment(\.scenePhase).

Jetpack Compose: nowoczesny UI Android

Jetpack Compose (2021) to deklaratywny framework UI Androida
— koniec z XML layouts i View inflation.
UI opisujesz jako funkcje Kotlin reagujące na stan.

Compose

```
// UserCard Kotlin
@Composable
fun UserCard(u: User, onClick: () -> Unit) {
    Card(onClick = onClick, modifier =
    Modifier.fillMaxWidth().padding(8.dp)) {
        Row(Modifier.padding(16.dp), verticalAlignment =
        Alignment.CenterVertically) {
            Text(u.name, style = MaterialTheme.typography.titleMedium)
        }
    }
}
```

```
// State hoisting SearchBar Kotlin
@Composable
fun SearchBar(q: String, onQ: (String) -> Unit) =
    OutlinedTextField(q, onQ, label = { Text("Szukaj...") }, modifier =
    Modifier.fillMaxWidth().padding(16.dp))
```

Compose vs XML Views

Deklaratywność

Compose: UI = f(state). Zmień state → UI się odświeży automatycznie.

XML: Imperatywne: findViewById().setText(). Synchronizacja ręczna.

Rekomponowanie

Compose: Inteligentne: tylko zmieniające się Composable odświeżone (skip).

XML: Cały layout musi być ręcznie zarządzany.

Modifiers

Compose: Modifier.fillMaxWidth().padding().clickable() — chain API.

XML: layout_width, paddingStart, android:onClick — XML atrybuty.

Testy

Compose: composeTestRule.onNodeWithText() — semantyki UI.

XML: Espresso: onView(withId(R.id.button)).perform(click()).

Interop

Compose: ComposeView w XML, AndroidView w Compose — pełna interoperacyjność.

XML: Praca z Maps, Custom Views, CameraX — wciąż relevantny.

SwiftUI: Deklaratywny UI dla całego ekosystemu Apple

Property Wrappers, czyli zarządzanie stanem:

@State

Lokalny stan Composable widoku. Zmiana wartości → rerenderowanie. Tylko typ wartości (struct). Przykład: @State var count = 0.

@Binding

Dwukierunkowe połączenie z stanem rodzica. Przekazujesz \$state do dziecka. Dziecko może modyfikować. Przykład: @Binding var isPresented: Bool.

@StateObject

ObservableObject tworzony i posiadany przez widok. ViewModel. Żyje z widokiem. Różnica vs @ObservedObject: nie niszczy przy re-renderze rodzica.

@ObservedObject

ObservableObject z zewnątrz widoku. Widok nie jest właścicielem. Może być nil/zresetowany. Dependency injection w SwiftUI.

@EnvironmentObject

Dependency injection bez przekazywania przez parametry. .environmentObject(vm) w rodzicu. Dostępny wszędzie w hierarchii. Uważaj na crashes.

SwiftUI (2019) działa na iOS, macOS, watchOS, tvOS i visionOS — jeden kod na 5 platform. Deklaratywna składnia, **property wrappers** jako mechanizm stanu.

SwiftUI: lista, navigation i Combine

```
struct ArticleListView: View {  
    @StateObject var vm = ArticleVM()  
    var body: some View {  
        NavigationStack {  
            List(vm.articles) { a in NavigationLink(a.title, value: a) }  
                .navigationTitle("Artykuły")  
                .navigationDestination(for: Article.self) { ArticleDetailView(article:  
$0) }  
            .searchable(text: $vm.searchQuery)  
        }.task { await vm.loadArticles() }  
    }  
}
```

Swift

Observation Framework (Swift 5.9+)

@Observable zastępuje ObservableObject + @Published. Prostsze, wydajniejsze (granular updates). Wymaga iOS 17+. @Bindable zamiast @Binding dla @Observable klas.

Flutter: Cross-Platform UI Framework od Google

Architektura i cechy

Własny silnik renderowania

Skia/Impeller — Flutter rysuje każdy piksel sam (nie używa natywnych widgetów). Pixel-perfect na każdej platformie. 60/120fps. Brak problemu z wyglądem natywnym.

Dart — AOT + JIT

Dart JIT w dev (hot reload < 1s). AOT w produkcji (native ARM). Dart: silnie typowany, GC, async/await natywny. Nauka: prosty dla Java/Kotlin/Swift developerów.

Widget tree

Wszystko jest Widget. Stateless i Stateful. Composition over inheritance.BuildContext — lokalizacja w drzewie. InheritedWidget = DI.

Platform Channels

MethodChannel, EventChannel — komunikacja Flutter ↔ Native kod Kotlin/Swift. Dostęp do kamery, BT, GPS przez plugin (pub.dev). Plugin = bridging.

pub.dev Packages

Ekosystem pakietów: 35 000+ packages. Popularne: provider, riverpod, bloc, go_router, dio, freezed, drift (SQLite). Jakość zmienna — sprawdzaj pub points.

Flutter (2018, Dart) to framework do budowania natywnie kompilowanych aplikacji dla iOS, Android, Web, Desktop i Embedded z jednej bazy kodu.

Kod widgetu

```
final counterProvider = StateProvider((_) => 0);  
  
class CounterScreen extends ConsumerWidget {  
  const CounterScreen({super.key});  
  @override  
  Widget build(BuildContext c, WidgetRef ref) {  
    final n = ref.watch(counterProvider);  
    return Scaffold(  
      appBar: AppBar(title: const Text('Licznik')),  
      body: Center(child: FilledButton(  
        onPressed: () => ref.read(counterProvider.notifier).state++,  
        child: Text('$n'),  
      )),  
    );  
  }  
}
```

Hot Reload vs Hot Restart

Hot Reload (r) — wstrzyknięcie kodu bez utraty stanu: < 1 sekunda. Hot Restart (R) — restart app ze stanem: 2-3 sekundy. Kluczowa przewaga Flutter nad natywnym rozwojem przy iteracjach UI.

React Native: Cross-Platform z JavaScript i JSX

React Native (Meta, 2015) używa JavaScript/TypeScript + React do tworzenia aplikacji renderowanych natywnie.
Nowa architektura JSI (2024) eliminuje stary bridge.

Architektura i komponenty React Native

```
export default function ArticleList() {  
  const [data, setData] = useState<Article[]>([]);  
  useEffect(() => { (async () => {  
    const r = await fetch('https://api.example.com/articles');  
    setData(await r.json());  
  })(); }, []);  
  return (  
    <FlatList data={data} keyExtractor={x => x.id}  
      renderItem={({item}) => <Text>{item.title}</Text>} />  
  );  
}
```

TypeScript

Porównanie cross-platform

Kryterium	Flutter	React Native
Silnik UI	Własny (Impeller/Skia) — pixel perfect	Natywne komponenty systemu (UIView/View)
Język	Dart (nowy, ale prosty)	JavaScript/TypeScript (znany dla webdev)
Performance	Zbliżony do natywu (AOT, Impeller)	JSI eliminuje bridge overhead (2024)
Hot Reload	< 1s (doskonały)	Fast Refresh (~2–3s)
Ekosystem	pub.dev 35k+ pakietów	npm (miliony pakietów, jakość zmienna)
Adopcja	BMW, eBay, Alibaba, Nubank	Facebook, Instagram, Shopify, Discord
Native access	Platform Channels (Kotlin/Swift bridge)	Native Modules + Turbo Modules (nowe)

Kiedy Flutter? Gdy potrzebujesz pixel-perfect UI na wielu platformach i nie masz webdev background.
Kiedy RN? Gdy masz webdev team i potrzebujesz natywnego look&feel.

Wzorce architektoniczne: MVC, MVP i MVVM

Wzorce architektury nie są dodatkiem. To są narzędzia, które decydują o **testowalności**, **utrzymywalności** i **rozszerzalności** aplikacji.

MVVM — Model-View-ViewModel (standard):

```
class ArticleVM(private val repo: Repo) : ViewModel() {  
    sealed interface UiState { data object Loading; data class  
    Ok(val a: List<Article>): UiState; data class Err(val m: String):  
    UiState }  
    private val _s = MutableStateFlow<UiState>(UiState.Loading)  
    val s: StateFlow<UiState> = _s  
    fun load() = viewModelScope.launch { runCatching  
    { repo.getArticles() }  
        .onSuccess { _s.value = UiState.Ok(it) }  
        .onFailure { _s.value = UiState.Err(it.message ?: "Błąd") } }  
}
```

Kotlin

Porównanie wzorców

MVC (Model-View-Controller)

iOS UIKit klasyczna architektura. Controller pośredniczy $M \leftrightarrow V$.
Problem: 'Massive View Controller'. Wszystko w jednym pliku ViewController.

MVP (Model-View-Presenter)

Testowalna: Presenter nie zna Androida/iOS → unit test. View interfejs.
Problem: boilerplate — dużo interfejsów. Popularna przed ViewModel.

MVVM (Model-View-ViewModel)

Standard Android (ViewModel API) i iOS (SwiftUI @StateObject). ViewModel nie zna View. Data binding / Combine / StateFlow. Testowalny ViewModel.

MVI (Model-View-Intent)

Unidirectional data flow. State → Event → Intent → Reducer → nowy State.
Przewidywalny. Popularne z Compose: Orbit MVI, Decompose.

Clean Architecture (warstwowa)

Presentation → Domain → Data. Reguła zależności (wewnętrzna warstwa nie zna zewnętrznej). Use Cases (Interactors). Nadmiarowa dla małych projektów.

Nawigacja, czyli Navigation Component, NavController i NavHost

Nawigacja między ekranami to fundamentalna część aplikacji mobilnej.
Zły system nawigacji = trudne w utrzymaniu spaghetti back-stack.

Android — Jetpack Navigation Compose

```
// Navigation Compose type-safe routes
@Serializable object Home
@Serializable data class Detail(val id: Int)

@Composable fun AppNav() {
    val nav = rememberNavController()
    NavHost(nav, Home) {
        composable<Home> { HomeScreen
    }
    { nav.navigate(Detail(it)) } }
    composable<Detail> { DetailScreen(it.toRoute<Detail>().id) }
}
}
```

Kotlin

iOS — UINavigationController i SwiftUI routing

```
// iOS NavigationPath (iOS 16+) type-safe / stos ekranów
@State private var path = NavigationPath()
NavigationStack(path: $path) {
    HomeView { path.append($0) }
    .navigationDestination(for: Article.self) { ArticleDetailView(article:
$0) }
}

protocol Coordinator { func start() }
final class AppCoordinator: Coordinator {
    let nav = UINavigationController()
    func start() { nav.setViewControllers([HomeVC(coordinator: self)],
animated: false) }
    func showDetail(_ a: Article)
{ nav.pushViewController(DetailVC(article: a), animated: true) } }
```

Swift

Deep Links i Universal Links

Android: Intent Filters (<data android:scheme='myapp' android:host='article'/>). iOS: Universal Links (apple-app-site-association JSON na serwerze + Associated Domains entitlement). QR kod → uruchamia app na odpowiednim ekranie.

Persystencja danych — Room, SharedPreferences i SwiftData

Każda aplikacja musi przechowywać dane. Od prostych preferencji użytkownika po złożone relacyjne bazy danych — wybór warstwy ma ogromny wpływ na architekturę.

Android — Room Database (SQLite ORM)

```
@Entity data class ArticleEntity(  
    @PrimaryKey val id: Int, val title: String)  
@Dao interface ArticleDao {  
    @Query("SELECT * FROM ArticleEntity") fun all():  
  
    Flow<List<ArticleEntity>>  
    @Upsert suspend fun upsert(a: ArticleEntity)  
}
```

Kotlin

iOS — SwiftData i UserDefaults

```
@Model final class Article {  
    var id = UUID(); var title: String; var publishedAt: Date = .now  
    init(title: String) { self.title = title }  
}  
  
Button("+") {  
    context.insert(Article(title: "Nowy"))  
    try? context.save()  
}
```

Swift

DataStore (Android) zastępuje SharedPreferences

Proto DataStore: Protobuf schema, type-safe. Preferences DataStore: key-value async (Flow<Preferences>). Nie blokuje UI wątku. `androidx.datastore:datastore-preferences`.

Keychain (iOS): bezpieczne przechowywanie

Keychain Services API → szyfrowane, poza sandboxem. Swift: KeychainSwift library. Dla tokenów i haseł. Android ekwiwalent: EncryptedSharedPreferences.

Sieć i REST API — OkHttp, Retrofit i URLSession

Niemal każda aplikacja mobilna komunikuje się z backendem przez REST API. Właściwa obsługa błędów, timeoutów i braku sieci to warunek konieczny dobrego UX.

Android — Retrofit + OkHttp + Kotlin Serialization

```
interface Api { @GET("articles") suspend fun articles():  
List<ArticleDto> } Kotlin  
  
val api = Retrofit.Builder()  
    .baseUrl(BASE_URL)  
    .addConverterFactory(Json.asConverterFactory(  
"application/json".toMediaType()))  
    .build()  
    .create(Api::class.java)
```

iOS — URLSession + async/await + Codable

```
func fetch<T: Decodable>(_ ep: String) async throws -> T { Swift  
    var req = URLRequest(url: baseUrl.appending(path: ep))  
    req.setValue("Bearer \(token)", forHTTPHeaderField:  
"Authorization")  
    let (data, res) = try await URLSession.shared.data(for: req)  
    guard (res as? HTTPURLResponse)?.statusCode ?? 0 < 300 else  
    { throw APIError.invalid }  
    return try JSONDecoder().decode(T.self, from: data)  
}
```

Offline-first strategy

Cache-then-network: pokaż dane lokalne → załaduj z sieci → aktualizuj. Room/SwiftData jako single source of truth. NetworkMonitor (NWPathMonitor) do wykrywania dostępności sieci.

Bezpieczeństwo sieci

SSL Pinning: certyfikat zakodowany w app (nie zawsze dobry pomysł, bo wymaga release gdy cert wygaśnie). Certificate Transparency. Network Security Config (Android). ATS (App Transport Security — iOS).

Dystrybucja: Google Play, App Store i Proces Review

Budowa aplikacji to 60% pracy. Dystrybucja, review, wersjonowanie, release notes i odpowiedź na oceny to pozostałe 40%. App Store Review może trwać 1-3 dni.

Google Play — publikacja krok po kroku:

- 1 Konto dewelopera**
play.google.com/console → \$25 jednorazowo. Weryfikacja tożsamości (dowód + selfie). 48h aktywacja konta.
- 2 App Bundle (AAB)**
Android App Bundle (.aab) — nie APK! Play Dynamic Delivery dzieli app na moduły. Mniejszy download użytkownika. bundletool do testowania.
- 3 Store Listing**
Opis (maks. 4000 znaków), screenshots (min. 2 na urządzenie), feature graphic (1024×500). Tłumaczenie dla rynków: EN + PL + DE + ES.
- 4 Content Rating IARC**
Kwestionariusz treści → automatyczny rating (PEGI/ESRB). Wymagany przed publikacją. Gry z przemocą → 18+.
- 5 Release tracks**
Internal → Alpha → Beta → Production. Percentage rollout: 10% → 20% → 100%. Staged rollout dla monitorowania crashy.

Apple App Store — review i wymagania:

- 1 Xcode Archive + Signing**
Product → Archive → Distribute. Certyfikat Distribution. Provisioning Profile z App Store Connect. Bitcode opcjonalny (deprecated iOS 16+).
- 2 App Store Connect**
appstoreconnect.apple.com. Screenshots: iPhone 15 Pro Max (6.7") + iPad (12.9"). Privacy nutrition labels obowiązkowe.
- 3 TestFlight**
Do 10 000 zewnętrznych testerów. Email invitation lub public link. Wersja beta = szybszy review (1-2 dni). Crashes auto-reported.
- 4 App Review (Human)**
1-3 dni. Automated checks + human review. Najczęstsze odrzucenia: niekompletny UX, brak polityki prywatności, placeholder content.
- 5 Metadata + ASO**
App Store Optimization: tytuł (30 znaków), subtitle (30), keywords (100). Reviews & Ratings: aktywnie zarządzaj. A/B testy icon/screenshot.

CI/CD dla mobile: Fastlane (open-source) automatyzuje build → sign → upload dla obu platform.
GitHub Actions + Fastlane = w pełni zautomatyzowany pipeline.

Testowanie Aplikacji Mobilnych: Piramida i Strategie

Testowanie mobilne: unit tests (logika), integration tests (baza/sieć) i UI tests (end-to-end).
Dobre pokrycie = odwaga przy refactoringu.

Android — JUnit5 + MockK + Compose Testing

```
@Test fun `load -> Success`() = runTest {  
    val repo = FakeRepo().apply { emit(listOf(Article(1,"A"))) }  
    val vm = ArticleVM(repo); vm.load()  
    assertTrue(vm.s.value is UiState.Ok)  
}  
  
@Test fun add_showsItem() {  
    rule.setContent { Screen(vm) }  
    rule.onNodeWithText("Dodaj").performClick()  
    rule.onNodeWithText("Nowy").assertIsDisplayed()  
}
```

Kotlin

iOS — XCTest + Quick + Nimble

```
// iOS Unit Test — XCTest + async  
func testLoad_success() async throws {  
    let repo = MockRepo(articles: [Article(id: 1, title: "Swift")])  
    let vm = ArticleVM(repository: repo)  
    await vm.loadArticles()  
    XCTAssertEqual(vm.articles.count, 1)  
}
```

Swift

Unit Tests

ViewModel, UseCase, Repository, Utilities

[JUnit5/XCTest](#), [MockK/Mockito](#), [Swift Testing](#)

Integration Tests

Room DB queries, API real calls (WireMock)

[Robolectric](#), [OkHttp MockWebServer](#), [HTTPStubs](#)

UI / E2E Tests

User flows, navigation, accessibility

[Compose Testing](#), [Espresso](#), [XCUITest](#), [Detox](#)

CI/CD: automatyzacja Buildu, testów i deploymenty

Continuous Integration/Delivery to standard profesjonalnego mobile development. Automatyczny build i testy przy każdym commit = wczesne wykrywanie problemów.

GitHub Actions: pipeline Android

```
name: Android CI YAML
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with: { java-version: '21', distribution: 'temurin' }
      - run: ./gradlew test lint bundleRelease
```

Narzędzia CI/CD dla mobile

Fastlane

Open-source. match (certs/profiles), gym (build), deliver (upload). Działa z GitHub Actions, CircleCI, Bitrise. Najszerzej adoptowane.

Bitrise

Dedykowany mobile CI. Gotowe Steps dla Android/iOS. Apple Silicon runners. Dobra integracja z Xcode Cloud. Płatny (darmowy plan = 200 min/mies).

Xcode Cloud (Apple)

Natywny CI dla iOS/macOS. Integracja z App Store Connect. TestFlight auto-upload. Free 25h/mies. Wymaga Apple Developer Program.

Firebase App Distribution

Dystrybucja build do testerów (alternatywa TestFlight dla Android). Firebase Crashlytics — raporty błędów z produkcji w real-time. Darmowy.

Gradle Build Scans

Analiza wydajności buildu Gradle. Visualizacja task graph. Identyfikacja wąskich gardeł. Darmowy na Develocity Scans (5 skanów/mies).

Performance: optymalizacja, profiling i metryki

Mobile performance to UI fluidity (60fps), czas uruchamiania, zużycie pamięci i baterii. Użytkownik odinstaluje 30% aplikacji z powodu slow performance.

Android — metryki i narzędzia

Frame Rate (Jank)

Cel: 60fps = 16.67ms/frame. 120Hz: 8.3ms. Renderowanie > 16ms = dropped frame = jank. Narzędzie: systrace, Perfetto, CPU Profiler.

App Startup Time

Cold start: od kliknięcia ikony do Interactive. Cel < 500ms. Warm/Hot start. Baseline Profiles (AOT Compose code) → 30-40% szybszy cold start.

Memory (ANR / OOM)

ANR (Application Not Responding) = UI wątek > 5s. OOM = Out Of Memory crash. Heap analyzer: Memory Profiler. LeakCanary wykrywa memory leaks.

Battery (Doze compliance)

BatteryHistorian narzędzie. WakelockManager — ilość i czas. WorkManager Doze-aware. JobScheduler → batched jobs. Target: < 3% baterii/h w background.

iOS — Instruments i metryki Apple

Time Profiler

Instrument do analizy CPU. 1ms sampling. Call tree: identyfikuj najdroższe funkcje. Backtrace → widok stosu wywołań. ALWAYS profile on device, nie Simulator.

Allocations + Leaks

Allocations: heap snapshot. Leaks: automatyczna detekcja retain cycles. Instruments mark generation dla identyfikacji leaków konkretnych operacji.

Core Animation / Display

FPS live graph. Offscreen rendering (żółte). Blended layers (czerwone). Color Copied Images. Narzędzie: Instruments Core Animation instrument.

MetricKit (iOS 13+)

Systemowe metryki produkcyjne: launch time (histogramy), hang rate, memory pages, CPU instructions, scroll hitch rate. Bez instrumentacji kodu. Bezpośrednio w App Store Connect.

Zasada: Measure first, optimize second. Nie optymalizuj bez danych z profilera. Przedwczesna optymalizacja to źródło 95% problemów.

UX dla wszystkich: dostępność (a11y) i lokalizacja (i18n)

Dostępność to nie dodatek — to wymóg prawny w UE i USA (WCAG 2.2, Section 508). Lokalizacja to nowe rynki. Obie zaczynają się na etapie projektowania, nie po.

Dostępność — TalkBack / VoiceOver

```
@Composable Kotlin  
fun Like(liked: Boolean, onClick: () -> Unit) =  
    IconButton(onClick, Modifier.semantics {  
        contentDescription = if (liked) "Usuń z ulubionych" else  
        "Dodaj do ulubionych"  
        stateDescription = if (liked) "Polubione" else  
        "Niepolubione"  
        role = Role.Button  
    }) { Icon(Icons.Default.Favorite, contentDescription =  
        null) }
```

```
Button(action: toggle) { Image(systemName: fav ?  
    "heart.fill" : "heart") }  
    .accessibilityLabel(fav ? "Usuń z ulubionych" : "Dodaj  
do ulubionych")  
    .accessibilityAddTraits(.isButton)
```

Kotlin

Lokalizacja (i18n) — krok po kroku

Strings externalization

Android: res/values/strings.xml (EN), values-pl/strings.xml (PL).
iOS: Localizable.strings / Localizable.xcstrings (Xcode 15+).
Nigdy nie hardcoduj tekstu w kodzie.

Pluralizacja

Android: < plurals > tag. iOS: stringsdict. Przykład: '1 artykuł' vs '3 artykuły' — reguły różnią się językowo (PL: 1, 2-4, 5+). ICU message format.

Formaty dat i liczb

DateFormat.getDateInstance(SHORT, Locale.PL). iOS: Date.FormatStyle(.date).
Nigdy hardcode dd/MM/yyyy, a w USA będzie MM/dd/yyyy.

RTL (Right-to-Left)

Arabski, Hebrajski. Android: android:supportsRtl='true'. Compose:
LocalLayoutDirection. iOS: automatycznie z Locale. Użyj leading/trailing nie
left/right.

Tłumaczenia workflow

Narzędzia: Phrase (fka PhraseApp), Crowdin, Lokalise. Export → CSV/XLIFF →
tłumacz → import. Integracje z CI/CD. Pamiętaj: tłumaczenia wydłużają/skracają tekst
20-40%.

Bezpieczeństwo aplikacji mobilnych — OWASP Mobile Top 10

Bezpieczeństwo aplikacji mobilnej zaczyna się od pierwszego commitu. OWASP Mobile Top 10 (2024) definiuje najczęstsze podatności — znaj każdą z nich.

M1

Improper Credential Usage

Hardcoded API keys, passwords w kodzie lub resources.
BuildConfig + GitHub Secrets. Keychain/Keystore. Serwer proxy dla secrets.

M3

Insecure Authentication

Weak password policy. Brak biometrii. JWT stored insecure.
Biometria + Keystore. OAuth 2.0 + PKCE. FIDO2/WebAuthn. Short-lived tokens.

M5

Insecure Communication

HTTP (brak TLS). Brak Certificate Pinning. MitM możliwy.
TLS 1.3 tylko. Network Security Config. Certificate/Public Key Pinning selektywny.

M7

Insufficient Binary Protections

Reverse engineering APK (jadx). Secret keys widoczne.
ProGuard/R8 obfuscation. DexGuard. Frida detection (root/jailbreak check).

M2

Inadequate Supply Chain Security

Podatne zależności (npm/Gradle). Niezaufane pakiety.
Dependabot/Renovate automatyczne updates. SBOM (Software Bill of Materials).

M4

Insufficient Input/Output Validation

SQL Injection przez Room. XSS w WebView. Path traversal.
Prepared statements (Room).
WebView.setJavaScriptEnabled(false). Input sanitization.

M6

Inadequate Privacy Controls

Over-permissive. PII w logach. Analytics bez zgody (RODO).
Minimal permissions. Cleartext banning. Consent management (OneTrust/Didomi).

M8

Security Misconfiguration

Debug mode w produkcji. Backup allowBackup=true.
android:debuggable=false. allowBackup=false.
NSAppTransportSecurity tigt.

Rynek pracy Android/iOS Developer, ścieżki i zarobki 2025

Programista mobilny to jedna z najlepiej opłacanych specjalizacji w IT. Popyt przewyższa podaż. Cross-platform skills (Flutter/RN) otwierają dodatkowe możliwości.

~25k

Ofert mobile dev
w Polsce 2024

14-22k

Zarobki midlevel
brutto PLN/mies.

22-40k+

Zarobki senior/lead
brutto PLN/mies.

€60-120k

Zarobki UE/zdalnie
rocznie EUR

Ścieżki kariery i stos technologiczny:

Junior Android Dev (0-2 lata)

Kotlin, Compose, Room, Retrofit, Coroutines/Flow, MVVM. Bonus: Hilt DI, Navigation.

Mid Android Dev (2-4 lata)

+ Clean Architecture, Modularyzacja, CI/CD Fastlane, Testing (JUnit5 + MockK), Compose advanced.

Senior Android Dev (4+ lat)

+ SDK design, Performance Profiling, Baseline Profiles, Security hardening, Tech leadership, code review.

Junior iOS Dev (0-2 lata)

Swift, SwiftUI, URLSession, Codable, UserDefaults/SwiftData, MVVM. Bonus: Combine, SPM.

Mid iOS Dev (2-4 lata)

+ UIKit (legacy), XCTest, Instruments, SPM packages, StoreKit (IAP), WidgetKit.

Flutter Dev (wszystkie poziomy)

Dart, Riverpod/Bloc, dio, go_router, Drift, flutter_test. pub.dev package authoring. Cross-platform.

Portfolio > certyfikaty. Pracodawcy chcą zobaczyć 2-3 aplikacje na GitHubie lub App Store/Play. Kontrybuuj do open-source, pisz tech blog.

Trendy i przyszłość — AI Native, Spatial Computing i Edge ML

Programowanie mobilne w 2025 to coś więcej niż CRUD + lista. On-device AI, spatial computing (Vision Pro) i Web3 zmieniają paradygmat developmentu.

AI Native Apps (Gemini Nano / Apple Intelligence)

On-device LLM 3–7B parametrów wbudowane w OS. API: Google AI Core (Gemini Nano), Apple Intelligence frameworks. Text summarization, image generation, smart replies — bez chmury. Kwiecień 2025: Gemini Nano on Android 14+ otwarty dla developerów. Prywatność: dane nie opuszczają urządzenia.

Spatial Computing (visionOS / Apple Vision Pro)

visionOS SDK: RealityKit, ARKit, SwiftUI 3D. Okna w przestrzeni (Volumes). Gesty oczami + palcami. React Native visionOS (Meta open-source). Nowe UX patterns: przestrzenne UI, bliskość obiektu, gaze interactions. Rynek spatial XR: \$50B do 2028.

Wearables Programming (watchOS / Wear OS)

watchOS 11: Widgets Smart Stack. WidgetKit + Intent Configuration. Wear OS 5: Kotlin DSL API. Tile (kafelek) development. HealthKit + Health Connect cross-platform. Bardzo małe ekrany = nowe wzorce UX.

Kotlin Multiplatform (KMP) / Compose Multiplatform

KMP stable (JetBrains 2024). Shared business logic: Repository, ViewModel, Network, Database. UI: Compose Multiplatform (iOS beta). Spotify, McDonald's, Netflix używają KMP production. 'Write once, run everywhere' dla logiki — nie UI.

Edge ML i CoreML / NNAPI

TensorFlow Lite / PyTorch Mobile / ONNX Runtime dla on-device. CoreML Tools: PyTorch/ONNX → .mlpackage. MediaPipe Solutions: pose, face, hands off-the-shelf. Federated Learning (Google): model trenuje na urządzeniu bez wysyłania danych.

App Clips / Instant Apps

iOS App Clips: < 15MB doświadczenie bez instalacji (QR, NFC, URL). SwiftUI App Clip Target. Android Instant Apps analogia. Use case: parking payment, menu restauracji, event check-in. First impressions bez friction.

Dobre praktyki i Code Quality — Zasady profesjonalnego developmentu

Zły kod działa. Dobry kod działa, jest czytelny i łatwy do zmiany. Zasady SOLID, Clean Code i code review to inwestycja w przyszłe godziny pracy.

SOLID w kodzie mobilnym:

S — Single Responsibility

Jeden ViewModel = jeden ekran. Repository = jedna encja danych. Composable = jeden cel UI.

O — Open/Closed

Protokoły/interfejsy → rozszerzaj przez implementację, nie modyfikuj bazowej klasy.

L — Liskov Substitution

Każda implementacja Repository musi działać jak bazowy interfejs (Fake/Real swap).

I — Interface Segregation

Małe interfejsy: ArticleReader + ArticleWriter osobno, nie jeden fat interface.

D — Dependency Inversion

ViewModel zależy od interfejsu Repository, nie konkretnej klasy → łatwe mocki w testach.

Code review i narzędzia jakości:

ktlint + Detekt (Android)

ktlint: formatowanie kodu Kotlin (Kotlin Coding Conventions). Detekt: ~500 reguł — complexity, smell, security. Uruchamiaj w CI jako gate.

SwiftLint + SwiftFormat (iOS)

SwiftLint: 200+ reguł, konfigurowalny .swiftlint.yml. SwiftFormat: auto-formatowanie. Xcode Build Phase integration → błędy w IDE.

SonarQube / SonarCloud

Static analysis: code smells, duplications, coverage. Dashboard dla całego projektu. Free dla open-source. CI integration.

Code Review checklist

Logika poprawna Testy napisane Brak magic numbers Error handling
No hardcoded secrets a11y sprawdzona Lokalizacja

Git workflow — GitHub Flow

main → feature/xxx → PR → review (2 approvals) → merge. Konwencja commitów: Conventional Commits (feat/fix/docs/chore). Semantic Release.

Cel i struktura projektu zaliczeniowego

Opis wymagań

Semestralny projekt polega na zaprojektowaniu i wdrożeniu kompletnej aplikacji mobilnej (frontend + backend) w warunkach zbliżonych do pracy w firmie IT. Zespół 3-osobowy pracuje w repozytorium Git z wykorzystaniem branchingu, pull requestów, code review, CI/CD, testów automatycznych oraz dokumentacji technicznej i użytkowej.

Projekt polega na zaprojektowaniu i wdrożeniu aplikacji mobilnej, która **realnie wykorzystuje możliwości smartfona**, a nie jest jedynie „przeniesioną wersją aplikacji desktopowej”. Aplikacja musi działać na **fizycznym urządzeniu (Android)** i zostać przygotowana do publikacji w sklepie play.

Checklista

1. Aplikacja działa na fizycznym smartfonie (prezentacja na zajęciach)
2. Wykorzystanie min. 2 natywnych funkcji urządzenia, np.:
 - aparat (Camera API)
 - GPS / lokalizacja
 - powiadomienia push
 - czujniki (akcelerometr, żyroskop)
 - biometria (odcisk palca / Face Unlock)
 - przechowywanie lokalne (np. offline-first)
3. Integracja z backendem (auth + operacje na danych)
4. Działający przepływ end-to-end (logowanie + operacja domenowa)
5. Minimum 20 zamkniętych issue + 2 milestone (sprinty)
6. Repozytorium z PR i code review
7. CI: build + testy przy każdym PR
8. Testy jednostkowe (≥60% pokrycia logiki)
9. Testy integracyjne API (min. 5 scenariuszy)
10. Dokumentacja API (OpenAPI/Swagger)
11. Wygenerowany podpisany build (AAB/APK)
12. Przygotowany opis aplikacji do Google Play (opis, screeny, ikona, polityka prywatności)
13. Próba publikacji w Google Play (kanał testowy lub produkcyjny)
14. 5-minutowe demo + prezentacja procesu CI/CD

Zakres obowiązków wg ról (checklista indywidualna)

1. Product Lead & UX

- Min. 15 user stories z kryteriami akceptacji
- Wyraźne uzasadnienie: dlaczego aplikacja wymaga smartfona
- Prototyp uwzględniający interakcję mobilną (gesty, kontekst lokalizacji, aparat itp.)
- MVP vs funkcje dodatkowe
- Opis aplikacji do Google Play (krótka + pełny opis, słowa kluczowe)
- Przygotowanie screenów i materiałów promocyjnych
- Checklista testów akceptacyjnych
- Changelog + instrukcja użytkownika

2. Mobile Developer (Frontend)

- Implementacja min. 5 ekranów
- Integracja min. 2 funkcji natywnych urządzenia
- Obsługa uprawnień systemowych (runtime permissions)
- Obsługa stanów: loading / error / offline
- Integracja z API (auth + min. 3 operacje CRUD)
- Min. 10 testów jednostkowych
- Konfiguracja podpisanego builda (keystore, wersjonowanie)
- Przygotowanie wersji AAB/APK gotowej do Play Console

3. Backend & DevOps Engineer

- API: min. 5 endpointów (w tym rejestracja/logowanie)
- Uwierzytelnianie (np. JWT) + hashowanie haseł
- Model bazy danych + migracje
- Min. 5 testów integracyjnych API
- CI backendu (testy automatyczne)
- Deployment (np. chmura / hosting publiczny)
- Walidacja danych, CORS, podstawowe zabezpieczenia
- Przygotowanie polityki prywatności (wymaganej do Play Store)

Model oceny

Ocena zespołowa (40%)

- działanie aplikacji na fizycznym urządzeniu,
- poprawność integracji mobile-backend,
- jakość CI i testów,
- gotowość do publikacji w Google Play.

Ocena indywidualna (60%)

- realizacja checklisty roli,
- jakość kodu i dokumentacji,
- aktywność w PR i code review,
- odpowiedzialność za własny obszar.

Projekt ma symulować realny proces wytwarzania i publikacji aplikacji mobilnej.

-> Od koncepcji, -> przez implementację, -> aż do dystrybucji w sklepie.

Projekt semestralny i wymagania przedmiotu

Kurs Online

<https://e-learning2025.prz.edu.pl/course/view.php?id=55>

Materiały

<https://mpomianek.v.prz.edu.pl/materiały-dydaktyczne/programowanie-aplikacji-mobilnych>



The screenshot shows a course page with a blue header. On the left, there is a circular profile picture with the initials 'MP' and the name 'Mateusz Pomianek'. To the right of the name is the course title 'Programowanie Aplikacji Mobilnych' in a large, bold, white font. Below the title is a white button with the text 'RESUME COURSE'. On the right side of the header, there is a photograph of a person's hands typing on a laptop keyboard, with a smartphone displaying a mobile app interface next to it. Below the header, there is a paragraph of text in a light blue font. At the bottom right of the page, there is a small white square button with an upward-pointing arrow.

MP Mateusz Pomianek

Programowanie Aplikacji Mobilnych

RESUME COURSE

Ten kurs wprowadzi Cię krok po kroku w świat programowania aplikacji mobilnych. Poznasz architekturę aplikacji, nauczysz się korzystać z popularnych narzędzi programistycznych, zaprojektujesz własny interfejs użytkownika i dowiesz się, jak testować oraz uruchamiać swoje projekty na urządzeniach mobilnych. Dzięki praktycznym ćwiczeniom i przystępnej

Projekt semestralny i wymagania przedmiotu

Platforma

Android (Kotlin + Jetpack Compose), iOS (Swift + SwiftUI), Flutter, React Native, Unity, Python-> Decyzja do połowy semestru.

Minimum scope

≥ 4 ekrany, persystencja danych (Room/SwiftData), REST API call, obsługa błędów (brak sieci, puste stany), testy jednostkowe, ViewModel.

Architektura

MVVM obligatoryjnie. Warstwa Repository oddzielona od ViewModel.
Clean code: brak magic numbers, czytelne nazwy, brak God-class.

CI/CD

GitHub Actions lub Fastlane: automatyczne testy przy każdym push na main.
Code coverage > 60% ViewModel. Lint jako CI gate.

Bonus punkty

Dostępność (a11y) TalkBack/VoiceOver. Ciemny motyw. Lokalizacja (PL + EN).
Widget (WidgetKit/Glance). Powiadomienia push. Animacje.