Kotlin and Compose Multi-Platform App

Gonçalo Medeiros[1], Paulo Proença[1], Rui Almeida[2]

[1] Instituto Superior de Engenharia do Porto, Porto, Portugal
[2] Mindera, Portugal

Resumo. A diversidade crescente de dispositivos e sistemas operativos constitui um desafio no desenvolvimento de aplicações móveis. Este artigo descreve o desenvolvimento de uma aplicação móvel multiplataforma desenvolvida em colaboração entre a Mindera e a Federação Portuguesa de Rugby. Recorrendo a Kotlin Multiplatform e Compose Multiplatform, a solução permite gerar aplicações para Android, iOS e desktop a partir de uma única base de código, promovendo reutilização, eficiência e consistência. São discutidas as vantagens e limitações da abordagem, com vista a orientar futuros projetos nesta área. Os resultados demonstram que a abordagem multiplataforma é viável e eficiente, fornecendo uma base sólida para o desenvolvimento de aplicações móveis

Palavras-chave: Kotlin Multiplatform, Compose Multiplatform, Aplicações móveis.

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Projeto/Estágio (PESTI), em colaboração com a empresa Mindera e a Federação Portuguesa de Rugby (FPR), surgindo como resposta à crescente necessidade de soluções que simplifiquem o desenvolvimento de aplicações móveis, num contexto em que a diversidade de plataformas e sistemas operativos aumenta significativamente os custos e a complexidade do processo, a par da necessidade de centralizar a informação atualmente dispersa da FPR.

O projeto propõe o desenvolvimento de uma aplicação móvel multiplataforma recorrendo ao uso das tecnologias *Kotlin Multiplatform* (KMP) e *Compose Multiplatform* (CMP), que permitem criar aplicações para Android, iOS e desktop a partir de uma base de código única, com elevado grau de reutilização e desempenho próximo do nativo [1].

Neste contexto, os objetivos incluem o estudo do Estado da Arte, implementação de funcionalidades essenciais, adoção de boas práticas de programação para uma solução escalável, análise das tecnologias utilizadas e documentação do processo como referência para trabalhos futuros.

Assim, o trabalho realizado com suporte na *framework* Kanban moderniza o acesso à informação da FPR, permite à Mindera testar novas tecnologias, documentando as respetivas vantagens e limitações, com utilidade para projetos futuros e promove a adoção de novas tecnologias no setor desportivo. Por fim, o projeto poderá inspirar outras organizações desportivas a adotar soluções tecnológicas semelhantes.

2 Estado da arte

2.1 Trabalhos relacionados

Diversas soluções digitais têm modernizado a forma como o rugby é acompanhado, aproximando adeptos, jogadores e federações e melhorando a experiência dos utilizadores.

Entre as aplicações analisadas, destaca-se a app do torneio *Six Nations*, que oferece cobertura em tempo real, conteúdos exclusivos, notificações personalizadas, análises interativas jogos *fantasy* rugby, focando na interação e envolvimento dos fãs [2]. Por outro lado, a aplicação da Federação Italiana de Rugby (FIR) privilegia a informação institucional e a ligação à comunidade, disponibilizando calendários, notícias, perfis de jogadores e integração com redes sociais [3].

Estas plataformas serviram como referência para o desenvolvimento do presente projeto, inspirando funcionalidades como notícias, perfis e informação atualizada dinamicamente. Contudo, não se optou pelo uso de notificações push devido a custos associados, mantendo-se em aberto a possível futura integração de jogos e análises interativas.

2.2 Tecnologias existentes

A escolha da tecnologia é essencial e depende dos requisitos do projeto. Apresentam-se, por isso, as soluções existentes para justificar a abordagem selecionada.

Soluções Multiplataforma. As tecnologias multiplataforma constituem uma abordagem moderna ao desenvolvimento de software, permitindo a criação de aplicações a partir de uma base de código única, com capacidade de compilação para diversos sistemas operativos e plataformas.

KMP, lançada em 2023 pela JetBrains, é uma tecnologia *open source* que permite partilhar código entre plataformas como Android, iOS, web e desktop, a partir de uma base única. O código *Kotlin* é transformado numa representação intermédia (IR), sendo depois compilado consoante a plataforma de destino, para *bytecode* da JVM para Android ou qualquer ambiente que suporte Java, *JavaScript* em aplicações web e binários nativos em iOS, macOS, Windows e Linux, através do *Kotlin/Native* com suporte do LLVM [4] [5].

CMP é uma *framework* para interfaces gráficas multiplataforma, baseado no Jetpack Compose do Android e mantido pela JetBrains. Suporta Android, iOS, desktop e web, permitindo o desenvolvimento de interfaces declarativas reutilizáveis. A renderização é assegurada pelo motor *Skia*, usado diretamente no Android e, nas restantes plataformas, através da biblioteca *Skiko*, garantindo desempenho e consistência visual [6] [7].

Além destas tecnologias, destacam-se outras soluções multiplataforma, como o Flutter, da Google, que utiliza a linguagem *Dart* para criar aplicações para Android, iOS, desktop e web a partir de um único código-fonte. O Flutter garante um desenho consistente da interface em todas as plataformas através do seu motor gráfico, que evoluiu do *Skia* para o mais eficiente *Impeller*.

O *React Native*, da Meta, permite desenvolver aplicações móveis em *JavaScript* com base no *ReactJS*, seguindo uma abordagem declarativa e modular próxima da experiência web. Apesar de permitir elevada reutilização de código entre plataformas, o seu desempenho é inferior ao nativo, devido à necessidade de comunicação através de uma ponte entre o motor *JavaScript* e os componentes nativos, o que introduz *overhead*.

Por fim, o *Skip* é uma tecnologia emergente centrada numa abordagem *iOS-first*, desenvolvida em *Swift* e *SwiftUI*, que gera automaticamente código equivalente para Android utilizando *Kotlin* e *Jetpack Compose*. Esta abordagem permite tirar partido do ecossistema nativo iOS, mantendo a compatibilidade com Android sem duplicação manual de lógica [1] [8] [9] [10] [11]. A Tabela 1 apresenta uma síntese comparativa das suas principais características e diferenças.

***Tabela 1**. Comparação entre as tecnologias existentes*

	KMP CMP	Flutter	React Native	Skip
Single Code	Sim	Sim	Sim	Sim
Linguagem	Kotlin	Dart	JavaScript	Swift
Execução	Compilado	Compilado	Interpretado	Compilado
Desempenho	Alto	Alto	Baixo	Alto
Render Canvas	Skia	Skia	Flexbox	Impeller/Skia
Popularidade	Alta	Muito alta	Muito alta	Alta

Injeção de dependências. A injeção de dependências é uma técnica que promove uma arquitetura mais modular, facilitando a reutilização e a testabilidade do código [12].

Koin é uma biblioteca de gestão de dependências escrita em *Kotlin*, conhecida pela simplicidade e fácil integração com projetos KMP. Combina características de injeção de dependências e *Service Locator*, oferecendo flexibilidade na resolução de dependências [13] [14].

Dagger é uma biblioteca de injeção de dependências para Android e *Java* que opera em tempo de compilação, garantindo eficiência e segurança, ao validar o grafo de dependências e evitar erros em tempo de execução [15].

Kotlin Inject é uma biblioteca de injeção de dependências totalmente escrita em *Kotlin*, que oferece segurança em tempo de compilação e uma API similar ao *Dagger*, mas mais idiomática para o ecossistema *Kotlin* [15].

Testes. A criação de *mocks* permite isolar componentes e garantir uma validação eficaz em ambientes multiplataforma KMP. As limitações da *MockK* fora da JVM tornam necessário recorrer a alternativas com suporte multiplataforma. Entre as opções mais utilizadas, *Mockative* recorre a *KSP* para gerar *mocks* compatíveis com múltiplos *targets*, enquanto a *Mokkery* utiliza um plugin de compilador para produzir *mocks* totalmente suportados em KMP [16] [17]. A Tabela 2 compara a compatibilidade das *frameworks* de *mocking* com targets KMP.

***Tabela 2**. Frameworks de mocking em KMP*

	JVM	**iOS**
Mockk	Sim	Não
Mockative	Sim	Sim
Mokkery	Sim	Sim

Base de dados em tempo real. Uma base de dados em tempo real permite a sincronização imediata de dados entre dispositivos, refletindo atualizações em milissegundos, sendo especialmente útil em contextos interativos [18].

Supabase, lançada em 2020, é uma base de dados relacional que oferece serviços de *backend*, como autenticação e armazenamento, para aplicações Android, iOS e web [18].

Firebase, criado em 2011 e adquirido pela Google em 2014, é uma base de dados *NoSQL* que disponibiliza serviços semelhantes aos do *Supabase*, incluindo autenticação e armazenamento de dados [18]. A Tabela 3 sintetiza as principais diferenças entre as bases de dados Supabase e Firebase, destacando características relevantes para aplicações multiplataforma.

***Tabela 3**. Comparação entre Firebase e Supabase*

	Supabase	Firebase
Tipo	Relacional	NoSQL
Autenticação	E-mail, senha e terceiros	E-mail, senha e terceiros
Armazenamento	Buckets com políticas de segurança	Google Cloud Storage para multimédia
Hospedagem	Indisponível	Disponível para conteúdo estático e dinâmico
Preços	API ilimitada (plano gratuito); até 10.000 utilizadores	Cobra por operações; utilizadores ilimitados
Self-Hosting	Disponível (open source)	Indisponível

3 Análise e desenho da solução

3.1 Análise do Problema

O projeto centrou-se na seleção nacional de rugby ("Os Lobos"), tendo em consideração a relevância e a atenção que, esta, tem vindo a obter no contexto desportivo português. A solução foi desenvolvida sob a forma de um *Minimum Viable Product* (MVP), com foco exclusivo na seleção nacional. A aplicação tem suporte multiplataforma, abrangendo os ambientes Android, iOS e desktop.

3.2 Domínio do Problema

A modelação do domínio baseou-se nos princípios do *Domain-Driven Design* (DDD). O núcleo funcional da aplicação assenta na gestão de jogos, cujos eventos influenciam diretamente as estatísticas e classificações. O domínio da aplicação abrange entidades como equipas, jogadores, treinadores, membros do staff, grupos de liga, épocas desportivas, campos de jogo, entre outros.

3.3 Requisitos Funcionais e Não Funcionais

A engenharia de requisitos é uma etapa essencial no desenvolvimento de software, definindo tanto as funcionalidades do sistema como os critérios de qualidade, nomeadamente desempenho, segurança e usabilidade [19].

Requisitos Funcionais (RF). Em termos funcionais, foram identificados os seguintes RF:

- **Consulta de jogadores:** permitir ao utilizador consultar informação detalhada sobre jogadores;
- **Notícias:** aceder a notícias da modalidade;
- **Resultados e calendário:** visualizar resultados e o calendário de jogos;
- **Classificações e equipas:** consultar classificações e dados das equipas;
- **Atualização:** garantir um mecanismo de atualização forçada da aplicação;

Requisitos Não Funcionais (RNF). Para a classificação dos requisitos não funcionais, recorreu-se ao modelo FURPS+. Assim, foram identificados os seguintes RNF:

- **Funcionalidade:** utilização de *HTTPS* nas comunicações com o servidor;
- **Usabilidade:** adotar a identidade visual da FPR e apresentar uma experiência consistente entre plataformas suportando gestos nativos;
- **Desempenho:** tempos de resposta < 100 ms em conexões ideais e fluidez ≥ 60 fps;
- **Manutenibilidade:** arquitetura modular baseada em *Clean Architecture* com padrão *MVVM*;
- **Suportabilidade:** A arquitetura deve ser modular, facilitando manutenção, escalabilidade e integração de novas funcionalidades;
- **Extra (+) Restrições de implementação:** uso de KMP e CMP, para compilação iOS é necessário *macOS*, adoção de uma arquitetura *Clean Architecture* com *MVVM*, e a cobertura por testes deve ultrapassar os 70%.

3.4 Arquitetura do Sistema

O sistema adota uma arquitetura modular baseada na *Clean Architecture*, organizada em camadas que seguem os princípios SOLID, respeitando a regra da inversão de dependências, segundo a qual as camadas internas do domínio são independentes das camadas externas. Este modelo garante que o núcleo do domínio permanece independente das camadas externas, promovendo isolamento, testabilidade e facilidade de manutenção. A arquitetura do sistema organiza-se em seis módulos principais, *Domain*, *Data*, *Presentation*, *DI*, *Shared* e *App* [20].

Inicialmente, a arquitetura era composta pelos módulos *Presentation*, *Data*, *Domain* e *Shared*. Contudo, esta estrutura revelou-se insuficiente para uma segregação eficaz de dependências, nomeadamente devido à dependência direta e desnecessária entre o arranque da aplicação e

Domain, bem como o desalinhamento estrutural causado pela coexistência, no *Shared*, de utilitários e da configuração de injeção de dependências. Os utilitários não necessitam de conhecer os restantes módulos, enquanto a injeção de dependências exige uma dependência sobre eles, o que comprometia o princípio de isolamento das dependências. Para resolver estas limitações, como demonstrado na Fig. 1, foram introduzidos dois novos módulos: *App*, responsável pela inicialização da aplicação, e *DI*, dedicado à configuração da injeção de dependências. Esta reformulação permitiu isolar a lógica partilhada no *Shared* e centralizar a gestão de dependências no *DI*, eliminando problemas de dependências circulares.

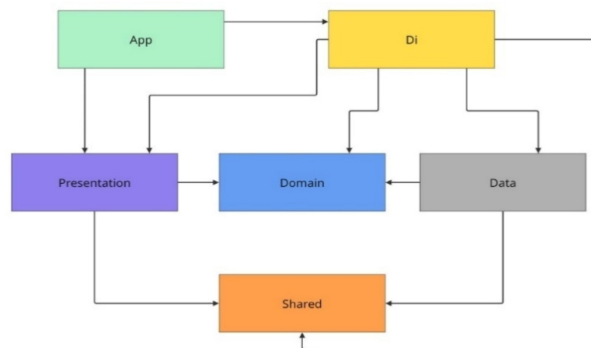

Fig. 1. *Arquitetura do projeto*

4 Implementação da Solução

4.1 Descrição da implementação

A aplicação foi implementada com base numa arquitetura modular, com o objetivo de garantir uma separação clara de responsabilidades, promover a escalabilidade e facilitar a manutenção do sistema. O fluxo de dados e as interações entre os módulos podem ser visualizados na Fig. 2.

A disponibilização de dados inicia-se no módulo *Data*, conforme a Fig. 2, os *Data Sources* comunicam com a base de dados através da API, transformando os dados com *mappers* e encapsulando-os em *Data Transfer Objects* (DTOs). A camada de repositório executa operações de forma assíncrona através de *coroutines*, evitando bloqueios na *main thread* e tratando erros com objetos *Result*.

No módulo *Domain* encontram-se definidos os modelos de domínio, as interfaces de repositório e os casos de uso, que servem de intermediários entre o domínio e a camada de apresentação. Os casos de uso executam operações de forma assíncrona e devolvem os resultados aos *ViewModels*, evidenciado na Fig. 2, promovendo a reutilização do código e centralizando a lógica de negócio, sem impacto nas restantes camadas.

O módulo Presentation gere a interface gráfica seguindo o padrão *MVVM* e o paradigma *Unidirectional Data Flow* (UDF). Conforme detalhado na Fig. 2, cada ecrã é controlado por um *ViewModel*, responsável pela gestão de múltiplos *ViewStates* (como carregamento, sucesso ou erro) e pela execução dos casos de uso através de *coroutines*. A interface foi desenvolvida com Jetpack Compose, adotando *state hoisting* para separar lógica e renderização através de

composables stateful e *stateless*. A aplicação suporta ainda atualizações em tempo real, asseguradas por um ouvinte que deteta alterações na base de dados e atualiza automaticamente o estado apresentado ao utilizador.

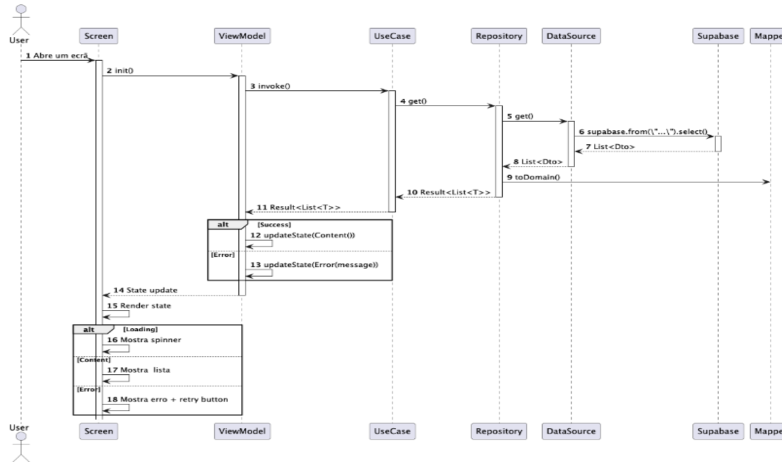

Fig. 2. *Vista de Processo Nível 3*

O módulo *DI* centraliza a gestão de dependências com Koin, integrando os módulos de dados, domínio, apresentação e utilitários. Esta configuração garante a disponibilização consistente dos componentes ao longo do ciclo de vida da aplicação, promovendo modularidade, testabilidade e facilidade de manutenção.

O módulo *Shared* agrega recursos comuns, como *dispatchers* para operações concorrentes, suportando tarefas de I/O, processamento intensivo e execução na *main thread*, sem conhecer os restantes módulos, o que reforça a sua independência.

O módulo *App* representa o ponto de entrada da aplicação, sendo responsável pela inicialização de toda a infraestrutura base, ativação do sistema de injeção de dependências e definição da interface gráfica com *Jetpack Compose*.

Inicialmente, optou-se pela utilização da biblioteca *Voyager* para suportar *deeplinks* e gestos nativos. Contudo, a sua rigidez motivou a adoção da navegação nativa do Jetpack Compose, recorrendo a *NavHostController* e *type tokens* para assegurar uma passagem de dados segura. Com a posterior integração destas capacidades na biblioteca oficial da JetBrains, a solução adotada ficou consolidada.

4.2 Testes

Para validação do sistema foram realizados testes unitários, de integração e de interface (UI). Os testes unitários verificaram componentes isolados. Embora existam outras bibliotecas de *mocking* mencionadas compatíveis com KMP, optou-se por utilizar a *MockK*, devido à sua maior adoção e familiaridade na comunidade, o que exigiu a criação de testes específicos por plataforma para contornar as limitações fora da JVM, afetando a consistência. Os testes de integração asseguraram o funcionamento conjunto dos componentes, validando a comunicação entre camadas com dados simulados. Por fim, os testes de UI validaram a interação e apresentação da interface, com

tempos de resposta próximos de <100 ms e fluidez da UI ≳60 fps, avaliados através de ferramentas de *profiling* do IDE, adaptados por plataforma devido a limitações de compatibilidade.

A cobertura dos testes, indicador essencial da qualidade da validação, situou-se acima dos 70% nos principais módulos o módulo *Data* atingiu 75,7%, conforme ilustrado na Fig. 3; o módulo *Domain* alcançou 78,2%, como mostrado na Fig. 4; e o módulo *Presentation* registou 73,2%, segundo a Fig. 5. Apesar de a cobertura elevada não garantir ausência de erros, aumenta a confiança e reduz regressões, limitada pela falta de testes multiplataforma.

data: Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	75.7% (48/64)	72.2% (85/118)	65.3% (139/212)	68.4% (1668/2439)	62.1% (4627/7451)

Fig. 3. *Cobertura do Módulo Data*

domain: Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	78.2% (20/26)	74.8% (25/33)		61.1% (87/143)	57.1% (397/696)

Fig. 4. *Cobertura do Módulo Domain*

presentation: Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	73.2% (136/186)	64.8% (219/337)	49.6% (275/554)	59.2% (1371/2317)	54.9% (12348/22492)

Fig. 5. *Cobertura do Módulo Presentation*

4.3 Avaliação da solução

A avaliação da solução baseou-se na realização de testes unitários, de integração e de UI, garantindo o funcionamento correto e a conformidade com os requisitos. Estes testes permitiram identificar erros precocemente e assegurar uma experiência de utilização consistente. Complementarmente, a aplicação foi avaliada pelo supervisor, Rui Almeida e *stakeholder* do projeto, Pedro Seruca, através de um questionário de usabilidade numa escala de 0-5, cujos resultados, detalhados na Tabela 4, foram positivos, reforçando a confiança na solução e apontando oportunidades para melhorias.

Tabela 4. *Resultados Questionário*

	Pedro Seruca	**Rui Almeida**
Os dados são carregados corretamente?	5	5
A UI é consistente com o design?	4	4
As pesquisas funcionam corretamente?	5	5
A app funciona em diferentes dispositivos/sistemas?	5	5
O desempenho é consistente em listas longas?	5	5
A navegação é consistente nas plataformas nativas?	4	4
A navegação é confortável com uma mão? (mobile)	5	5
A aplicação é estável e livre de bugs?	4	4
Informação fácil de encontrar?	5	5

5 Vantagens e Limitações

5.1 Vantagens

Destaca-se a partilha eficiente da lógica de negócio entre várias plataformas a partir de uma única base de código, o que reduziu esforço e facilitou a manutenção. A reutilização das interfaces garantiu consistência visual e comportamental, enquanto a arquitetura modular garantiu escalabilidade ao projeto. A integração de novas plataformas torna-se facilitada, e as funcionalidades nativas foram asseguradas, suportadas por uma comunidade ativa e boa documentação.

5.2 Limitações

Foram identificadas limitações relevantes, como a incompatibilidade da biblioteca de *mocks* com KMP/CMP, que exigiu testes unitários separados por plataforma, e a ausência de suporte multiplataforma para testes de UI. Inicialmente, a navegação nativa não suportava gestos e *deeplinks* no iOS, embora tenha sido corrigida em versões posteriores. Por fim, algumas bibliotecas, como a de navegação para desktop, apresentaram instabilidade.

6 Conclusões

Foi possível concluir que o projeto atingiu todos os objetivos inicialmente propostos.

Durante o desenvolvimento, enfrentaram-se algumas limitações, como atrasos na entrega de designs por parte da equipa de design, que condicionaram a implementação visual e funcional de certos componentes. As tecnologias adotadas apresentaram também restrições, nomeadamente a ausência de suporte direto para testes unitários e de UI multiplataforma, contrariando o objetivo de manter uma base de código totalmente comum.

Apesar destes constrangimentos, o projeto representa uma base promissora para futuros desenvolvimentos. A aplicação criada tem potencial de expansão, com adoção inicial prevista para a seleção nacional de rugby e, futuramente, para todo o panorama da modalidade em Portugal.

Agradecimentos. Agradeço ao ISEP e aos seus docentes, em especial ao Professor Eng. Paulo Proença, pelo acompanhamento ao longo da licenciatura e deste projeto. À Mindera, agradeço a oportunidade de estágio curricular e ao Eng. Rui Almeida pelo suporte técnico essencial.

Bibliografia

1. Stanic, N., Ćirković, S.: Analysis of Approaches to Developing Kotlin Multiplatform Applications and Their Impact on Software Engineering. 53–59 (2024).
2. Six Nations Rugby: Six Nations Mobile App. https://www.sixnationsrugby.com/en/app, last accessed 2025/04/22
3. Federazione Italiana Rugby: Federazione Italiana Rugby - Official App. Mac App Store. https://apps.apple.com/mt/app/federazione-italiana-rugby/id1074242258, last accessed 2025/04/22

4. Weninger, M.: Tracing Performance Metrics in Kotlin Multiplatform Projects. Master's thesis, Austria (2024)
5. Mir, S.: Kotlin Multiplatform - The Future of Cross-Platform Development. Medium (2024). https://medium.com/@mrsaykatm4/kotlin-multiplatform-the-future-of-cross-platform-development-5b95a548879c, last accessed 2025/03/21
6. Exyte: Jetpack Compose Multiplatform Android & iOS. ProAndroidDev (2023). https://proandroiddev.com/jetpack-compose-multiplatform-android-ios-4a87ba417caa, last accessed 2025/03/29
7. Kella, S.: Skia in Jetpack Compose: The Core Graphics Engine. Medium (2024). https://medium.com/@sandeepkella23/skia-in-jetpack-compose-the-core-graphics-engine-77e4f34b6695, last accessed 2025/03/25
8. Aripirala, S.S., Airan, P., Peddint, D.R.: The Polyglot's Playground: Navigating KMP, Flutter, and React Native in Cross-Platform Ecosystems. International Journal for Research in Applied Science and Engineering Technology 12 (2024)
9. Skip Tools: Skip and Kotlin Multiplatform (2024). https://skip.tools/blog/skip-and-kotlin-multiplatform/#caveats, last accessed 2025/03/14
10. Skip Tools: Skip Comparison Matrix. https://skip.tools/compare/, last accessed 2025/03/15
11. Emerline: React Native vs Flutter: Which Is Better for App Development? (2023). https://emerline.com/blog/flutter-vs-react-native, last accessed 2025/03/16
12. Android Developers: Dependency injection in Android. https://developer.android.com/training/dependency-injection, last accessed 2025/03/31
13. Chiriac, A.L.: Development of a multi-platform application for the University Virtual Campus. Bachelor's thesis, Universitat Oberta de Catalunya, Barcelona, Spain (2025)
14. Koin: Why Koin? https://insert-koin.io/docs/setup/why/#koin-a-dependency-injection-framework, last accessed 2025/05/10
15. Porciúncula, F.: From Dagger & Hilt into the multiplatform world with kotlin-inject (2023)
16. Mockative: Mocking for Kotlin/Native and Kotlin Multiplatform using KSP. GitHub (2025). https://github.com/mockative/mockative
17. Lupuuss: Mokkery - Mocking library for Kotlin Multiplatform. GitHub. https://github.com/lupuuss/Mokkery, last accessed 2025/11/24
18. Amanuel, A.: Supabase vs Firebase: Evaluation of performance and development of Progressive Web Apps. Master's thesis, Metropolia University of Applied Sciences, Finland (2022)
19. GeeksForGeeks: Functional vs. Non-Functional Requirements. https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/, last accessed 2025/04/14
20. Martin, R.C.: Clean Architecture: A Craftsman's Guide to Software. Prentice Hall (2018)
21. Sczip, J.G.B.: Clean Architecture - A Little Introduction. Medium (2020). https://medium.com/swlh/clean-architecture-a-little-introduction-be3eac94c5d1, last accessed 2025/06/22